For DOS Systems

3½" disk version

Microsoft® QuickBASIC™

Modern Programming System

Microsoft® QuickBASIC

Learning to Use Microsoft QuickBASIC

Programming in BASIC

Version 4.5
For IBM® Personal Computers and Compatibles

Microsoft Corporation

Information in this document is subject to change without notice and does not represent a commitment on the part of Microsoft Corporation. The software described in this document is furnished under a license agreement or nondisclosure agreement. The software may be used or copied only in accordance with the terms of the agreement. It is against the law to copy the software on any medium except as specifically allowed in the license or nondisclosure agreement. No part of this manual may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying and recording for any purpose without the express written permission of Microsoft.

RESTRICTED RIGHTS: Use, duplication, or disclosure by the U.S. Government is subject to restrictions as set forth in subparagraph (c) (1) (ii) of the Rights in Technical Data and Computer Software clause at DFARS 252.227-7013 or subparagraphs (c) (1) and (2) of Commercial Computer Software—Restricted Rights at 48 CFR 52.227-19, as applicable. Contractor/Manufacturer is Microsoft Corporation, One Microsoft Way, Redmond, Washington 98052-6399.

Copyright 1988, 1990 Microsoft Corporation. All rights reserved.

Simultaneously published in the U.S. and Canada.

Printed and bound in the United States of America.

Microsoft, MS, MS-DOS, and CodeView are registered trademarks and Windows is a trademark of Microsoft Corporation.

IBM is a registered trademark of International Business Machines Corporation.

Intel is a registered trademark of Intel Corporation.

Lotus is a registered trademark of Lotus Development Corporation.

WordStar is a registered trademark of MicroPro International Corporation.

Document No. DB14510-1190
OEM D707-4Z

Microsoft® QuickBASIC

Learning to Use Microsoft QuickBASIC

Table of Contents

Introduction . xxv

Instant Feedback xxv
Instant Help xxv
Hardware Requirements xxvi
The QuickBASIC Package xxvi
Printed Documentation xxvii
On-Line Documentation xxviii
Document Conventions xxix
Requesting Assistance xxxi

PART 1 Getting Started

Chapter 1 Setting Up Microsoft® QuickBASIC 5

The Setup Program 5
The Setup Main Menu 6
Installation 6
QB Express 7
Getting Into QuickBASIC 7
If You Are a Novice Programmer 7
If You Already Know BASIC 8
If You Use a Mouse 8

Chapter 2 Using Menus and Commands 9

Starting QuickBASIC 9

The QuickBASIC Environment 10

The Menu Bar 10

The Reference Bar 11

Opening Menus 11

QuickBASIC's On-Line Help 12

Using On-Line Help 12

The Help Menu 12

Choosing Commands 14

Using Dialog Boxes 15

Anatomy of a Dialog Box 15

Displaying a Dialog Box 16

Other Dialog-Box Features 17

Issuing Commands Directly 18

Shortcut Keys for Commands 19

Other Key Combinations 19

Exiting from QuickBASIC 20

For More Information 20

Chapter 3 QuickBASIC's Windows 23

Windows Available with Easy Menus 23

The View Window 25

Loading Programs in the View Window 25

Moving the Cursor 27

Scrolling Text 28

Starting a New Program 28

Changing Window Sizes 30

QuickBASIC's Other Windows . . . 30
Moving between Windows . . . 30
Executing Code in the Immediate Window . . . 31
Monitoring Variables with the Watch Window . . . 33
The Help Window . . . 34
Context-Sensitive Help . . . 34
Hyperlinks . . . 35
Exiting from QuickBASIC and Saving Programs . . . 37
For More Information . . . 38

Chapter 4 Interlude: BASIC for Beginners . . . 39
What is a Program? . . . 40
Comments . . . 40
Displaying Words and Numbers on the Screen . . . 40
Variables . . . 42
Storing Data in Memory . . . 42
Variable Types . . . 44
Integer Variables . . . 44
Floating-Point Variables . . . 44
String Variables . . . 45
Assigning Values to Variables . . . 45
Calculations . . . 47
Integer Division and the Remainder Operator . . . 47
Precedence of Operations . . . 48
Math Functions . . . 49
Expressions . . . 51
Displaying Variables and Expressions . . . 52
Entering Data with the INPUT Statement . . . 54

Arrays of Variables 55
Declaring Arrays 56
Specifying Array Elements 56
Logical Relations Used in Decision-Making 57
Relational Operators 57
Boolean Expressions 59
Compound Expressions 60
Using Logical Statements to Control Program Flow 61
Repeating Program Operations 62
The FOR...NEXT Loop 63
The DO...LOOP 65
The DO WHILE Loop 66
The DO UNTIL Loop 67
Writing a Simple BASIC Program 68
For More Information 69
Books about BASIC 69
Books about DOS 70

PART 2 Hands On with QuickBASIC

Chapter 5 The QCARDS Program 75
Building QCARDS 75
Loading a Program When You Start QuickBASIC 76
A Quick Tour of QCARDS 76
The QCARDS Program 77
Declarations and Definitions 78
Comments 78

Statements Following the END Statement . . . 78
Calling QCARDS Procedures from the Immediate Window . . . 79
Breaking an Unconditional Loop from QCARDS . . . 80
The Module-Level Code . . . 80
Structured Programming with Procedures . . . 80
A Profile of the Parts of the Program . . . 82
Defining a Procedure in QCARDS . . . 83
Saving Edited Text . . . 85
For More Information . . . 86

Chapter 6 Editing in the View Window . . . 87

The Smart Editor . . . 87
Automatic Formatting 88
Syntax Checking . . . 89
Errors Detected When You Run Your Program . . . 91
Help for Error Messages . 93
Overtyping the Error . . . 94
Automatic Procedure Declarations 94
Working with Selected Text Blocks . . . 95
Cutting or Copying Selected Text . . . 95
Pasting a Text Block . . . 97
Manipulating Selected Text 98
Searching and Replacing Text . . . 99
Defining the Symbolic Constant . . . 99
Replacing Multiple Occurrences of Text . 100
Checking Your Work . . . 102
For More Information . . . 103

Chapter 7 Programming with On-Line Help 105
Using On-Line Help to Construct Statements 105
On-Line Help for Keywords 105
Hyperlinks in On-Line Help 107
On-Line Help for Program Symbols 108
Printing Screens from On-Line Help 111
For More Information 113

Chapter 8 Using Example Code from On-Line Help 115
Copying Example Code from On-Line Help 115
Indenting a Block of Code 118
Copying Large Code Blocks from On-Line Help 118
Editing the Block Copied from On-Line Help 122
Finishing the QCARDS Code 125
Using QCARDS 126
For More Information 127

Chapter 9 Debugging While You Program 129
Debugging Commands 129
Debug-Menu Commands 130
Debugging Commands on the Run Menu 130
Function Keys Used in Debugging 131
Debugging a Procedure 131
Learning about Procedures 134
Continuing a Suspended Program 135
Isolating a Bug 136
Closing the Watch Window 140
Automatic Procedure Declarations 141
Creating a Stand-Alone Executable File 142

Learning about QuickBASIC's Other Menu Items 143
For More Information 143

PART 3 QuickBASIC Menus and Commands

Chapter 10 Getting Around in QuickBASIC 149

10.1 Starting QuickBASIC 150
10.1.1 The QB Command 150
10.1.2 The /NOHI Option 152
10.1.3 The QuickBASIC Screen 152
10.2 Opening Menus and Choosing Commands 155
10.2.1 Keyboard Technique 156
10.2.2 Using Shortcut Keys 157
10.3 Using Dialog Boxes 159
10.4 Using Windows 161
10.4.1 Window Types 162
10.4.2 Splitting the View Window (Full Menus Only) 163
10.4.3 Changing the Active Window 164
10.4.4 Changing Window Size 164
10.4.5 Scrolling in the Active Window 164
10.5 Using the Immediate Window 165
10.5.1 Statements Not Allowed 167
10.5.2 Doing Calculations 167
10.5.3 Testing Screen Output 167
10.5.4 Invoking Procedures 168
10.5.5 Changing the Values of Variables 168
10.5.6 Simulating Run-Time Errors 169
10.6 Using the Watch Window 170
10.7 Using the Mouse 172

10.8 Using On-Line Help 174
10.8.1 Help Features 174
10.8.2 Hyperlinks 175
10.8.3 Moving in Help Windows 176
10.8.4 Help Files 176
10.8.5 Hard-Disk System 177
10.8.6 Removable-Disk System 177

Chapter 11 The File Menu 179

11.1 New Program Command 180
11.2 Open Program Command 181
11.2.1 Specifying a File 181
11.2.2 Listing Directory Contents 182
11.3 The Merge Command (Full Menus Only) 183
11.4 Save Command (Full Menus Only) 184
11.5 Save As Command 185
11.6 Save All Command (Full Menus Only) 185
11.7 Create File Command (Full Menus Only) 186
11.8 Load File Command (Full Menus Only) 188
11.9 Unload File Command (Full Menus Only) 189
11.10 The Print Command 191
11.11 DOS Shell Command (Full Menus Only) 191
11.12 Exit Command 192

Chapter 12 Using the Editor 195

12.1 Entering Text 195
12.2 Selecting Text 196
12.3 Indenting text 196
12.4 Using Placemarkers in Text 197

12.5 The Smart Editor . . . 197
12.5.1 When Is the Smart Editor On? . . . 198
12.5.2 Automatic Syntax Checking . . . 198
12.5.3 Error Messages . . . 199
12.5.4 Automatic Formatting . . . 202
12.6 Entering Special Characters . . . 202
12.7 Summary of Editing Commands . . . 203

Chapter 13 The Edit Menu . . . 207
13.1 Understanding the Clipboard . . . 207
13.2 Undo Command (Full Menus Only) . . . 208
13.3 Cut Command . . . 208
13.4 Copy Command . . . 209
13.5 Paste Command . . . 209
13.6 Clear Command (Full Menus Only) . . . 210
13.7 New SUB Command (Full Menus Only) . . . 210
13.7.1 Creating a New SUB Procedure . . . 211
13.7.2 Default Data Types for Procedures . . . 211
13.7.3 Changing a Procedure's Default Type . . . 212
13.7.4 Saving and Naming Procedures . . . 213
13.8 New FUNCTION Command (Full Menus Only) . . . 213

Chapter 14 The View Menu . . . 215
14.1 SUBs Command . . . 216
14.2 Next SUB Command (Full Menus Only) . . . 218
14.3 Split Command (Full Menus Only) . . . 218
14.4 Next Statement Command (Full Menus Only) . . . 219
14.5 Output Screen Command . . . 219

14.6 Included File Command (Full Menus Only) . . . 219
14.6.1 Nesting Include Files . . . 220
14.6.2 Finding Include Files . . . 220
14.7 Included Lines Command . . . 221

Chapter 15 The Search Menu . . . 223
15.1 Defining Target Text . . . 223
15.2 Find Command . . . 224
15.3 Selected Text Command (Full Menus Only) . . . 226
15.4 Repeat Last Find Command (Full Menus Only) . . . 226
15.5 Change Command . . . 227
15.6 Label Command (Full Menus Only) . . . 229

Chapter 16 The Run Menu . . . 231
16.1 Start Command . . . 232
16.2 Restart Command . . . 232
16.3 Continue Command . . . 233
16.4 Modify COMMAND$ Command (Full Menus Only) . . . 233
16.5 Make EXE File Command . . . 234
16.5.1 Creating Executable Files . . . 234
16.5.2 Quick Libraries and Executable Files . . . 236
16.5.3 Types of Executable Files . . . 236
16.5.3.1 Programs that Use the Run-Time Module . . . 236
16.5.3.2 Stand-Alone Programs . . . 237
16.5.4 Run-Time Error Checking in Executable Files . . . 237
16.5.5 Floating-Point Arithmetic in Executable Files . . . 238
16.6 Make Library Command (Full Menus Only) . . . 239
16.6.1 Unloading and Loading Modules . . . 239
16.6.2 Creating Libraries . . . 240

16.7 Set Main Module Command (Full Menus Only) . . . 241
16.7.1 Changing the Main Module . . . 242
16.7.2 The .MAK File . . . 243

Chapter 17 Debugging Concepts and Procedures . . . 245
17.1 Debugging with QuickBASIC . . . 245
17.2 Preventing Bugs with QuickBASIC . . . 246
17.3 QuickBASIC's Debugging Features . . . 247
17.3.1 Tracing (Full Menus Only) . . . 247
17.3.2 Breakpoints and Watchpoints . . . 248
17.3.3 Watch Expressions . . . 248
17.3.4 Watch Window . . . 249
17.3.5 Immediate Window . . . 249
17.3.6 Other Debugging Features . . . 250

Chapter 18 The Debug Menu . . . 253
18.1 Add Watch Command . . . 254
18.2 Instant Watch Command . . . 256
18.3 Watchpoint Command (Full Menus Only) . . . 257
18.4 Delete Watch Command . . . 258
18.5 Delete All Watch Command (Full Menus Only) . . . 259
18.6 Trace On Command (Full Menus Only) . . . 259
18.7 History On Command (Full Menus Only) . . . 260
18.7.1 History Back . . . 260
18.7.2 History Forward . . . 260
18.8 Toggle Breakpoint Command . . . 261
18.9 Clear All Breakpoints Command . . . 261
18.10 Break on Errors Command (Full Menus Only) . . . 262
18.11 Set Next Statement Command . . . 263

Chapter 19 The Calls Menu (Full Menus Only) 265
19.1 Using the Calls Menu 265
19.2 Active Procedures 267

Chapter 20 The Options Menu 269
20.1 Display Command 270
20.2 Set Paths Command 271
20.3 Right Mouse Command (Full Menus Only) 272
20.4 Syntax Checking Command(Full Menus Only) 273
20.5 Full Menus Command 274

Chapter 21 The Help Menu 275
21.1 Index Command 275
21.2 Contents Command 276
21.3 Topic Command 277
21.4 Help On Help Command 278

Glossary . 279

Index . 293

Common Questions . 309

Product Assistance Request . 321

Figures

Figure 2.1 QuickBASIC Invocation Screen 10
Figure 2.2 The File Menu 11
Figure 2.3 Help Dialog Box for the Help Menu 13
Figure 2.4 Open Program Dialog Box 15
Figure 2.5 Display Dialog Box 17
Figure 2.6 Help on Output Screen Command 19
Figure 3.1 Windows Available with Easy Menus 24
Figure 3.2 Open Program Dialog Box 26
Figure 3.3 Output Screen for Code Lines 29
Figure 3.4 Immediate Window Showing Lines Just Entered 32
Figure 3.5 Placing Variables in the Watch Window 33
Figure 3.6 QuickSCREEN for PRINT Statement 35
Figure 3.7 Example Screen for PRINT Statement 36
Figure 3.8 Save As Dialog Box 37
Figure 4.1 Addresses in Memory 42
Figure 4.2 Array Created by the BASIC Statement DIM Price (4) 56
Figure 5.1 QCARDS' Interface 77
Figure 5.2 Modules and Procedures 81
Figure 5.3 SUBs Dialog Box 82
Figure 5.4 Save As Dialog Box 85
Figure 6.1 Error-Message Dialog Box 90
Figure 6.2 Error Message 92
Figure 6.3 Help on Error Message 93
Figure 6.4 Find Dialog Box 97
Figure 6.5 Change Dialog Box 100
Figure 6.6 Change, Skip, Cancel Dialog Box 101

Figure 7.1	QuickSCREEN for IF...THEN...ELSE Statement	106
Figure 7.2	Symbol Help for CardNum	109
Figure 7.3	Instant Watch Dialog Box	110
Figure 7.4	Print Dialog Box (Help)	112
Figure 8.1	Example Screen for DO...LOOP Statement	116
Figure 8.2	QuickSCREEN for SELECT CASE Statement	119
Figure 8.3	Code Showing Call to AsciiKey Procedure	123
Figure 9.1	Selecting the SELECT CASE Choice$ Block	132
Figure 9.2	DirectionKey Procedure in View Window	133
Figure 9.3	Restart Program Error Message	136
Figure 9.4	Setting a Breakpoint	137
Figure 9.5	Symbol Help for Symbolic Constant HOME	138
Figure 9.6	Error in Naming Symbolic Constant for the END Key	139
Figure 9.7	Make EXE File Dialog Box	142
Figure 10.1	Top Half of QuickBASIC Invocation Screen	153
Figure 10.2	Bottom Half of QuickBASIC Invocation Screen	154
Figure 10.3	The File Menu	155
Figure 10.4	Load File Dialog Box	159
Figure 10.5	Display Dialog Box	160
Figure 10.6	QuickBASIC Screen with Five Windows Open	163
Figure 10.7	Immediate Window	165
Figure 10.8	Simulated Run-Time Error	170
Figure 10.9	Watch Window	171
Figure 10.10	Help on the PRINT Statement	175
Figure 10.11	Dialog Box for Missing Help File	176
Figure 11.1	Open Program Dialog Box	181
Figure 11.2	Merge Dialog Box	183

Figure 11.3 Save Dialog Box . . . 184
Figure 11.4 Create File Dialog Box . . . 186
Figure 11.5 Load File Dialog Box . . . 188
Figure 11.6 Unload File Dialog Box . . . 190
Figure 11.7 Print Dialog Box . . . 191
Figure 11.8 Exit Dialog Box . . . 192
Figure 12.1 Syntax Error Message . . . 200
Figure 12.2 Initial Help Screen on the OPEN Keyword . . . 201
Figure 13.1 New SUB Dialog Box . . . 211
Figure 13.2 New FUNCTION Dialog Box . . . 214
Figure 14.1 SUBs Dialog Box . . . 216
Figure 14.2 Move Dialog Box . . . 217
Figure 15.1 Find Dialog Box . . . 224
Figure 15.2 Change Dialog Box . . . 227
Figure 15.3 Change, Skip, Cancel Dialog Box . . . 228
Figure 15.4 Label Dialog Box . . . 229
Figure 16.1 Modify COMMAND$ Dialog Box . . . 234
Figure 16.2 Make EXE File Dialog Box . . . 235
Figure 16.3 Make Library Dialog Box . . . 240
Figure 16.4 Set Main Module Dialog Box . . . 242
Figure 18.1 Add Watch Dialog Box . . . 254
Figure 18.2 Instant Watch Dialog Box . . . 256
Figure 18.3 Watchpoint Dialog Box . . . 257
Figure 18.4 Delete Watch Dialog Box . . . 258
Figure 19.1 Sequence of Procedures on Calls Menu . . . 266
Figure 20.1 Display Dialog Box . . . 270
Figure 20.2 Set Paths Dialog Box . . . 272

Figure 20.3 Right Mouse Dialog Box 273
Figure 21.1 Index Entries for Help 276
Figure 21.2 Table of Contents for Help 276
Figure 21.3 Help on the FOR...NEXT Statement 277
Figure 21.4 Help on Help Screen 278

Tables

Table 10.1	QuickBASIC Shortcut Keys	157
Table 10.2	Scrolling	165
Table 10.3	Mouse Commands	173
Table 10.4	Help Keys	176
Table 12.1	QuickBASIC Indentation Controls	196
Table 12.2	Editing Commands	204
Table 15.1	Search Options	225
Table 15.2	Search Restrictions	225
Table 17.1	Additional Debugging Features	250

QCARDS Code Entries

QCARDS Code Entry 1 83
QCARDS Code Entry 2 85
QCARDS Code Entry 3 88
QCARDS Code Entry 4 89
QCARDS Code Entry 5 90
QCARDS Code Entry 6 94
QCARDS Code Entry 7 94
QCARDS Code Entry 8 95
QCARDS Code Entry 9 96
QCARDS Code Entry 10 97
QCARDS Code Entry 11 98
Optional QCARDS Code Entry 1 99
Optional QCARDS Code Entry 2 100
Optional QCARDS Code Entry 3 101
QCARDS Code Entry 12 105
QCARDS Code Entry 13 107
QCARDS Code Entry 14 111
QCARDS Code Entry 15 112
QCARDS Code Entry 16 116
QCARDS Code Entry 17 117
QCARDS Code Entry 18 119
QCARDS Code Entry 19 120
QCARDS Code Entry 20 122
QCARDS Code Entry 21 122
QCARDS Code Entry 22 124
Optional QCARDS Code Entry 4 125

QCARDS Code Entry 23 125
QCARDS Code Entry 24 132
QCARDS Code Entry 25 133
QCARDS Code Entry 26 136
QCARDS Code Entry 27 139
QCARDS Code Entry 28 140
QCARDS Code Entry 29 141
QCARDS Code Entry 30 141

Introduction

Microsoft® QuickBASIC is a programming environment that includes all the tools you need for writing, editing, running, and debugging programs. These tools are integrated with a powerful version of the BASIC programming language and an on-line help system that explains everything about both the environment and the language.

Instant Feedback

Microsoft QuickBASIC speeds your programming and learning by giving virtually instant feedback for your ideas. When you write a program, you enter "code" (sequences of QuickBASIC statements) that describe what you want the program to do. QuickBASIC checks the validity of each line as you enter it, then immediately translates your code into a form the computer can execute. If your code contains errors that make it impossible to translate, QuickBASIC specifies the error and helps you correct it. As soon as your code is correct, you can press a key and immediately run it. If your code doesn't run as you expect, you can use QuickBASIC's sophisticated debugging tools to track and correct flaws in your code's logic. You get the speed and power of a compiled language without the tedious cycles of separate editing, compiling, running, and debugging.

Instant Help

Microsoft QuickBASIC's on-line help system includes two parts. The Microsoft QB Advisor contains explanations and runnable examples of every statement in the QuickBASIC language. Also, you can get on-line help for every aspect of the QuickBASIC environment, all symbols you define in your program, and error messages.

In this manual you will see how to get information on the following with on-line help:

- QuickBASIC statements and functions
- Menu and dialog-box items and options
- Special keys and key combinations for editing and debugging
- How to correct errors in your program
- Symbols you define in your programs

Hardware Requirements

Microsoft QuickBASIC requires an IBM Personal Computer or IBM®-PC compatible with at least 384 kilobytes (K) of available memory and a minimum of 720K disk-drive capacity. A hard disk and 640K of memory are recommended for best performance.

The QuickBASIC environment fully supports any mouse that is compatible with the Microsoft Mouse. See Section 10.7 for complete information on using your mouse with QuickBASIC.

NOTE *Throughout this manual, the term "DOS" refers to both the MS-DOS® and IBM Personal Computer DOS operating systems. The name of a specific operating system is used when it is necessary to note features that are unique to that system.*

The QuickBASIC Package

The Microsoft QuickBASIC package includes this document, an extensive on-line help system, example programs, and other on-line documentation. The next sections describe the contents of the package.

Printed Documentation

This document includes two manuals: *Learning to Use Microsoft QuickBASIC* and *Programming in BASIC*. These manuals are bound together and each has its own table of contents and index. Pages are numbered sequentially from the beginning of each manual.

Learning to Use Microsoft QuickBASIC discusses the QuickBASIC programming environment. The first two parts are a tutorial on how to use QuickBASIC. The last part is a reference on QuickBASIC menus and commands. The following list describes these parts in more detail:

Part	Content
"Getting Started"	General description of how to use QuickBASIC and a brief introduction to BASIC programming.
"Hands On with QuickBASIC"	Tutorial introduction to the Easy-Menus version of the QuickBASIC environment. The tutorial gives you practice with editing, debugging, and on-line help and guides you in creating your first QuickBASIC application, QCARDS.BAS.
"QuickBASIC Menus and Commands"	Reference to the QuickBASIC environment that presents detailed information on each command on QuickBASIC's Full Menus. Full Menus provide the complete multiple-module functionality of QuickBASIC. Use this section to extend your knowledge after you feel comfortable with the fundamentals of QuickBASIC.
"Glossary"	Definitions of terms used in the documentation.

Learning to Use Microsoft QuickBASIC provides the following information that you'll need to begin programming with QuickBASIC:

- The QuickBASIC environment
- The QuickBASIC implementation of the BASIC language
- The principles of programming

If you are new to programming, new to the BASIC language, or both, you will need to learn the environment, the QuickBASIC version of the BASIC language, and general programming skills. You should begin with Chapters 1–4 in Part 1, "Getting Started." If you already know how to program in BASIC, you should be able to get right to programming after reading Chapters 1–3 in Part 1.

In the tutorial in Part 2, "Hands On with QuickBASIC," you will practice using some of the QuickBASIC environment's most exciting features. You'll use the sophisticated editor, on-line help, and debugging features. When you finish the tutorial, you will have built a useful database application program, QCARDS.BAS.

This document includes another manual, *Programming in BASIC*, which discusses the following:

- Specific, practical programming topics, each illustrated by extensive programming examples
- The action and syntax of each QuickBASIC statement and function with tables listing the statements and functions by task groups
- Information on topics such as converting interpreted BASIC programs to run in QuickBASIC, changes in Version 4.5 from previous versions of QuickBASIC, and compiling and linking from DOS

Use *Programming in BASIC* for in-depth information on programming topics and a synopsis of the BASIC language.

On-Line Documentation

Microsoft QuickBASIC includes comprehensive on-line help and several disk files that provide supplementary information and programming examples. These are described in the following list:

Documentation	Purpose
On-Line Help	All information needed for programming in QuickBASIC. When you use help commands within the environment, QuickBASIC searches several files to provide the information. The file QB45QCK.HLP includes summaries and syntax descriptions of each QuickBASIC statement and function. The file QB45ADVR.HLP contains comprehensive information on using each statement and function, including example code you can copy and run. The file QB45ENER.HLP has explanations of all environment menus, commands, dialog boxes, and error messages.

README.DOC	A file describing changes to QuickBASIC that occurred after the manuals were printed.
PACKING.LST	A file describing each file on the QuickBASIC distribution disks.
LEARN.COM	Microsoft QB Express, a computer-based training program that gives an overview of QuickBASIC.
Sample programs	All the programs from *Programming in BASIC*. Check the file called PACKING.LST to locate them. Other sample programs are also included. Many demonstrate advanced programming techniques and algorithms.

Document Conventions

This manual uses the following document conventions:

Example of Convention	Description
QB.LIB, ADD.EXE, COPY, LINK, /X	Uppercase (capital) letters indicate file names and DOS-level commands. Uppercase is also used for command-line options (unless the application accepts only lowercase).
SUB, IF, LOOP, PRINT, WHILE, TIME$	Bold capital letters indicate language-specific keywords with special meaning to Microsoft BASIC. Keywords are a required part of 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 can use uppercase letters or lowercase letters.
`CALL Proc(arg1!, var2%)`	This kind of type is used for program examples, program output, words you should type in, and error messages within the text.

```
CONST FALSE = 0
  .
  .
  .
CHAIN "PROG1"
END
```

A column of three dots indicates that part of the example program has been intentionally omitted.

```
' Make one pass
```

The apostrophe (single right quotation mark) marks the beginning of a comment in sample programs.

filespec

Italic letters indicate placeholders for information, such as a file name, that you must supply. Italics are also occasionally used in the text for emphasis.

[[*optional-item*]]

Items inside double square brackets are optional.

ALT+F1

Small capital letters are used for the names of keys and key sequences, such as ENTER and CTRL+R.

A plus (+) indicates a combination of keys. For example, CTRL+E means to hold down the CTRL key while pressing the E key.

The carriage-return key, sometimes marked with a bent arrow, is referred to as ENTER.

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

The key names used in this manual correspond to the names on the IBM Personal Computer keys. Other machines may use different names.

"defined term"	Quotation marks usually indicate a new term defined in the text.
Video Graphics Array (VGA)	Acronyms are usually spelled out the first time they are used.

The following syntax block shows how some of these document conventions are used in this manual:

GET [[#]] *filenumber* [[,[[*recordnumber*]][,*variable*]]]

.

.

.

PUT [[#]] *filenumber* [[,[[*recordnumber*]][,*variable*]]]

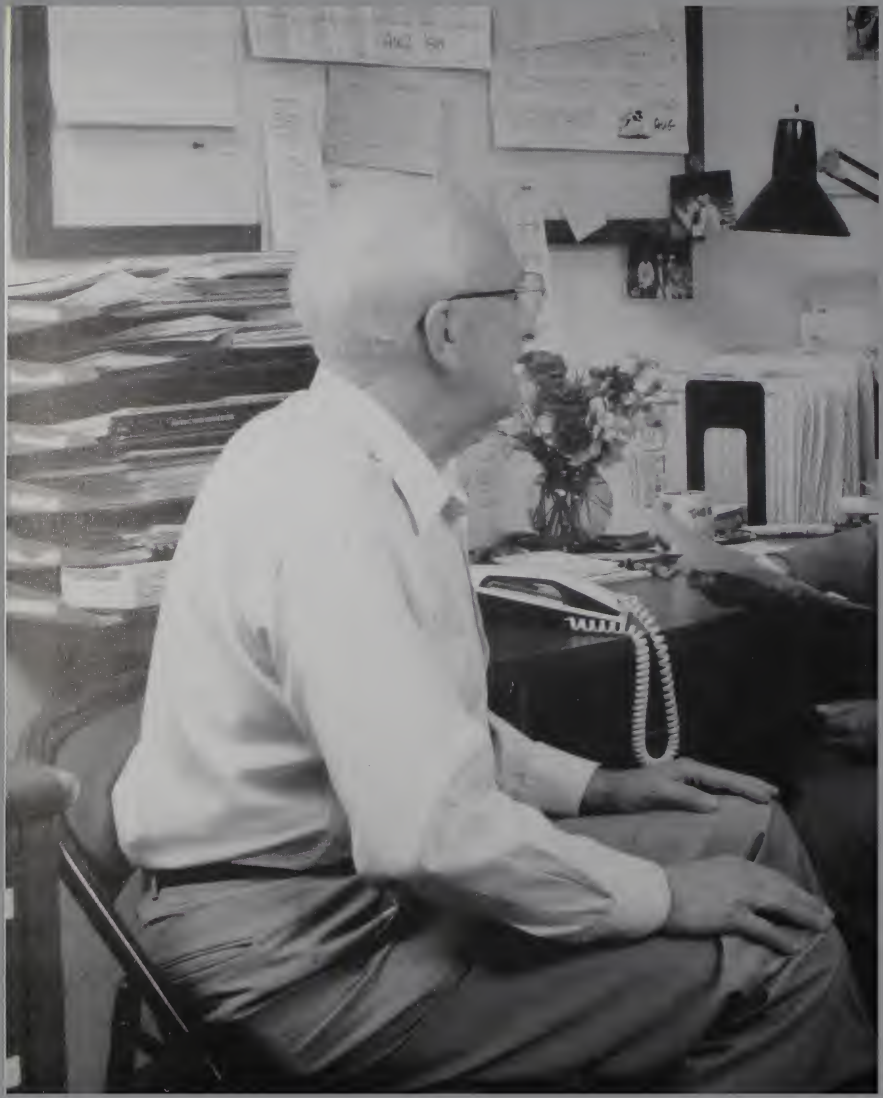
Requesting Assistance

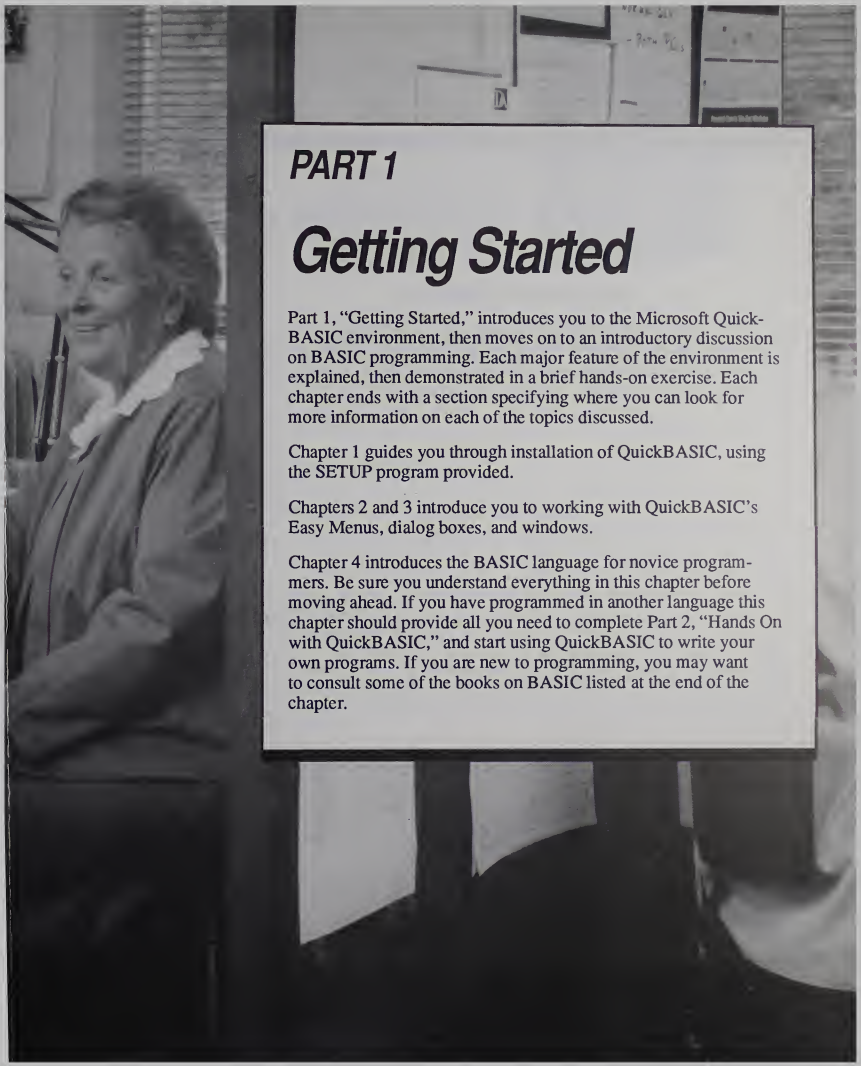
If you feel you have discovered a problem in the software, please read the section "Common Questions" in *Learning to Use Microsoft QuickBASIC*. That section discusses some of the most frequently asked questions about QuickBASIC. If your problem is not discussed in "Common Questions," report the problem to Microsoft. The section "Product Assistance Request" at the end of *Learning to Use Microsoft QuickBASIC* tells how to contact Microsoft to report a problem or receive assistance.

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

PART 1

Getting Started





PART 1

Getting Started

Part 1, "Getting Started," introduces you to the Microsoft QuickBASIC environment, then moves on to an introductory discussion on BASIC programming. Each major feature of the environment is explained, then demonstrated in a brief hands-on exercise. Each chapter ends with a section specifying where you can look for more information on each of the topics discussed.

Chapter 1 guides you through installation of QuickBASIC, using the SETUP program provided.

Chapters 2 and 3 introduce you to working with QuickBASIC's Easy Menus, dialog boxes, and windows.

Chapter 4 introduces the BASIC language for novice programmers. Be sure you understand everything in this chapter before moving ahead. If you have programmed in another language this chapter should provide all you need to complete Part 2, "Hands On with QuickBASIC," and start using QuickBASIC to write your own programs. If you are new to programming, you may want to consult some of the books on BASIC listed at the end of the chapter.

CHAPTERS

1 *Setting Up Microsoft® QuickBASIC* 5

2 *Using Menus and Commands* 9

3 *QuickBASIC's Windows* 23

4 *Interlude: BASIC for Beginners* 39

CHAPTER 1

Setting Up Microsoft® QuickBASIC

Your Microsoft QuickBASIC package includes the file SETUP.EXE (on the Setup/Microsoft QB Express disk). Since the files on the distribution disks are in a compressed format, you must use this program to install QuickBASIC. You cannot run QuickBASIC directly from the distribution disks.

This chapter discusses

- Installing Microsoft QuickBASIC
- Using Microsoft QB Express
- Exploring QuickBASIC, depending on your experience level

The Setup Program

If you do not have a hard disk, be sure you have five formatted, 360K removable disks (or an equivalent amount of space on disks of other sizes). If you have a hard disk, you need 1.8 megabytes of space available to install all QuickBASIC files. To start SETUP

➤ Place the Setup/Microsoft QB Express disk in your disk drive, and type `setup` at the DOS prompt.

Each screen in the SETUP program contains the following:

- An explanation of the current screen or the following screen.

- The Setup menu, which you use to choose what you want to do next. Generally, the Setup menu on each screen lets you continue, go back a screen, examine the options you've chosen, or return to the starting screen. To move within the menu and make choices, use the DIRECTION keys, then press ENTER when your choice is highlighted.
- A prompt at the bottom of the screen tells you how to move on to the next step.

NOTE *If you make a mistake when installing QuickBASIC with SETUP, you can go back to any previous point and start again from there. You can run SETUP as many times as you like.*

The Setup Main Menu

The items on the Setup Main menu allow you to choose the defaults for all options (Easy Setup) and customize the options you choose during setup (Full Setup).

If you are a novice programmer, you should choose Easy Setup. As you learn more about QuickBASIC, you can reset these options from within the environment without running SETUP again.

If you are an experienced programmer, Full Setup lets you tailor the appearance and certain behaviors of QuickBASIC. Chapter 20, "The Options Menu," describes how QuickBASIC handles the search paths and colors you choose during setup. (They correspond to the Options menu's Display and Set Paths commands.)

Whether you choose Easy or Full Setup, use the Show Options menu command to look at the final settings before completing the installation.

Installation

After you have decided about options, choose Perform Installation from the menu to begin automated installation. QuickBASIC prompts you for each disk you need to insert in your disk drive as installation progresses.

If you have a hard disk, QuickBASIC places the files you need in the directories specified in either the Easy Setup or Full Setup options. If the directories don't already exist, QuickBASIC creates the directories you specify on your hard disk.

If you do not have a hard disk, you must start the installation phase with five formatted removable disks. QuickBASIC prompts you when to install each disk. You might want to label the disks, using the same names as appear on the original disks.

QB Express

After installation is complete, return to the Main Setup menu. Even if you've used QuickBASIC before, you should choose Exit and Run QB Express from the main menu. Microsoft QB Express, a brief computer-based training program, introduces the QuickBASIC environment and gives you some practice in the things you'll do when you fire up QuickBASIC itself. If you've used QuickBASIC before, QB Express alerts you to Version 4.5 enhancements to the environment.

Getting Into QuickBASIC

QuickBASIC is a superb environment in which to learn programming, but it is not specifically designed as a "learn to program" environment. QuickBASIC provides all the tools you need for programming—editing, debugging, and on-line help—plus a powerful version of the BASIC language.

Chapters 2–9 of this book are a tutorial on QuickBASIC. A typewriter icon marks all the practice exercises.

If You Are a Novice Programmer

In learning to use QuickBASIC, you need to distinguish between the tools provided by the environment, and the language itself. The best way to acquaint yourself with the environment is by reading Chapter 2, "Using Menus and Commands," and Chapter 3, "QuickBASIC's Windows." Then, try Chapter 4, "Interlude: BASIC for Beginners," which should give you enough perspective on BASIC programming to complete Part 2, "Hands On with QuickBASIC."

Part 2, "Hands On with QuickBASIC," is a tutorial that introduces you to editing, using on-line help, and debugging in the context of an extended program example. In a series of exercises, you will complete the coding of QCARDS.BAS, a real application program. Each of these chapters takes one to two hours to complete.

Although "Hands On with QuickBASIC" is not a tutorial on programming, QCARDS.BAS provides a good model of program structure. You can also use the code from QCARDS in other programs you write. But remember, the purpose of the QCARDS tutorial is to teach you quickly how to use the tools in the QuickBASIC environment.

You don't have to understand all the programming techniques to type in the QCARDS code. To learn more about programming, read some of the BASIC programming texts listed at the end of Chapter 4, "Interlude: BASIC for Beginners." *Programming in BASIC*, included with this package, presents extended examples and explanations of selected programming topics.

If You Already Know BASIC

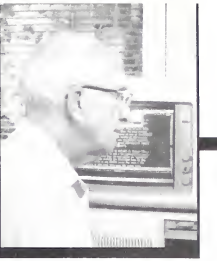
If you are already familiar with BASIC, you should be able to move quickly through Chapter 2, "Using Menus and Commands," and Chapter 3, "QuickBASIC's Windows," which teach the fundamentals of menus and windows, and opening, clearing, and saving files. You can skip Chapter 4 completely and move right to Part 2, "Hands On with QuickBASIC," to learn the use of QuickBASIC's advanced editing, on-line help, and debugging features. Even if you have used a previous version of QuickBASIC, you may find some new programming techniques in QCARDS.BAS, the example program you will build as you practice advanced environment features. The comments in the QCARDS code explain how the program works. If the comments do not provide enough information, the programming techniques are discussed in *Programming in BASIC*.

If You Use a Mouse

QuickBASIC fully supports pointing devices that are compatible with the Microsoft Mouse. See Section 10.7 for complete information on using QuickBASIC with a mouse.

CHAPTER 2

Using Menus and Commands



This chapter teaches you how to start Microsoft QuickBASIC and use the Easy Menus. A "menu" is a list of commands. In QuickBASIC, you can choose between two sets of menus—Easy Menus and Full Menus. Easy Menus, which you'll work with here, contain all the commands a novice programmer needs to use QuickBASIC. Part 3, "QuickBASIC Menus and Commands" describes the advanced commands on the Full Menus.

When you have completed this chapter, you'll know how to

- Open and use menus and dialog boxes
- Consult QuickBASIC's on-line help to answer questions
- Use function keys and shortcut-key combinations

This chapter will take one to one and one-half hours to complete.

Starting QuickBASIC

Start QuickBASIC now:

1. Type `qb` at the DOS prompt. The QuickBASIC screen appears.
2. Press ENTER to display essential information about the QuickBASIC environment. You can use PGUP and PGDN to move around in this text.
3. Press ESC to clear this information from the screen.

NOTE *If your computer has a monochrome or liquid-crystal display, or if you use a Color Graphics Adapter (CGA) with a black-and-white monitor, see Sections 10.1.1, "The QB Command," and 10.1.2, "The /NOHI Option," for additional QB invocation options.*

If you start QuickBASIC with the `/nohi` *or* `/b /nohi` *option, what are referred to as high-intensity letters in this manual may have a different appearance.*

The QuickBASIC Environment

QuickBASIC's programming tools are instantly available. They include features for program organization, editing, syntax checking, file management, printing, and debugging—even customizing the colors of the environment.

The Menu Bar

The menus on the menu bar at the top of the QuickBASIC screen contain the QuickBASIC environment commands (see Figure 2.1). You'll use them as you write and modify programs in the View window.

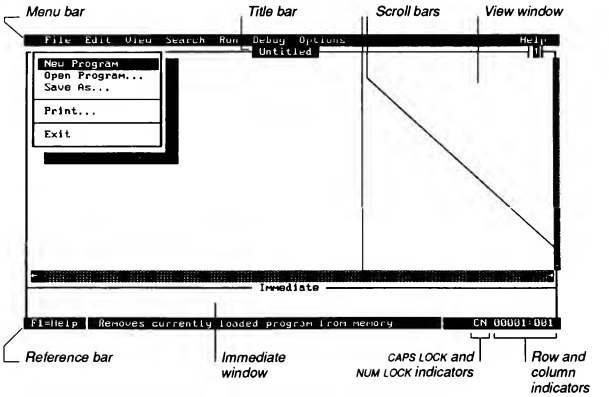


Figure 2.1 QuickBASIC Invocation Screen

The Reference Bar

The reference bar at the bottom of the screen tells which keys to press to perform important actions (see Figure 2.1). As you work on a program, the reference bar updates information about the environment and your program. When a command is highlighted, the reference bar briefly describes how to use the command. Row and column counters indicate where you are in your program, and CAPS LOCK and NUM LOCK indicators tell how your keyboard is set.

Opening Menus

You can execute most QuickBASIC commands by choosing them from the menus. When you press the ALT key, QuickBASIC highlights the file menu, and one letter in each menu name appears in high-intensity video. To open a menu, press the ALT key, then press the key corresponding to the high-intensity letter in the menu name. You can open the menus and move among them using the DIRECTION keys (the UP, DOWN, LEFT, and RIGHT arrow keys).

To open a menu

1. Press ALT to select the File menu.

   When you "select" (highlight) a menu, its name is shown in a reverse-video block. If you press ALT again, the menu name loses the highlight.

2. Press ENTER to open the File menu (see Figure 2.2).

Figure 2.2 The File Menu

- Press the UP or DOWN key to select successive commands in an open menu.
- Press the RIGHT or LEFT key to open adjacent menus.

3. Press ESC to close the menu without executing a command. The cursor returns to the View window, the part of the screen where you edit your program. You can always back out of any situation by pressing ESC.

QuickBASIC's On-Line Help

If you need additional information about menus or other QuickBASIC features, consult on-line help. You can get help on BASIC keywords, the elements of the language, the QuickBASIC environment, and error messages.

Using On-Line Help

There are several ways to get on-line help in QuickBASIC. For help on menu names or commands, select the name or command, then press F1. If you want help on any part of a program, place the cursor within the word, then press F1.

To move around in on-line help, do the following:

- Press PGDN to see additional screens.
- Press PGUP to move back through screens you have already seen.
- Press ESC to cancel on-line help.

If you need more information on using any on-line help function, hold down the SHIFT key while pressing F1 for a general description of on-line help.

The Help Menu

Most on-line help is available through the Help menu. Try this to use on-line help:

1. Press ALT to select the File menu without opening it.
2. Press LEFT to select the Help menu.

3. Press F1. On-line help displays the information shown in Figure 2.3 about the Help menu:

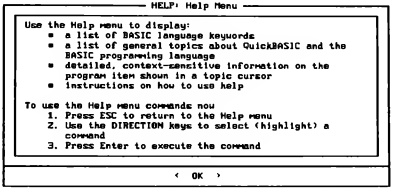


Figure 2.3 Help Dialog Box for the Help Menu

4. Press ENTER when you finish reading the text.
5. Press DOWN to open the Help menu.

You can get help with each of the items on the Help menu by moving the highlight among the commands with the DIRECTION keys, and pressing F1 for explanations.

If there is no on-line help available for the current item, or if the necessary help file is unavailable, QuickBASIC either displays a message, beeps, or both.

The Help menu has the following commands:

Command	Action
Index	Displays an alphabetical listing of all QuickBASIC keywords. Place the cursor on a keyword and press F1 to get help on that keyword.
Contents	Displays a visual outline of the on-line help's contents. For detailed information on any category listed, place the cursor on the category and press F1.
Topic	Displays help for any word at the cursor position. If there is a word at the cursor position, it appears in the Help menu following the Topic command. If there is no word at the cursor position, the Topic command is inactive.
Help on Help	Explains how to use on-line help.

Choosing Commands

To "choose" (execute) a command, select the command by pressing the UP or DOWN key, then press ENTER or the high-intensity letter in the command name. (If a command has no high-intensity letter, it is not currently available for you to use.)

If a command is followed by three dots (. . .), a dialog box (description to follow) appears when you choose the command. Commands not followed by dots execute immediately when you choose them.

To learn more about a command before choosing it, press F1 for on-line help. After you read the help information, press ESC to clear help, then press ENTER if you want to choose the command. Try the following steps:

1. Press ALT, then press O to select the Options menu.

   This key combination opens the Options menu. In this manual, combinations of keys are written as key names connected with a plus sign. The sequence above is usually shown as ALT+O.

2. Press DOWN until Full Menus is highlighted.

3. Press F1 to display on-line help while Full Menus is selected.

4. Press ESC to clear on-line help.

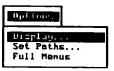
5. Press ENTER to turn off Full Menus if the bullet (•) is present. Press ESC to close the Options menu without executing any of its commands if Full Menus does not have a bullet.

   The Full Menus command is a "toggle." If it is on (that is, if it has a bullet), then choosing it turns it off. If it is off (no bullet), choosing it turns it on.

6. Press ALT+O to open the Options menu again. Make sure there is no bullet beside Full Menus, then press ESC to close the menu.

You only need to use DIRECTION keys when you want help with a command. If you don't need help, just choose commands by pressing the letter corresponding to the high-intensity letter in the command name.

Using Dialog Boxes



A "dialog box" appears when you choose any menu command followed by three dots (. . .). QuickBASIC uses dialog boxes to prompt you for information it needs before executing a command. Dialog boxes also appear when you

- Enter a line of code containing a syntax error
- Run a program that contains an error
- Choose the Help command button in a dialog box
- Request help for a menu or menu command
- Try to end a session without saving your work

Anatomy of a Dialog Box

Dialog boxes include several devices you can use to give QuickBASIC information (see Figure 2.4).

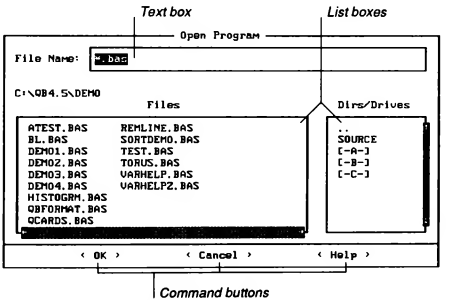


Figure 2.4 Open Program Dialog Box

The following list explains how to use these devices:

Device	Description
Text box	Type text, such as a file name, here
List box	Select one item from a group of listed items
Command button	Use to confirm entries, get help, or escape from the dialog box

Displaying a Dialog Box

When you first open a dialog box, it contains "default" settings—the settings people use most often. After that, when you display a dialog box, it reflects the choices you made the last time the box was displayed. The angle brackets on the default command button appear in high intensity. When you press ENTER, the default command button is chosen. You can use ALT key combinations or TAB to move around in a dialog box.

Follow these steps to open and examine the File menu's Open Program dialog box:

1. Press ESC to close the Options menu if it is still open.
2. Press ALT+F, then press O to choose the File menu's Open Program command.

   The Open Program dialog box (shown in Figure 2.4) appears. It lets you load programs from disk files into the View window.
3. Press F1 to choose the Help command button.

   A help dialog box opens on top of the Open Program dialog box. You can use DIRECTION keys to move through the help text.
4. Press ESC to clear the help dialog box and return the cursor to the Open Program dialog box.
5. Press ALT to turn on the high-intensity letters you can use in combination with the ALT key.
6. Press TAB to move the cursor into the Files list box.

   If there are numerous entries in the list box, you can move among them either by pressing DIRECTION keys or the initial letter of the item you wish to select.

7. Press TAB or ALT+D to move the cursor to the Dirs/Drives list box.
8. Press ESC to close the dialog box without opening a program.

Other Dialog-Box Features

Other dialog-box features include option buttons and check boxes. You'll use these features in the Options menu's Display dialog box (see Figure 2.5) to change options—such as screen colors, scroll bars, and the distance between tab stops—chosen when you installed QuickBASIC.

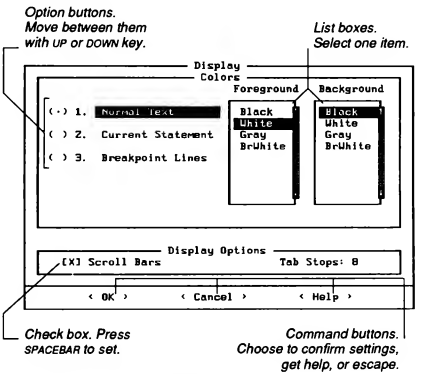


Figure 2.5 Display Dialog Box

Option buttons are turned on if a bullet (•) appears within the parentheses. Only one option in each group of option buttons can be turned on at a time. An X within the brackets indicates that a check box is turned on. You can turn on as many check boxes as you like.

To set options, do the following:

1. Press ALT+O, then press D to choose the Option menu's Display command. The Display dialog box (shown in Figure 2.5) appears.
2. Press ALT+T to move the cursor to the Tab Stops text box.

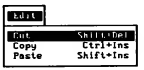
   Type a new number if you want to change the number of spaces between tab stops while you are editing. For programming, `4` is a good choice.
3. Press ALT+S to move the cursor to the Scroll Bars check box. Press SPACEBAR to put an `X` in an empty check box or remove the `X` if the box already has one. (Scroll bars, shown on Figure 2.1, are only useful if you have a mouse.)
4. Set the three types of foreground and background colors, as follows:
   a. Press ALT plus the highlighted number (`1`, `2`, or `3`) to select the option whose colors you want to change.
   b. Press TAB to move to the foreground and background test boxes and press UP or DOWN to select the colors.
   c. Repeat the previous process two steps to change the Current Statement and Breakpoint Lines options.
5. Press ENTER to choose the command, confirming any new choices you've just made, or press ESC to return all settings to what they were when the dialog box was first displayed.

The Current Statement option determines the attributes of the currently executing line in a running program. The Breakpoint Lines option sets the color for lines at which you set breakpoints. Note that Breakpoint Lines and Current Statement colors show only when you have a program running.

Issuing Commands Directly

You can use shortcut keys and key combinations to issue commands directly without opening menus to choose items. Shortcut keys are available for many QuickBASIC tasks.

Shortcut Keys for Commands



If a command has a shortcut key, it is listed beside the command. For example, instead of opening the Edit menu and choosing the Cut, Copy, or Paste commands, you can use SHIFT+DEL, CTRL+INS, and SHIFT+INS instead.

Other Key Combinations

There are other special keys and key combinations. For example, SHIFT+F1 always displays help on help; and F1 displays on-line help for menu names and commands or any keyword or program symbol at the cursor. Other key combinations use function keys, keys on the numeric keypad like INS and DEL, and many combinations of the SHIFT, ALT, or CTRL keys with other keys.

For example, you can use the RIGHT and LEFT keys to move the cursor right and left on the screen. To move the cursor to the next word beyond the current cursor position, hold down CTRL and press LEFT or RIGHT.

Try this exercise to see how common shortcut keys and key combinations work:

1. Press ALT+V to open the View menu.

   The SUBs command is automatically selected. Note the shortcut key (F2) to the right of the command name.

2. Press DOWN to select the Output Screen command. Note that its shortcut key is F4.

3. Press F1 to consult on-line help for an explanation of Output Screen (see Figure 2.6).

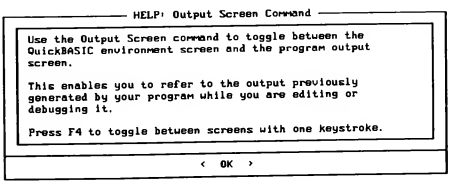


Figure 2.6 Help on Output Screen Command

4. Press ENTER to close the help screen. ENTER is the shortcut key for the default (or currently selected) command button.
5. Press ESC to close the View menu.

Exiting from QuickBASIC

When you end a session, QuickBASIC checks to see if you have any unsaved work in progress. If you do, a dialog box asks if you want to save your work. Because you haven't entered any text in this session, press N if this dialog box appears when you do the following:

➤ Press ALT+F, then press X to choose the File menu's Exit command to return to DOS.

If you made and confirmed changes in the Displays dialog box, they are saved in a file called QB.INI when you exit. They become the defaults the next time you start QuickBASIC.

For More Information

For more information on the topics covered in this chapter, see the following:

Where to Look	What You'll Find
Section 10.1.1, "The QB Command," and 10.1.2, "The /NOHI Option"	Explanations of QuickBASIC's invocation options
Section 10.8, "Using On-Line Help," and Chapter 21, "The Help Menu"	Complete description of on-line help

Section 10.2, "Opening Menus and Choosing Commands," and 10.3, "Using Dialog Boxes"	More information on menus and dialog boxes
Section 10.2.2, "Using Shortcut Keys"	A listing of the common shortcut keys
Section 12.7, "Summary of Editing Commands"	A complete list of key combinations used for editing

CHAPTER 3

QuickBASIC's Windows

In this chapter you will learn how to use Microsoft QuickBASIC's windows as you write a short program. You'll learn how to

- Open an existing program and move the cursor
- Activate, move between, and change the window size
- Execute BASIC statements from the Immediate window
- Monitor values of variables with the Watch window
- Use the Help window
- Save programs as files on disk

This chapter takes about one to one and one-half hours to complete.

Windows Available with Easy Menus

QuickBASIC is called a "window-oriented" environment because the screen is divided into separate areas in which you can perform different tasks. Some windows, such as the View window and the Help window, display material loaded

from disk files. Other windows, such as the Watch window, display information QuickBASIC has about the current program. With Easy Menus, the windows shown in Figure 3.1 are available.

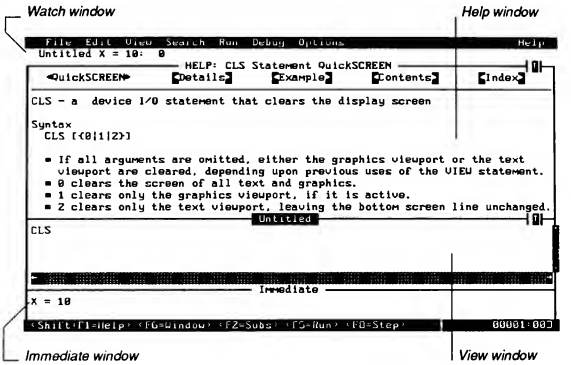


Figure 3.1 Windows Available with Easy Menus

Window	Description
View	The window in which you do most of your programming. When you start QuickBASIC the View window is the "active window," the one containing the editing cursor. When you load a disk file, its contents appear in the View window.
Immediate	The window at the bottom of the screen. In it you can use single lines of BASIC code to test programming ideas and perform miscellaneous calculations. You can type in the Immediate window, but you cannot load disk files into it.

Help	A window that opens at the top of the screen when you press F1 or SHIFT+F1. Information about QuickBASIC keywords, symbols you define in your program, and using help is displayed in the Help window.
Watch	A window that opens at the top of the screen when you choose certain commands from the Debug menu.

You can have several windows open at a time, but only one can be active at a time. By opening multiple windows, you can view different types of information simultaneously. For example, you can have program code in the View window, then display on-line help for the statement you want to use.

The View Window

The View window is where you do most of your programming. This section explains how to load an existing program from a disk file into the View window and move the cursor around in its text. Then you'll clear that program and write and run some lines of your own code.

Loading Programs in the View Window

When you load a file from a disk, its text appears in the View window. To load a program

1. Start QuickBASIC using the `QB` command you learned in Chapter 2.
2. Press ESC to clear the Welcome dialog box.

3. Press ALT+F, then press O to choose the File menu's Open Program command. QuickBASIC displays a dialog box like the one in Figure 3.2.

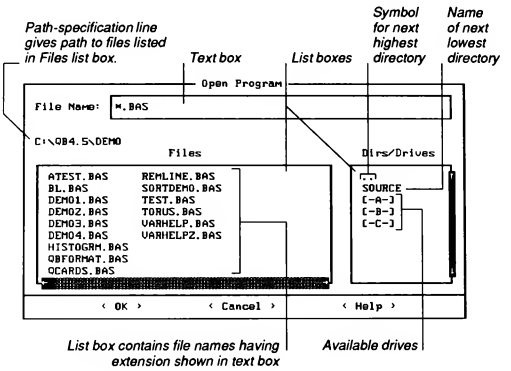


Figure 3.2 Open Program Dialog Box

The cursor is in the text box where `*.bas` is selected. Anything you type while `*.bas` is selected replaces it. The `*.bas` in the text box tells QuickBASIC to list all the files having that extension in the Files list box.

This dialog box has the following features:

Feature	Function
Text box	Box for typing text, such as a file name
Files list box	Box that displays files in the directory specified on the path-specification line
Dirs/Drives list box	Box that shows the directories above and below the directory specified on the path-specification line and the drives available on your system

The files in your list box should include QCARDS.BAS. To open it, type its name in the text box, or use the list box to select it, as described here:

1. Press TAB to move the cursor into the Files list box.
2. Press DIRECTION keys until the name of the program you want is selected. (Press SPACEBAR if the program you want is the first entry.)
3. Select QCARDS.BAS.

   The name QCARDS.BAS replaces *.bas in the text box.
4. Press ENTER to load QCARDS.BAS in the View window.

   QCARDS is large (almost 40K), so it may take a few moments to load. When it is loaded, the name QCARDS.BAS appears in the View window's title bar.

Moving the Cursor

You can use the DIRECTION keys on the numeric keypad (the LEFT, RIGHT, UP, and DOWN arrow keys as well as HOME, END, PGUP, and PGDN) to move the cursor in a window. The DIRECTION keys move the cursor a single row or column at a time. HOME and END move the cursor to the first or last column of the current line. However, you can combine the CTRL key with DIRECTION keys to move the cursor in greater increments, as follows:

Key Combination	Cursor Movement
CTRL+RIGHT	Right one word
CTRL+LEFT	Left one word
CTRL+HOME	First line of current window
CTRL+END	Last line of current window

Scrolling Text

A window can contain far more text than is visible within its on-screen boundaries. "Scrolling" describes the movement of text into and out of the bounded screen area. You can scroll text using the following keys and combinations:

Key Combination	Scrolling Action
PGUP	Up by one whole screen
PGDN	Down by one whole screen
CTRL+PGDN	Right by one whole screen
CTRL+PGUP	Left by one whole screen

NOTE *The CTRL+PGDN combination can be disorienting if you don't have text between columns 79–159. To confirm your place in a file, check the row and column counters in the lower right corner of the reference bar. You can press HOME to move the cursor to the first column of the current line.*

Starting a New Program

When you load an existing program from disk, QuickBASIC clears everything else from memory before loading the program. In this section, you'll clear QCARDS.BAS from QuickBASIC's memory with the New Program command, then you'll type in a simple program. As you type in lines of code, notice that QuickBASIC capitalizes the keywords in the statements when you press ENTER. If you get an error message, press ESC to clear the message, then retype the line at the cursor.

Follow these steps to clear QCARDS.BAS, then write and run some code:

1. Press ALT+F, then press N to choose the File menu's New Program command.
2. Type the following BASIC statements

```
cls
print "This is my first QuickBASIC program"
print 2 + 2
```

3. Press ALT+R, then press S to choose the Run menu's Start command.

   If you typed everything correctly, QuickBASIC displays the program's output on the "output screen," which should look like the screen in Figure 3.3.

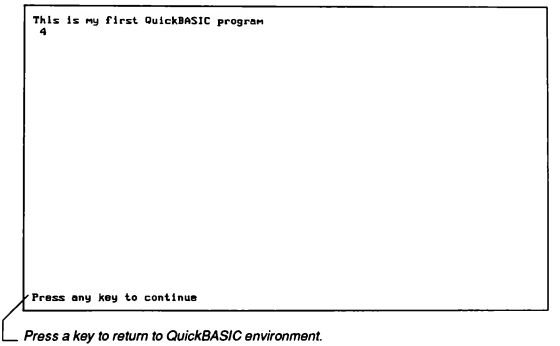


Figure 3.3 Output Screen for Code Lines

   In this program, CLS clears the screen, then the PRINT statement displays the quoted text in the upper left corner of the screen. Because PRINT can display the results of numeric expressions, the sum of the expression 2+2 appears on the next line.

4. Press a key to return to QuickBASIC's View window.

5. Press ALT+V, then press U to choose the View menu's Output Screen command to take another look at your program's results.

6. Press a key to return to QuickBASIC's View window.

NOTE *You can get help for any keywords you don't understand by placing the cursor on the keyword, then pressing F1.*

Changing Window Sizes

You can easily increase or decrease the size of the windows on your screen. To increase the size of the active window, press ALT+PLUS once for each line you want to add. To reduce the size by one line, press ALT+MINUS. (Use the PLUS and MINUS keys on the numeric keypad.)

To temporarily expand the active window to full screen, press CTRL+F10. You can restore the windows to their previous sizes by pressing this toggle again. The following steps illustrate this and prepare the windows for the exercise in the next section:

1. Press ALT+PLUS twice while the cursor is in the View window.

   This adds lines to the View window and compresses the Immediate window beneath it.

2. Press ALT+MINUS seven times to reduce the size of the View window.

   This decreases the number of lines in the View window while increasing the size of the Immediate window.

3. Press CTRL+F10 to expand the size of the current window temporarily to full-screen status.

4. Press CTRL+F10 again to restore the screen to its previous proportions.

QuickBASIC's Other Windows

When you start QuickBASIC and clear the opening dialog box, the View window contains the cursor, so it is the active window. Only one window can be active at a time. However, other windows are available. Note that the title bar of the active window is highlighted.

Moving between Windows

To move the cursor down one window, press the F6 key. To move the cursor up one window, press SHIFT+F6. If you have used CTRL+F10 to expand the current window, F6 and SHIFT+F6 still make the next and previous windows active.

Executing Code in the Immediate Window

The Immediate window at the bottom of the screen can be used as a utility window as you program in the View window. For example, if you want to clear the output screen, you can execute the CLS statement from the Immediate window. You can also use the Immediate window for doing calculations, or experimenting with programming ideas and previewing the results. When you exit from QuickBASIC, text in the Immediate window is lost.

When you run lines of code in the View window, they are executed sequentially. In the Immediate window, a single line of code is executed whenever you put the cursor anywhere on that line, then press ENTER.You can type in either uppercase or lowercase in the Immediate window. Note that QuickBASIC doesn't capitalize keywords in the Immediate window as it does in the View window.

Try the following procedure to execute lines of code in the Immediate window:

1. Press F6 to place the cursor in the Immediate window.
2. Type the following line:

   ```
   cls : print "This line comes from the Immediate window"
   ```

   This code consists of two BASIC statements separated by a colon. Separating statements with a colon allows you to put more than one statement on a line.
3. Press ENTER to execute the code on this line.

   QuickBASIC displays the output screen.
4. Press a key to return to the QuickBASIC environment. The cursor is placed on the next blank line in the Immediate window.
5. Type the following line, exactly as shown:

   ```
   CurrentTime$ = ""
   ```

   This defines a variable named `CurrentTime$`. The `$` tells QuickBASIC it is a "string" variable (one consisting of a series of characters). The quotation marks have no space between them, so the string has a length of zero.
6. Press ENTER to execute this statement and assign the zero-length string to `CurrentTime$`.

Next, you'll continue working in the Immediate window and use the BASIC **TIME$** function to assign a string representing the current time to the variable `CurrentTime$`:

1. Type the following line (exactly as shown) on the next line in the Immediate window:

   ```
   CurrentTime$ = time$
   ```

2. Press ENTER to execute the assignment statement you just typed.

3. Type the following line in the Immediate window, exactly as shown:

   ```
   print CurrentTime$
   ```

   Figure 3.4 shows how the Immediate window should now look:

Figure 3.4 Immediate Window Showing Lines Just Entered

4. Press ENTER to display the value of `CurrentTime$`.

   The value of `CurrentTime$` is displayed beneath the first line on the output screen.

5. Press a key to return to the environment.

   You can write up to 10 lines of statements in the Immediate window. If you type an eleventh line, the first line you typed is lost; if you type a twelfth, the second line you typed is lost, and so on. Section 10.5 illustrates other ways you can use the Immediate window.

Monitoring Variables with the Watch Window

When a program doesn't behave the way you expect, it is often because a variable is acquiring a value you did not anticipate. The Watch window lets you monitor the values of expressions while your program is running, so you can be sure their values are within the expected range. Try the following to see how the Watch window works:

1. Press F6 to move the cursor to the Immediate window if it is not already there.
2. Press ALT+D, then press A to choose the Debug menu's Add Watch command.

   QuickBASIC displays the Add Watch dialog box.
3. Type `CurrentTime$`, then press ENTER.

   The Watch window opens at the top of the screen. It contains the name from the View window's title bar, followed by `CurrentTime$`, the name of the variable you just entered (see Figure 3.5). The cursor returns to the Immediate window.

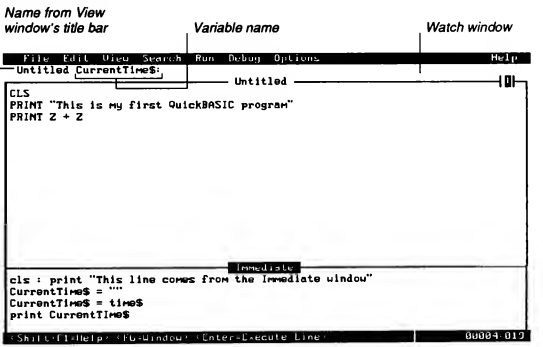


Figure 3.5 Placing Variables in the Watch Window

4. Press CTRL+HOME to place the cursor on the first line of the Immediate window, then press ENTER to execute each of the lines in the Immediate window.

   Look at the Watch window while you press ENTER on each line. As soon as the line `CurrentTime$ = time$` is executed, you will notice that the value for `CurrentTime$` appears in the Watch window.

With the Watch window you can monitor variable values as they change in the running program without having to use **PRINT** statements to display the value on the output screen.

The Help Window

Most on-line help appears in the Help window. There are several ways to get help on BASIC keywords, such as **PRINT** and **TIME$**, and the QuickBASIC environment. You can use the Help menu or press F1 while an item on screen (or a menu or command) contains the cursor or is highlighted.

NOTE *If you get error messages when you try to access on-line help, check Section 10.8.4, "Help Files," for more information on how to set up help properly.*

Context-Sensitive Help

With context-sensitive help, you can get information about BASIC keywords and program symbols. The following steps illustrate the use of context-sensitive help:

1. Press ESC to close the Help menu if it is open.
2. Use DIRECTION keys to move the cursor anywhere within the word `PRINT`.
3. Press F1.

   QuickBASIC displays a QuickSCREEN of the **PRINT** keyword (see Figure 3.6).

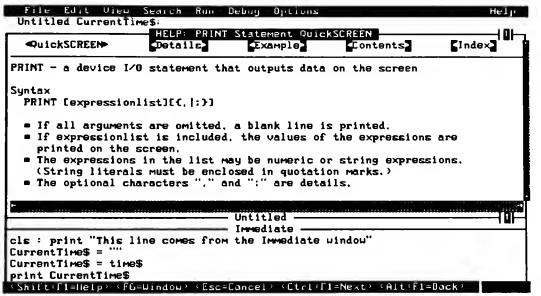


Figure 3.6 QuickSCREEN for PRINT Statement

The QuickSCREEN briefly describes the statement's action, then gives the syntax and explains each part of it. At the bottom, the QuickSCREEN indicates other uses of the keyword.

Hyperlinks

At the top of the screen, several bracketed items, called "hyperlinks," indicate related topics you can investigate. Hyperlinks provide instant connections between related topics. At the bottom of the screen, the other uses of the keyword are also

bracketed as hyperlinks. If you want more information on one of them, use the TAB key to move the cursor to the hyperlink, then press F1, as explained here.

1. Press F6 until the cursor appears into the Help window.
2. Press TAB to move the cursor to the Example hyperlink.

3. Press F1 to view the Example screen (see Figure 3.7).

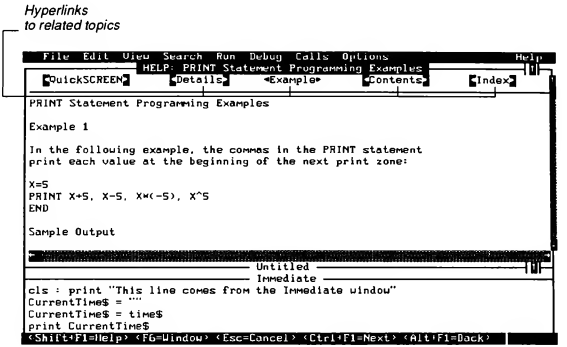


Figure 3.7 Example Screen for PRINT Statement

You can scroll within the Help window to examine all the **PRINT** examples.

4. Press ALT+F1 once or twice to return to the screen you linked from.

5. Repeat steps 1 and 2 to view the information available through the other bracketed hyperlinks.

You can use several successive hyperlinks without returning to the previous screen, QuickBASIC keeps track of the last 20 hyperlinks. You can use ALT+F1 to go back through the sequence of hyperlinks. When you reach the screen from which you started, QuickBASIC beeps if you press ALT+F1. You can press ESC any time to clear the Help window completely, even if you are several levels down in hyperlink screens.

For a reminder on which keys or key combinations to use for moving in or among Help screens, choose Contents from the Help menu, then choose the "Help keys" hyperlink. Also, SHIFT+F1 gives a general explanation of help.

Even some items that are not bracketed are hyperlinks. Press F1 any time you have a question about anything on the QuickBASIC screen. If QuickBASIC has more information on it, the information appears in the Help window; if no other information is available, QuickBASIC beeps.

NOTE *You cannot edit text in the Help window, but you can edit text in the View window or Immediate window while the Help window is open. In Chapter 8, "Using Example Code from On-Line Help," you'll learn to copy text from the Help window and paste it into your program.*

Exiting from QuickBASIC and Saving Programs

The following steps explain how to exit QuickBASIC and return to the DOS command line:

1. Press ALT+F, then press X to exit from QuickBASIC.

   A dialog box appears asking if you want to save the unsaved material in the View window.

2. Press ENTER to tell QuickBASIC that you want to save the program in the View window.

   QuickBASIC displays the Save As dialog box (see Figure 3.8):

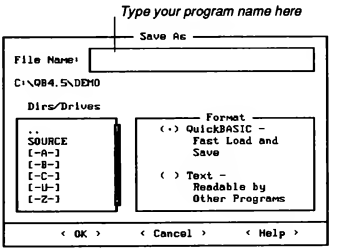


Figure 3.8 Save As Dialog Box

3. Type `program1` in the text box.

4. Press ENTER to save the lines in the View window on disk with the file name `PROGRAM1.BAS`.

For More Information

For more information on the topics covered in this chapter, see the following:

Where to Look	What You'll Find
Chapter 9, "Debugging While You Program," and Chapter 17, "Debugging Concepts and Techniques"	Debugging techniques and using the Watch window
Section 10.5, "Using the Immediate Window"	Ways to use the Immediate window
Chapter 11, "The File Menu"	Opening, clearing, and saving files in QuickBASIC

CHAPTER 4

Interlude: BASIC for Beginners

This chapter introduces the general principles of programming in BASIC. The focus is on the fundamentals, so you can start programming quickly.

If you have never programmed before, you should read this chapter carefully. When you finish this chapter, you will be able to write simple BASIC programs. You may want to read some intermediate-level books on BASIC before tackling advanced programming techniques in the following chapters.

If you already have programming experience with BASIC, you may want to skim this chapter to get a general idea of the differences between Microsoft BASIC and the BASIC dialect you know.

Many brief programming examples are given to show how each element of BASIC works. The quickest way to learn BASIC is to enter and run these examples as you read. You can enter multi-line examples in the View window. Or you can enter single-line examples in QuickBASIC's Immediate window, then press ENTER to see the resulting output. Study each example to understand why it works the way it does. Be sure you grasp the material in one section before going on to the next.

When you have completed this chapter, you will understand how to

- Display words and numbers on the screen
- Define and use variables
- Perform calculations and use BASIC's math functions
- Use arrays of variables
- Execute sections of your program selectively
- Repeat a group of instructions a specific number of times or until a condition is met

What is a Program?

A "program" is a sequence of instructions to the computer, in a language both you and the computer understand. Each instruction is called a "statement" and usually takes up one line in the program.

The goal of any program is to perform some useful job, such as word processing, bookkeeping, or playing a game. Programming is the process of deciding what you want the program to do. Then you select and arrange language statements to accomplish the individual tasks needed to reach the overall goal. This process is part science, part art. The science comes from books; the art is learned by writing your own programs and analyzing programs written by others.

Many BASIC statements have additional features that are not described in this chapter. If you want more information about a particular statement or language feature, please refer to Chapter 8, "Statement and Function Summary," in *Programming in BASIC* or consult QuickBASIC's on-line help.

Comments

Comments are an important part of programming. They explain the program to others, and remind you why you wrote it the way you did. Comments begin with an apostrophe, and can extend to the end of the line, as follows:

```
' This is a comment.

CLS                     ' The CLS statement clears the screen.
```

The apostrophe tells QuickBASIC that everything between the apostrophe and the end of the line is a comment.

Displaying Words and Numbers on the Screen

You are probably familiar with application software such as Lotus® 1-2-3® or Microsoft Word. These and most other programs communicate with the user by displaying words, numbers, and data on the screen.

The **PRINT** statement is BASIC's command for writing on the screen. Two of the things **PRINT** can display are strings and numbers.

A "string" is any group of characters (letters, numbers, punctuation marks) enclosed in quotes. The following example program shows how the **PRINT** statement displays a string:

1. Start QuickBASIC (if it is not already started) by typing `qb` and pressing ENTER.
2. Type the following line in the Immediate window, then press ENTER:

   ```
   PRINT "Hello, world"
   ```

   The output should appear as follows:

   ```
   Hello, world
   ```

   The quotes that surround a string are not part of the string, so they do not appear in the display.
3. Type `CLS` in the Immediate window and press ENTER to clear the output screen.

NOTE *BASIC is not case sensitive; you don't have to enter the examples with the cases as shown. QuickBASIC automatically capitalizes words that are part of the BASIC language but leaves everything else as you enter it. The examples in this chapter are shown the way QuickBASIC formats them, and each program's output is shown exactly as it appears on the screen.*

The **PRINT** statement can also display numbers as shown in the following example program:

1. Type the following line in the Immediate window, then press ENTER:

   ```
   PRINT 63
   ```

   The output should appear as follows:

   ```
    63
   ```
2. Type `CLS` in the Immediate window and press ENTER to clear the output screen.

Variables

Programs take data from the outside world (input), then process it to produce useful information (output). The input and output data are stored in the computer's memory. Variable names are used to keep track of where the data are stored.

Storing Data in Memory

Computer memory is divided into storage locations, each of which has a unique "address." The address is a number that gives the memory location, the same way your house address shows where you live on a street. Every piece of data has its own storage location and matching address (see Figure 4.1).

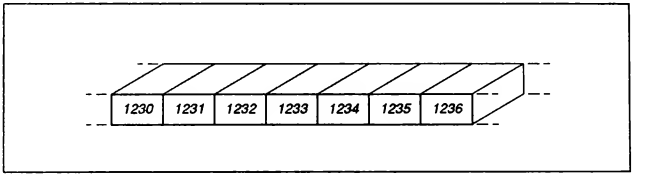


Figure 4.1 Addresses in Memory

Fortunately, you don't have to keep track of the address for every data item. BASIC uses a "variable name" to associate a word of your choice with a memory location in the computer.

BASIC does the associating automatically. Every time you use a variable name that BASIC hasn't seen before, BASIC finds an unused memory location and associates the new variable name with that address. You can then use the variable name without having to know its address.

The following example program demonstrates this. Note that `Hello, world` is not enclosed in quotes in this example.

1. Type the following line in the Immediate window, then press ENTER:

   ```
   PRINT Hello, world
   ```

   The output should appear as follows:

   ```
    0             0
   ```

2. Type `CLS` in the Immediate window and press ENTER to clear the output screen.

When BASIC encounters `Hello` and `world`, it has to decide what these words are. They aren't keywords (that is, they aren't part of the BASIC language), as **PRINT** is. They aren't strings, because they aren't enclosed in quotes. They are definitely not numbers. Therefore, `Hello` and `world` must be variables.

Once this decision is made, BASIC sets aside storage space for two new variables named `Hello` and `world`. New variables are always given an initial value of zero, so the **PRINT** statement displays two zeros.

A variable name can contain digits, but the name cannot begin with a digit. Any word starting with a digit is interpreted as a number.

Names allowed in BASIC include:

```
gerbils
birthdate
name3
ep451p33
```

These are illegal names:

```
5remove
4.3less
```

Variable Types

Data do not come in just one form. Programming languages, therefore, need different types of variables to hold different types of data. BASIC has three categories of variables: integer, floating point, and string.

Integer Variables

An integer number is a whole number, without a fractional part. For example, 0, –45, and 1278 are integers. Integer numbers are stored in integer variables. BASIC has two types of integer variables: integer and long integer.

A variable name ending in % is an integer variable. Integer variables can have any whole-number value between –32,768 and 32,767. A variable name ending in & is a long-integer variable and can have any whole-number value between –2,147,483,648 and 2,147,483,647.

Floating-Point Variables

A floating-point number can have a fractional part (though it doesn't have to). For example, 0.123, 78.26, 7.0, and 1922.001234 are all floating-point numbers. Floating-point numbers are stored in floating-point variables. BASIC has two types of floating-point variables: single precision and double precision.

A variable name ending in a letter, a digit, or ! is a single-precision floating-point variable. Single-precision floating-point numbers can have values ranging from about -3.4×10^{38} to 3.4×10^{38} (that's 34 with 37 zeros following). A single-precision floating-point variable is accurate to only about 7 decimal places.

A variable name ending in # is a double-precision floating-point variable and can have values from about -1.7×10^{308} to 1.7×10^{308} (that's 17 with 307 zeros following!). A double-precision floating-point variable can hold much larger numbers than a single-precision variable, and it is also more accurate— to about 15 decimal places.

Single-precision floating-point numbers are the kind most often used in calculations, since "real-world" applications usually need numbers with fractional parts. (Prices are a good example.) Since single-precision floating-point variables are so

commonly used, they are the "default" data type; that is, if a variable name does not end with a "type" character (%, &, #, !, or $), it is assumed to be a single-precision floating-point variable.

Note, however, that calculations with integer variables take much less computer time than those performed with floating-point variables. Integer variables should be used when speed is required.

The numerical variables in the remaining examples in this chapter are marked with the appropriate type suffix, either ! or %. Although most of these are single-precision floating-point variables, which need no type suffix, they are given the ! suffix for clarity.

String Variables

A string variable holds a sequence (a string) of characters. A variable name ending in $ is a string variable. Just as a numeric variable is given an initial value of 0, a newly defined string variable is assigned an initial length of zero. It contains no data at all, not even blank spaces. If `blank$` has not previously been given a value, the statement

```
PRINT blank$
```

displays an empty line.

Assigning Values to Variables

A variable gets its value through an "assignment" statement. An assignment statement has three components: the variable that receives a new value; an equal sign (=); and the number, string, or calculation whose value the variable takes on. These are all valid assignment statements:

```
age% = 41
myname$ = "Willie"
result! = result3! + leftval! + rightval!
```

In the next example program, two strings are combined with the plus sign (+) to make a longer string. The result is assigned to a third string.

1. Choose the File menu's New Program command to clear the View window (press ALT+F, then press N).
2. Type the following example in the View window:

```
string1$ = "Hello,"
string2$ = " there!"
bigstring$ = string1$ + string2$
PRINT bigstring$
```

3. Choose the Run menu's Start command (press ALT+R, then press S).

   The output should look like the following:

```
Hello, there!
```

4. Type `CLS` in the Immediate window and press ENTER to clear the output screen.

Although assignment is shown by the equal sign, it is not the same thing as algebraic equality. To BASIC, the assignment statement means: "Perform all the calculations and data manipulations on the right side of the equal sign. Then give the variable on the left side the result."

The same variable name can appear on both sides of the assignment statement. This occurs most often when you want to change the variable's value. Try the following example program:

1. Choose the File menu's New Program command to clear the View window.
2. Type the following example in the View window:

```
counter% = 6
PRINT counter%
counter% = counter% - 1
PRINT counter%
```

3. Choose the Run menu's Start command.

   The output should look like the following:

```
 6
 5
```

4. Type `CLS` in the Immediate window and press ENTER to clear the output screen.

You cannot assign numeric values to string variables (or vice versa). Therefore, the following statements are illegal. Either one halts execution and causes the error message `Type mismatch` when the program runs.

```
strg$ = 3.14159
number! = "hi, there"
```

Calculations

In BASIC, the mathematical operations of addition, subtraction, multiplication, and division are represented by **+**, **-**, ***** (the asterisk), and **/** (the slash), respectively. Exponentiation (raising a number to a power) is shown by **^** (the caret). You can type each of the following on a line in the Immediate window, then press ENTER to see the results of the operations:

Operation	Result
`PRINT 2 + 3`	`5`
`PRINT 2 - 3`	`-1`
`PRINT 2 * 3`	`6`
`PRINT 2 / 3`	`.6666667`
`PRINT 2 ^ 3`	`8`

Integer Division and the Remainder Operator

BASIC has two math operations you may not have seen before, integer division and the remainder operator.

Integer division retains the integer (whole-number) part of the division and discards the fractional part. Integer division is represented by \ (the backslash). You can type each of the following on a line in the Immediate window, then press ENTER to see the results of the operations:

Operation	Result
`PRINT 7 \ 3`	2
`PRINT 9.6 \ 2.4`	5

Both integer division and the remainder operator round off the numbers to be operated on before the calculation is performed. The number `9.6` becomes `10` and `2.4` becomes `2`. Therefore, the result of the second example above is 5, not 4.

The remainder operator is the complement of integer division. The whole-number part of the division is discarded, and the remainder is returned. Remainder division is performed by the BASIC keyword **MOD**. You can type each of the following on a line in the Immediate window, then press ENTER to see the results of the operations:

Operation	Result
`PRINT 7 MOD 3`	1
`PRINT 11 MOD 4`	3

Precedence of Operations

BASIC evaluates mathematical expressions from left to right, following the rules of algebraic precedence: exponentiation is performed first, then multiplication and division, then addition and subtraction. The following example program illustrates algebraic precedence:

1. Type the following line in the Immediate window, then press ENTER:

   ```
   PRINT 2 * 3 + 2 ^ 3 - 2 / 3
   ```

   The output should appear as follows:

   ```
    13.33333
   ```

2. Type `CLS` in the Immediate window and press ENTER to clear the output screen.

Parentheses can confirm or override the normal precedence. Try this example:

1. Type the following line in the Immediate window, then press ENTER:

```
PRINT (2 * 3) + (2 ^ 3) - (2 / 3)
```

   The output should appear as follows:

```
 13.33333
```

2. Type `CLS` in the Immediate window and press ENTER to clear the output screen.

The result is the same as in the last example because the precedence of operations dictated by the parentheses is no different from the usual order. However, the next example program produces a different result:

1. Type the following line in the Immediate window, then press ENTER:

```
PRINT 2 * (3 + 2) ^ (3 - 2 / 3)
```

   The output should appear as follows:

```
 85.49879
```

2. Type `CLS` in the Immediate window and press ENTER to clear the output screen.

The result here is different because the calculations within parentheses are performed first. Within parentheses, the usual precedence still applies, of course. The expression `(3 - 2 / 3)` evaluates as `(3 - .6666667)`, not `(1 / 3)`.

You might want to use parentheses to control the order of calculations, rather than depending on precedence. Parentheses prevent errors and make your programs easier to understand.

Math Functions

Along with the addition, subtraction, multiplication, and division operators just presented, BASIC provides a number of common math functions, described briefly below. To use them, follow the function name with the variable or

expression you want the function performed on (in parentheses). The expression may be any valid combination of variables, numbers, and math functions. (Expressions are explained further in the following section.)

Function	Description
ABS	Returns the absolute value of the expression
ATN	Returns the angle (in radians) whose tangent equals the expression
COS	Returns the cosine of an angle (in radians)
EXP	Returns *e* to the power of the expression
LOG	Returns the logarithm of the expression
SGN	Returns –1 if the expression is less than 0; returns 0 if it is 0; returns +1 if it is greater than 0
SIN	Returns the sine of an angle (in radians)
SQR	Returns the square root of the expression
TAN	Returns the tangent of an angle (in radians)

The following example program shows the use of the square-root function:

1. Choose the File menu's New Program command to clear the View window.
2. Type the following example in the View window:

```
adjacent! = 3.0
opposite! = 4.0
hypotenuse! = SQR((adjacent! ^ 2) + (opposite! ^ 2))
PRINT "the hypotenuse is:"; hypotenuse!
```

3. Choose the Run menu's Start command.

   The output should look like the following:

```
the hypotenuse is: 5
```

4. Type `CLS` in the Immediate window and press ENTER to clear the output screen.

Expressions

An "expression" is what most people would call a formula. An expression is any combination of numbers, strings, variables, functions, and operators that can be evaluated. For example, `2 + 2` is a simple expression. It evaluates to four. Enter the following statements in the View window to assign a variety of simple expressions to variables, as follows:

1. Choose the File menu's New Program command to clear the View window.
2. Type the following example in the View window:

```
square% = 3 ^ 2
root! = SQR(3 ^ 2)
greeting$ = "Hello," + " world! "
strange! = 9 + "cats"
```

   The last statement you typed was illegal.

3. Choose the Run menu's Start command.

   QuickBASIC displays the error message `Type mismatch`. You cannot add a number (or numerical variable) to a string (or string variable).

4. Type `CLS` in the Immediate window and press ENTER to clear the output screen.

Expressions may be as complex as you like, with the results of one expression passed to a function, the value returned by a function combined with other expressions, and so on.

Almost any place you can use a number or a string, you can use any expression (no matter how complex) that evaluates to a number or a string. This means expressions can be used in BASIC's math functions, **PRINT** statements, and assignment statements.

Displaying Variables and Expressions

The **PRINT** statement was first shown in this chapter as a way to display numbers and strings. The **PRINT** statement can also display the values of variables and expressions. For example, try this:

1. Type the following lines in the Immediate window, then press ENTER:

```
PRINT (2 + 8) / 4 + 1
```

   The output should appear as follows:

```
 3.5
```

2. Type `CLS` in the Immediate window and press ENTER to clear the output screen.

The **PRINT** statement is quite flexible; it can display any combination of strings, numbers, variables, and expressions. The items to be displayed are separated by semicolons (;) or commas (,). When a semicolon separates two items, the second item is displayed immediately after the first, as shown in the example program below:

1. Type the following line in the Immediate window, then press ENTER:

```
PRINT "Sum of 5.29 and"; 2.79; "is"; 5.29 + 2.79
```

   The output should appear as follows:

```
Sum of 5.29 and 2.79 is 8.08
```

2. Type `CLS` in the Immediate window and press ENTER to clear the output screen.

Here's another example program to try:

1. Choose the File menu's New Program command to clear the View window.
2. Type the following example in the View window:

```
number = 36
PRINT "The sum of"; number; "and 4 is"; number + 4
PRINT "The square of"; number; "is"; number * number
```

3. Choose the Run menu's Start command.

   The output should look like the following:

```
The sum of 36 and 4 is 40
The square of 36 is 1296
```

4. Type `CLS` in the Immediate window and press ENTER to clear the output screen.

If two items in a **PRINT** statement are separated by a comma, the second item is positioned at the next preset tab location. (The tab stops are 14 columns apart.) Note the difference between the output of the following **PRINT** statement and that of the similar one with semicolons, above:

1. Enter the following lines in the Immediate window, then press ENTER:

```
PRINT "Sum of 5.29 and", 2.79, "is", 5.29 + 2.79
```

   The output should appear as follows:

```
Sum of 5.29 and          2.79          is             8.08
```

2. Type `CLS` in the Immediate window and press ENTER to clear the output screen.

Commas and semicolons can be used in the same **PRINT** statement to position the items any way you like.

Each **PRINT** statement normally places its output on a new line, as the following example program shows:

1. Choose the File menu's New Program command to clear the View window.
2. Type the following example in the View window:

```
PRINT "How old are you?"
PRINT 41
```

   The output should appear as follows:

```
How old are you?
 41
```

3. Choose the Run menu's Start command.
4. Type `CLS` in the Immediate window and press ENTER to clear the output screen.

If you want to combine the output of several **PRINT** statements on the same line, place a semicolon (;) at the end of each **PRINT** statement. Change the last example program as follows:

1. Choose the File menu's New Program command to clear the View window.
2. Type the following example in the View window:

```
PRINT "How old are you?";
PRINT 41
```

3. Choose the Run menu's Start command.

   The output should look like the following:

```
How old are you? 41
```

4. Type `CLS` in the Immediate window and press ENTER to clear the output screen.

Entering Data with the INPUT Statement

The **INPUT** statement displays a prompt string, then waits for the user to enter some value. The value entered in response to the prompt is assigned to the variable specified in the **INPUT** statement.

```
INPUT "enter your age: ", age%
PRINT age%
```

If you typed `39` in response to the `enter your age:` prompt, the value 39 would be assigned to the variable `age%`, and that value would be printed by the following `PRINT age%` statement.

INPUT also accepts strings:

```
INPUT "What is your name? ", yourname$
PRINT yourname$
```

A string variable will accept whatever you type, since all entries—even numbers—are actually strings. (When you type `42`, you are not typing the number 42, but rather the characters `4` and `2`. If the input variable is numerical, BASIC converts this string into a number. If the input variable is a string, the character sequence `42` is simply assigned to the string.)

A numeric variable, however, can only be assigned a number. If you type `dog` when you should have entered a number, BASIC responds with the message `Redo from start` and a new prompt line.

Arrays of Variables

If you wanted to keep track of the high temperature for each day in July, you could use 31 floating-point variables, named `july1`, `july2`, `july3`, and so on through `july31`.

However, a group of related but individually named variables is clumsy to work with; each variable has to be handled separately. For example, if you wanted to prompt the user for each temperature in July, you would have to write 31 individual prompt statements.

```
INPUT "What is the temp value for July 1? ", july1
   .
   .
   .
INPUT "What is the temp value for July 31? ", july31
```

An analogous problem occurs when you're trying to write a program that retrieves the stored data. You wouldn't want to write a program that displayed all 31 variable names to be able to retrieve the value for any specific one (say, the high temperature for July 4—the value for `july4`). It would be much more convenient if you could just give the computer the date (the number `4`) and the computer would select the right variable and value.

The solution is to give the entire set of values the same name (`july` in our example) and distinguish the individual values by a numerical index called a "subscript." A group of identically named variables distinguished by their subscript values is called an "array." (See Figure 4.2 for an example.) Arrays can be especially powerful when used in conjunction with loops (see the section titled "Repeating Program Operations").

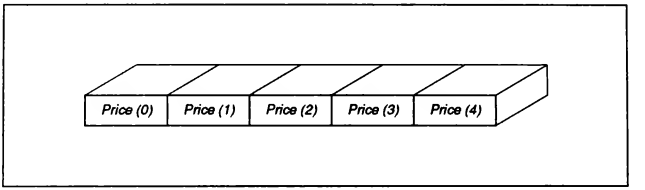


Figure 4.2 Array Created by the BASIC Statement DIM Price (4)

Declaring Arrays

An array variable must first be "declared" with a **DIM** statement, which establishes the size of the array. A **DIM** statement must appear before the array variable can be used.

The value given in the **DIM** statement specifies the maximum value for the variable's subscript. For example, the following statement declares an array of 32 floating-point variables named `july`, whose subscripts run from 0 to 31:

```
DIM july(31)
```

DIM also works with string variables:

```
DIM stringvar$(250)
```

Specifying Array Elements

You can specify the *n*th member (or "element") of an array simply by putting the number *n* (or a variable or expression with the value *n*) in parentheses following the variable name. In this way the program can connect a specific variable with a number entered by the user. For example, the fourth element of the `july` array could be specified by either `july(4)` or—if `d%` equals four—`july(d%)` or even `july(SQR(16))`.

The following code fragment prompts the user for the date, then prints the high temperature for that day in July (assuming that all the temperatures have been assigned to the array `july`).

```
DIM july(31)
INPUT "enter July date: ", date%
PRINT "high temperature for July"; date%; "was"; july(date%)
```

Logical Relations Used in Decision-Making

BASIC can decide whether two numbers or strings are the same or different. On the basis of such a decision, an appropriate group of program statements can be executed or repeated.

For example, suppose the standard income-tax deduction is $2500. If your itemized deductions are less than $2500, you would take the standard deduction. If your itemized deductions are greater than $2500, you would itemize. An income-tax program can make this decision and execute either the part of the program that performs itemized deductions or the part that takes the standard deduction. The user doesn't have to tell the program what to do.

The following sections explain how BASIC makes logical decisions that can be used in your programs.

Relational Operators

Numbers and strings are compared using "relational operators." There are six possible relationships, shown by the following operators:

Relational Operator	Meaning
=	Equal to
<>	Not equal to
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to

NOTE *BASIC uses the equal sign (=) both to assign a value to a variable and also to indicate logical equality. BASIC knows which usage is intended from the context.*

The following list shows how these relationships are evaluated:

Relational Operation	Evaluation
`7 = 35`	False
`7 <> 6`	True
`6 > 1`	True
`4 < 3`	False
`7 <= 7`	True
`8 >= 7`	True
`apples = oranges`	Depends on values of variables

As for the last statement—apples aren't oranges. But if `apples` and `oranges` were variable names, then the truth or falsity of the statement would depend on the values of these variables.

The following relational operations compare strings. Are they true or false?

```
"Tarzan" <> "Jane"
"Y" = "Y"
"Y" = "y"
"miss" > "misty"
```

`Tarzan` and `Jane` are different strings, so the first assertion is true. `Y` and `Y` are the same character, so the second assertion is true. `Y` and `y` are not the same character, so the third assertion is false.

The last statement may be confusing at first; how can one string be "larger" or "smaller" than another? It can be, and the decision is made this way. The strings are first compared, position by position, until BASIC finds characters that don't match. In this example, the fourth characters are the first nonmatching characters. The decision of "larger" or "smaller" is then based on these differing characters, in the following way.

Within the computer, characters are represented by numbers. Letters coming later in the alphabet have numeric values larger than those appearing earlier. Since `t` follows `s` in the alphabet, `t` is a "larger" letter, and the statement is therefore

false; `misty` is "larger" than `miss`. This ability to determine the relative value of strings makes it easy to write a program that can alphabetize a list of words or names.

In most of these examples, numbers were compared with numbers and strings with strings. But any combination of numbers and numerical variables can be compared; the same is true for strings and string variables. The only things you cannot compare are numerical values with string values because the comparison is logically meaningless.

Boolean Expressions

Logical assertions are called "Boolean expressions" (for George Boole, who formulated some of the rules of mathematical logic). Boolean operations are concerned with only one thing: is a Boolean expression true or false? Unlike mathematical expressions, which evaluate to numbers, Boolean expressions evaluate to one of two values: "true" or "false."

Computers use numbers to represent "true" and "false." To see how BASIC codes "true" and "false," enter and run these two lines, as follows:

1. Choose the File menu's New Program command to clear the View window.
2. Type the following example in the View window:

```
PRINT 5 > 6
PRINT 6 > 5
```

3. Choose the Run menu's Start command.

   The output should look like the following:

```
 0
-1
```

   The display shows `0` on the first line, `-1` on the second.

   The number `5` is not greater than `6`, so BASIC prints its value for "false": `0`.
   The number `6` is greater than `5`, so BASIC prints its value for "true": `-1`.

4. Type `CLS` in the Immediate window and press ENTER to clear the output screen.

Compound Expressions

In BASIC, a compound Boolean expression is created by connecting two Boolean expressions with a "logical operator." The two most commonly used logical operators are **AND** and **OR**.

The **AND** operator requires both expressions to be true if the compound expression is to be true. When the **OR** operator is used, only one of the expressions has to be true for the compound expression to be true.

The following examples are compound expressions that combine two simple expressions by using **AND** or **OR**:

Expression	Evaluation
`10 > 5 AND 100 < 200`	True
`3 < 6 AND 7 > 10`	False
`8 < 7 OR 90 > 80`	True
`2 < 1 OR 3 > 60`	False
`"Y" > "N" AND "yes" <> "no"`	True
`"Y" < "N" OR 4 <> 4`	False

The **NOT** operator reverses the truth or falsity of an expression:

Expression	Evaluation
`NOT (5 > 10)`	True
`NOT (8 < 7 OR 90 > 80)`	False
`NOT (3 < 6 AND 7 > 10)`	True
`NOT (0)`	True

The logic of the last example may not be immediately obvious. In BASIC, false is zero. Therefore, `NOT (0)` means "not false," or true. Combinations of **AND**, **OR**, and **NOT** can be used to build up very complex expressions. There is no practical limit to their complexity, except the obvious one of writing an expression that is difficult to read.

When Boolean expressions are evaluated, the precedence of evaluation is taken in the same order as arithmetic expressions, with **AND** equivalent to multiplication and **OR** equivalent to addition. Evaluation goes from left to right, with all the simple expressions evaluated first. All the **AND**-connected expressions are evaluated next, followed by the **OR**-connected expressions. As with arithmetic expressions, parentheses can be used to control the order of evaluation, rather than relying on the rules of precedence.

See Chapter 1, "Control-Flow Structures," in *Programming in BASIC*, for more explanation of why BASIC considers –1 equivalent to true.

The next sections show how Boolean expressions are used in programs.

Using Logical Statements to Control Program Flow

You are always deciding what to do next by evaluating your priorities and desires, asking yourself which is most important at the moment. Similarly, BASIC has a mechanism that allows your program to select which set of operations (that is, which part of the program) will be executed next.

This mechanism is the **IF...THEN...ELSE** statement. If a Boolean expression is true, the statements following **THEN** are executed. If the expression is false, the statements following **ELSE** are executed. A Boolean expression must be either true or false, so one or the other group of statements has to execute. Both cannot execute.

The process of selecting among two or more possible actions (or sections of program code) is called "branching." The analogy is with the way a tree branch subdivides into more and more branches.

This is the syntax of the **IF...THEN...ELSE** statement:

IF *booleanexpression* **THEN**
  statements to do something
ELSE
  statements to do something else
END IF

Try the following simple example to use the **IF...THEN...ELSE** statement to decide whether a number entered is greater than, less than, or equal to 100:

1. Choose the File menu's New Program command to clear the View window.
2. Type the following example in the View window:

```
INPUT "enter a number: ", n

IF n > 100 THEN
   PRINT n; "is greater than 100"
ELSE
   PRINT n; "is less than or equal to 100"
END IF
```

3. Choose the Run menu's Start command.

   The logical opposite of "greater than" is "less than or equal to" (not "less than"). Run this example with several values for *n*, including a value of 100 to verify that the `is less than or equal to 100` statement is displayed.

4. Type `CLS` in the Immediate window and press ENTER to clear the output screen.

See Chapter 1, "Control-Flow Structures," in *Programming in BASIC* for a more detailed explanation of the **IF...THEN...ELSE** statement and other decision structures available in QuickBASIC.

Repeating Program Operations

BASIC offers several ways to execute a group of program statements repeatedly. You can repeat them a fixed number of times or until a particular logical condition is met. If you want to execute a block of statements 100 times, you need only type them in once. BASIC's control structures then control the repetitions.

The ability to repeat program statements has many uses. For example, to raise a number *N* to the power *P*, *N* must be multiplied by itself *P* times. To enter data for a ten-element array, you must prompt the user ten times.

The FOR...NEXT Loop

One way to repeat a section of the program is the **FOR...NEXT** loop. The block of program statements between the **FOR** and **NEXT** keywords is executed repeatedly a specified number of times. Try the following:

1. Choose the File menu's New Program command to clear the View window.
2. Type the following example in the View window:

```
FOR count% = 1 TO 5
   PRINT "this line is printed 5 times; this is time"; count%
NEXT count%
```

3. Choose the Run menu's Start command.

   The output should look like the following:

```
this line is printed 5 times; this is time 1
this line is printed 5 times; this is time 2
this line is printed 5 times; this is time 3
this line is printed 5 times; this is time 4
this line is printed 5 times; this is time 5
```

4. Type `CLS` in the Immediate window, then press ENTER to clear the output screen.

In the example above, the variable `count%` is called the "loop variable" or "loop counter." The two numbers after the equal sign (separated by the keyword **TO**) are the start and end values for the loop counter. In this example, 1 is the start value, and 5 is the end value.

Before the statements in the **FOR...NEXT** loop are executed for the first time, the loop counter is given the start value (in this case, `1`). After each execution of the statements in the loop, BASIC automatically increments `count%` by one. This cycle continues until `count%` is larger than the end value (in this case, `5`). The statements between the **FOR** and **NEXT** keywords in this example are therefore executed five times.

To repeat a **FOR...NEXT** loop *n* times, the start and end values would normally be 1 and *n*. This is not a necessity, though. They can have any two values that differ by $n - 1$. For example, using –2 and 2 would also cause five repetitions.

The counter can be the subscript variable for an array. (Accessing each element in an array is a common use for a **FOR...NEXT** loop.) The next example prompts the user for the high temperature on each day in July:

```
DIM july!(31)

FOR date% = 1 TO 31
    PRINT "enter the high temp for July"; date%;
    INPUT " ", july!(date%)
NEXT date%
```

Neither the start nor the end value has to be a constant; either or both can be a variable. The following example requests start and end values, then prints out the square root of every value in that range:

```
INPUT "starting value? ", startval%
INPUT "ending   value? ", endval%
FOR count% = startval% TO endval%
    PRINT "the square root of"; count%; "is"; SQR(count%)
NEXT count%
```

The increment defaults to a value of one if it isn't specified. The **STEP** statement sets a different increment value, as shown in the following procedure:

1. Choose the File menu's New Program command to clear the View window.
2. Type the following example in the View window:

   ```
   FOR count% = 5 TO 1 STEP -1
      PRINT "this line is printed 5 times; this is time"; count%
   NEXT count%
   ```

3. Choose the Run menu's Start command.

   The output should look like the following:

   ```
   this line is printed 5 times; this is time 5
   this line is printed 5 times; this is time 4
   this line is printed 5 times; this is time 3
   this line is printed 5 times; this is time 2
   this line is printed 5 times; this is time 1
   ```

4. Type `CLS` in the Immediate window, then press ENTER to clear the output screen.

The step value must be consistent with the start and end values. If the end value is larger than the start value, the step must be positive. If the end value is smaller than the start value, the step must be negative. If the step has the wrong sign, the **FOR...NEXT** loop doesn't execute even once; it is skipped over.

The DO...LOOP

You have just seen one way to repeat a section of the program, the **FOR...NEXT** loop. The **FOR...NEXT** loop, however, has a serious limitation. Although you can specify the starting and ending points, the loop itself is fixed; it repeats a specific number of times, no more and no less.

This caused no problem in the example where you were prompted for the 31 high temperatures in July, because July always has 31 days. However, there are times you can't know how often to repeat a calculation or an operation.

Suppose you were searching through text, one line at a time, for a specific word. You don't know which line the word is on (if you did, you wouldn't have to search), so you can't specify the number of times to go through a **FOR...NEXT** loop. You need a flexible (rather than fixed) way to repeat groups of program statements.

The **DO...LOOP** is the more flexible mechanism for repetition needed here. Program statements between the **DO** and **LOOP** keywords are repeated an indefinite number of times. Try the following example:

1. Choose the File menu's New Program command to clear the View window:
2. Type the following example in the View window.

```
DO
   PRINT "Going around in circles..."
LOOP
```

3. Choose the Run menu's Start command.

   `Going around in circles...` is printed over and over, without end. (To stop the program, press CTRL+BREAK.)

4. Type `CLS` in the Immediate window and press ENTER to clear the output screen.

The simple **DO...LOOP** shown above is endless. BASIC has a way to terminate the loop at the right time. All you have to do is specify the logical condition you want to cause termination.

One way to do this is to use a **WHILE** condition, which tells the computer to continue executing the loop **WHILE** some condition *is* true. Another is to use an **UNTIL** condition, meaning continue executing the loop **UNTIL** some condition *becomes* true.

In a **DO WHILE** loop, the condition starts off true, and the loop executes as long as the condition remains true. In a **DO UNTIL** loop, the condition starts off false, and the loop executes until the condition becomes true.

The user-defined condition is a Boolean expression. This expression usually compares a variable with a constant. When the expression satisfies the logical condition set by the programmer (such as the variable equalling the constant), the loop terminates. (Any comparison is possible, but comparing the loop variable with a number or string is the most common.)

The DO WHILE Loop

A **DO WHILE** loop looks like this:

DO WHILE *booleanexpression*
  statements to be repeated
LOOP

Here is a simple **DO WHILE** loop that demonstrates how such a loop operates. The loop termination condition is for `big!` to equal `little!`.

```
big! = 256.0
little! = 1.0
DO WHILE big! <> little!
   PRINT "big ="; big!; "  little ="; little!
   big! = big! / 2.0
   little! = little! * 2.0
LOOP
```

Two variables, `big!` and `little!`, are initialized outside the loop. Within the loop, their values are first printed, then `big!` is divided by 2 and `little!` is multiplied by 2. The process repeats until both are equal to 16 and the loop terminates.

The initial values of `big!` and `little!` were chosen to guarantee that they would eventually be equal. If they are not so chosen, the loop repeats indefinitely. (In practice, the program eventually stops regardless of the values chosen. The variable `little!` is repeatedly doubled, ultimately becoming so large that its value cannot be represented by a BASIC floating-point variable. The program then halts, displaying an `Overflow` error message.)

Please note that for a **DO WHILE** loop to execute even once, the Boolean expression must be true. If `big!` and `little!` started with the same value, the expression `big! <> little!` would be false, and the loop would not execute at

all; it would be skipped over. Always be sure the variables in the Boolean expression have the values you really want before entering the loop.

The DO UNTIL Loop

A **DO UNTIL** loop is identical in form to a **DO WHILE** loop, except it uses the keyword **UNTIL**:

DO UNTIL *booleanexpression*
  statements to be repeated
LOOP

The loop shown in the last example could also have been written using **UNTIL**, as shown here:

```
big! = 256.0
little! = 1.0
DO UNTIL big! = little!
   PRINT "big ="; big!; "  little ="; little!
   big! = big! / 2.0
   little! = little! * 2.0
LOOP
```

A **DO UNTIL** loop continues until the Boolean expression becomes true, so a different terminating condition is needed. In this case, it is the opposite of the condition in the **DO WHILE** example above: `big! = little!`.

The following is a more practical example of a **DO...LOOP**. The user is repeatedly prompted for a Yes / No response (indicated by `Y` or `N`) until it is supplied:

```
DO UNTIL instring$ = "Y" OR instring$ = "N"
   INPUT "enter Yes or No (Y/N): ", instring$
LOOP
PRINT "You typed"; instring$
```

Note that `y` and `n` are not accepted, since `instring$` is compared only with uppercase characters.

NOTE *In the loop examples given, both the* ***WHILE*** *and* ***UNTIL*** *statements are located after* ***DO****. It is also possible to place them following the* ***LOOP*** *keyword. When that is done, the statements in the loop are executed first, then the condition is tested. This assures that the loop will always be executed at least once. See Chapter 1, "Control-Flow Structures," in Programming in BASIC, for an explanation of how this works.*

Writing a Simple BASIC Program

If you've worked through the examples, you should now know the fundamentals of programming in BASIC. To demonstrate your understanding, try writing the simple program described below.

The program computes the average of several numbers. It could be written in the following sequence:

1. Prompt the user for the quantity of numbers to be averaged.
2. If the quantity is zero or negative, print a warning message and do nothing else.
3. If the quantity is positive, pick a variable name for the running total and set it equal to zero.
4. Prompt for the numbers, one at a time. Tell the user each time which value is being entered (number 1, number 2, and so on).
5. When all the numbers have been entered, compute and print the average value.

This program uses only BASIC language elements that have been explained in this chapter, and no programming "tricks" are needed. A solution is given below, but please try to create your own program before peeking. This is an "open-book" project. If you get stuck, simply reread this chapter to find the program statements you need and to see how they are used.

There are many ways to write a program. If your program works, it's correct, even if it differs from the program given here.

```
INPUT "How many numbers do you wish to average? ", howmany%

IF howmany% <= 0 THEN
   PRINT "Not a valid quantity; must be greater than 0."
ELSE
   total! = 0.0                    ' Set running total to zero.
   FOR count% = 1 TO howmany%
       PRINT "number"; count%;
       INPUT " ", value!
       total! = total! + value! ' Add next value to total.
   NEXT count%
   PRINT "The average value is"; total! / howmany%
END IF
```

For More Information

Now you know how to write simple programs in BASIC. There is, however, a great deal more to BASIC programming.

QuickBASIC's manuals and on-line help provide complete information on using the QuickBASIC environment and each of the QuickBASIC statements and functions. However, the documentation does not include a complete introduction to BASIC, and does not teach the general principles of programming, or how to use DOS. 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 neither endorses these books nor recommends them over others on the same subjects.

Books about BASIC

Craig, John Clark. *Microsoft QuickBASIC Programmer's Toolbox.* Redmond, Wash.: Microsoft Press, 1988.

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.

Feldman, Phil, and Tom Rugg. *Using QuickBASIC 4.* Carmel, Ind.: Que Corporation, 1988.

Hergert, Douglas. *Microsoft QuickBASIC.* 2d ed. Redmond, Wash.: Microsoft Press, 1988

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

Inman, Don, and Bob Albrecht. *Using QuickBASIC.* Berkeley, Cal.: Osborne McGraw-Hill, 1988.

Books about DOS

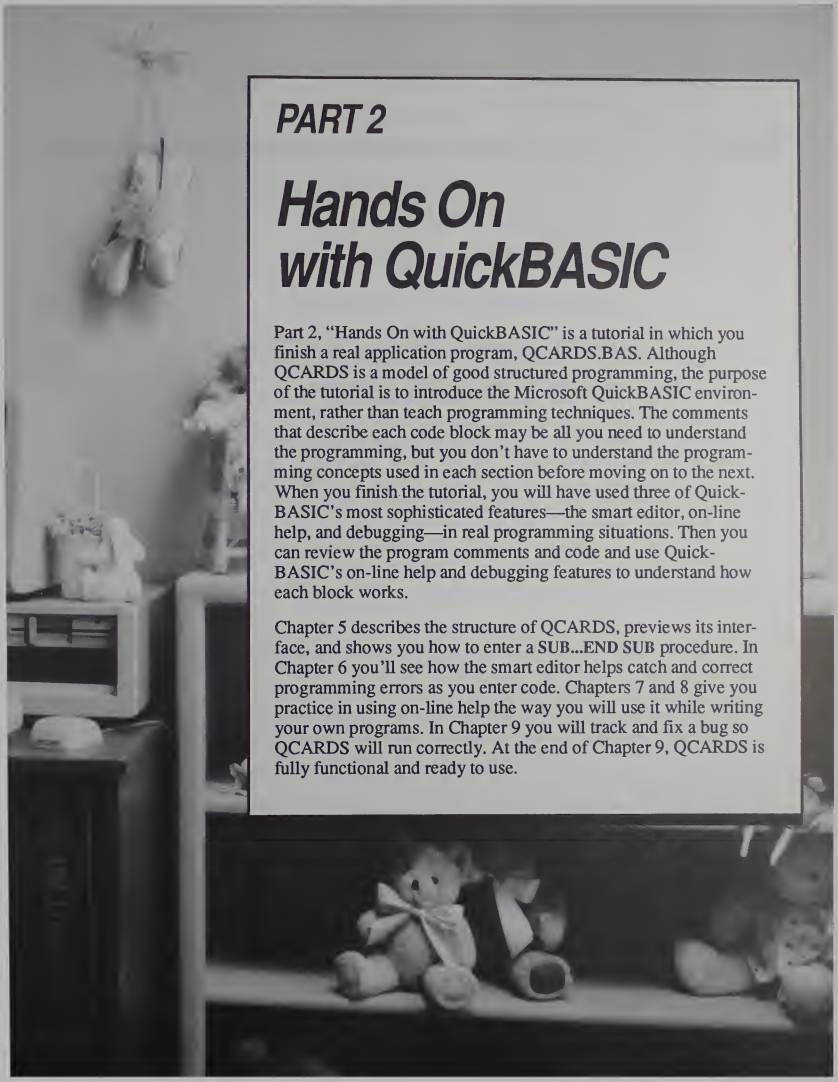
Duncan, Ray. *Advanced MS-DOS*. Redmond, Wash.: Microsoft Press, 1986.

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

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

PART 2

Hands On with QuickBASIC



PART 2

Hands On with QuickBASIC

Part 2, "Hands On with QuickBASIC" is a tutorial in which you finish a real application program, QCARDS.BAS. Although QCARDS is a model of good structured programming, the purpose of the tutorial is to introduce the Microsoft QuickBASIC environment, rather than teach programming techniques. The comments that describe each code block may be all you need to understand the programming, but you don't have to understand the programming concepts used in each section before moving on to the next. When you finish the tutorial, you will have used three of QuickBASIC's most sophisticated features—the smart editor, on-line help, and debugging—in real programming situations. Then you can review the program comments and code and use QuickBASIC's on-line help and debugging features to understand how each block works.

Chapter 5 describes the structure of QCARDS, previews its interface, and shows you how to enter a **SUB...END SUB** procedure. In Chapter 6 you'll see how the smart editor helps catch and correct programming errors as you enter code. Chapters 7 and 8 give you practice in using on-line help the way you will use it while writing your own programs. In Chapter 9 you will track and fix a bug so QCARDS will run correctly. At the end of Chapter 9, QCARDS is fully functional and ready to use.

CHAPTERS

5 The QCARDS Program 75

6 Editing in the View Window 87

7 Programming with On-Line Help 105

8 Using Example Code from On-Line Help 115

9 Debugging While You Program 129

CHAPTER 5

The QCARDS Program

This chapter describes QCARDS.BAS, the program on which the rest of the tutorial on QuickBASIC is based. You'll add some code in the form of a "procedure," a routine you write once and then call from different parts of the program, to perform a specific task. In this chapter, you will

- Load a program at the same time you invoke QuickBASIC
- Browse through the QCARDS.BAS code
- Call procedures from the Immediate window
- Create and call a **SUB** procedure

This chapter takes about one hour to complete.

Building QCARDS

As you work through the QuickBASIC tutorial in Chapters 5–9, you'll add lines of code to the QCARDS program. The code-entry sequences that are part of QCARDS appear under special headings, so you know which ones are necessary for finishing the tutorial program. You can save your work as you finish each numbered sequence. If you make a mistake during a code-entry sequence, just reload the previous saved version and repeat the current sequence.

NOTE *As you browse through QCARDS, you may notice some errors in the code. These errors are intentional. You will fix them as you work through the tutorial.*

Loading a Program When You Start QuickBASIC

You can start QuickBASIC and load a program at the same time by including the program name as a command-line "argument." An "argument" is information you supply to a command to modify the command.

NOTE *If you don't have a hard disk, copy the files QCARDS.BAS and QCARDS.DAT to an empty removable disk and use that disk to complete the tutorial.*

Type the following line at the DOS prompt to start QuickBASIC and have it automatically load QCARDS.BAS:

```
➤ qb qcards
```

When you start QuickBASIC this way, QuickBASIC searches the current working directory for a file with the name QCARDS and the extension .BAS. If QuickBASIC finds the file, it loads it. If the file is not found, QuickBASIC assumes you are starting a new program that you want to name QCARDS.BAS.

A Quick Tour of QCARDS

If you completed the computer-based training called QB Express, you'll recognize some of QCARDS. In Chapters 5–9 you'll work with the real QCARDS code.

If you are new to programming, some of this material may seem difficult because QCARDS is a real application program. Don't worry—you don't need to master all the material in each section before moving on. This tutorial is designed to teach the programming tools of the QuickBASIC environment, not programming techniques.

Even if you have programmed in BASIC before, you may learn some powerful techniques because QCARDS is a structured program that uses procedures. In fact, if you have used a BASIC interpreter (like GW-BASIC® or BASICA), you're in for a pleasant surprise.

The QCARDS Program

QCARDS is a database program. It provides a convenient way to work with many small collections of information (called records), which are stored together in a disk file. QCARDS manages the disk file—opening and closing it and updating the records as you modify, add, or delete specific pieces of information or whole records. Tne fun part of QCARDS is the "interface," the part of the program where the user interacts with QCARDS (see Figure 5.1).

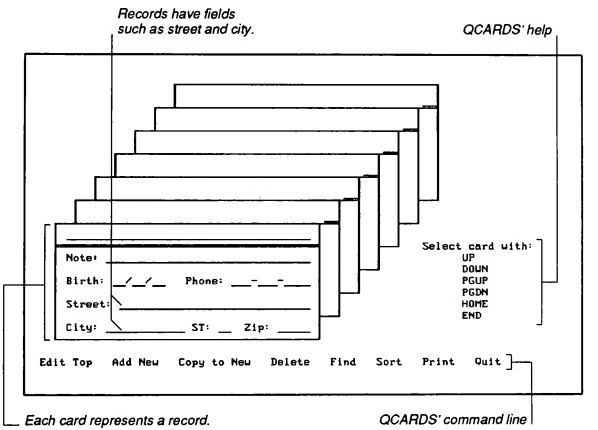


Figure 5.1 QCARDS' Interface

QCARDS uses the concept of an index-card file to present the data in a familiar way. Each card that appears on screen represents a record in the database. Each card has several "fields" (separate areas on the card) that are used to enter new data or change and reorder existing data.

Once you understand QCARDS, you can modify the data (and the program itself) and use it for any kind of information you wish to store. QCARDS is a large program, but it is organized according to structured programming principles (that is, it divides complicated tasks into simpler ones). This means you can learn its programming techniques gradually. As you read the next few sections, scroll down through QCARDS as each part is described.

Declarations and Definitions

QCARDS begins with a short section describing its purpose. Next comes a section of declarations and definitions. The declarations and definitions use statements like **DEFINT, CONST, TYPE...END TYPE, DECLARE, DIM,** and **REDIM.** They let QuickBASIC anticipate the amount of space to reserve in memory for the program's variables. For example, `DEFINT A-Z` tells QuickBASIC that any variable for which a data type is not explicitly declared should be considered an integer.

CONST statements improve program readability by giving descriptive names to values the computer regards only as numbers. For example, the value `0` (zero) represents the logical condition "false" in QuickBASIC. At about line 45, a **CONST** statement defines the symbolic constant `FALSE` as the value zero. This lets you use the word `FALSE` to represent the value `0` in logical comparisons.

TYPE...END TYPE constructions declare "aggregate" data types. Aggregate data types are those that can include both strings and all types of numeric values. For example, once the `PERSON` data type is declared (at about line 56), you can define variables to have this user-defined type.

Lines that begin with the **DECLARE** statement allow QuickBASIC to check procedure calls to make sure you use them correctly.

Comments

After the declarations and definitions comes a large section that contains mostly comments. Each comment block describes a block of executable statements that you will enter as you complete QCARDS. The statements define the program's overall flow of control. This section of QCARDS concludes with an **END** statement to mark the end of the program's normal execution sequence.

Statements Following the END Statement

The final section of the program starts after the **END** statement. Each group of statements in this section is preceded by a "label," a name followed by a colon. (A QuickBASIC label is similar to line numbers in BASIC interpreters.) The statements associated with these labels are executed only if other statements (that contain references to the labels) are executed. The first label (`MemoryErr`) labels an error handler. If a certain type of error occurs during execution of QCARDS,

the statements following `MemoryErr` display a message on the screen, then end the program. The remaining labeled sections contain **DATA** statements that are executed when the index cards are drawn on the screen.

Calling QCARDS Procedures from the Immediate Window

There are two levels of programming in QCARDS. The module level contains the program's most general logic. The procedure level contains many individual procedures that do the program's detailed work.

Just as you can use the Immediate Window to see the effects of a BASIC statement, you can use it to call a program's procedures. This lets you see the effects of a procedure without running the entire program. Each procedure performs a specific task. The following steps illustrate how to call procedures that display the QCARDS user interface on the output screen:

1. Press F6 to move the cursor into the Immediate window.
2. Type `CLS` and press ENTER to clear the output screen. Then press a key to return the cursor to the Immediate window.
3. Type `CALL DrawCards` and press ENTER.

   QuickBASIC executes the `DrawCards` procedure. Calling a procedure is just like executing a BASIC statement. The procedure performs its task, then QuickBASIC waits for the next instruction.
4. Press a key to return to the Immediate window, then type the following on the next line in the Immediate window, exactly as shown:

   ```
   CALL ShowCmdLine  : DO : LOOP
   ```
5. Press ENTER to execute the line you just typed.

   The `ShowCmdLine` procedure adds the QCARDS command line to the parts already displayed. This time, however, QuickBASIC does not immediately display the

   ```
   Press any key to continue
   ```

   prompt, because the `DO` and `LOOP` statements following `ShowCmdLine` create an unconditional loop (that is, an infinite loop).

Breaking an Unconditional Loop from QCARDS

While building QCARDS, you will need to escape from unconditional loops because QCARDS does not yet respond to key presses. However, QuickBASIC checks for one particular key-press combination, CTRL+BREAK, which lets you break out of an infinite loop and return to the QuickBASIC environment. Note that when you finish adding code to QCARDS, a user will be able to press any of the high-intensity letters on the QCARDS command line to execute commands. For example, the Q key will terminate the program and return control to QuickBASIC because QCARDS will check each key the user presses and carry out the requested action. Try this to break an unconditional loop:

➤ Press CTRL+BREAK to return control to QuickBASIC.

Notice that when you press CTRL+BREAK, the cursor does not return to the Immediate window (where the last statement you executed is located). Instead, it is placed in the View window.

The Module-Level Code

Every QuickBASIC program has a module level—it's what QuickBASIC displays in the View window when you first open the program. Most of the code you will add to QCARDS is called "module-level code." It uses BASIC statements and functions (in combination with QCARDS procedures), to carry out the program's tasks, which are described in the comments. The module-level code you add to QCARDS defines the general way QCARDS interacts with the user.

Structured Programming with Procedures

You can write programs that have only a module level. However, the details of each task can obscure the general program logic, making the program difficult to understand, debug, and enhance. For ease of understanding and debugging, QCARDS delegates most processing tasks to procedures (such as `DrawCards` and `ShowCmdLine`) that are called when their tasks need to be performed. Figure 5.2 illustrates the relationship between module-level code and procedures and their execution sequences.

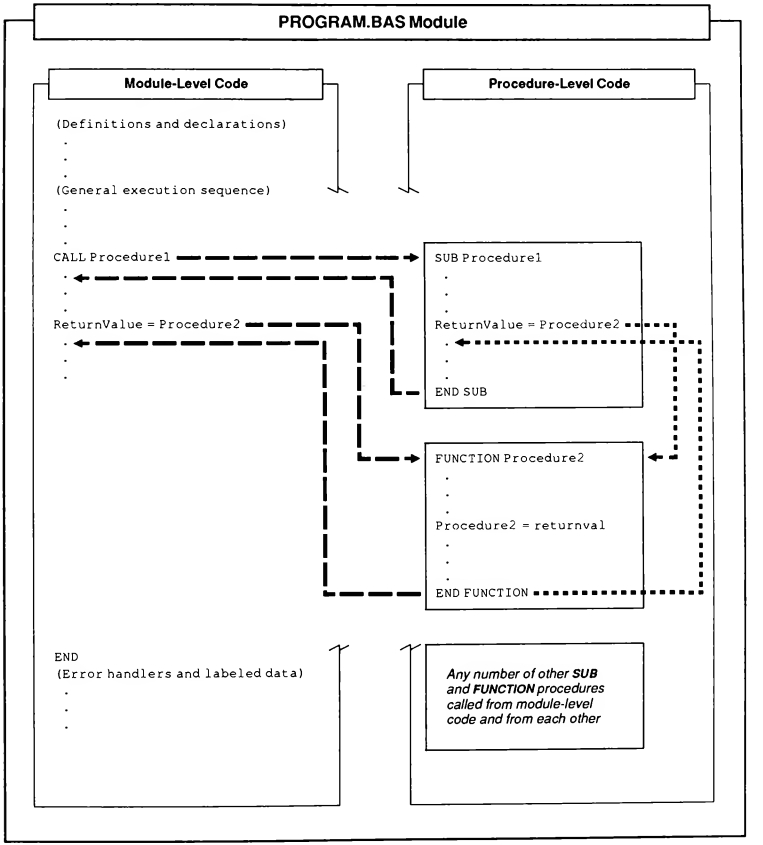


Figure 5.2 Modules and Procedures

A procedure is usually a group of statements that performs a single task. Procedures are saved as part of the same disk file as the module-level code. However, QuickBASIC treats procedures as logically self-contained entities, separate from the module-level code and from each other. Although every program has a module level, a procedure level does not exist until a procedure is actually defined. QuickBASIC executes the statements in a procedure only if the procedure is explicitly invoked. You can invoke procedures from the module-level code, from other procedures, or from the Immediate window.

A Profile of the Parts of the Program

Even though they are all in the same disk file, QuickBASIC keeps a program's procedures separate from the module-level code and from each other. Therefore, you can't have a procedure and module-level code (or two procedures) in the same window at the same time. To edit these different parts of a program, you move them in and out of the View window using the View menu's SUBs command. The SUBs dialog box lists all the procedures in a program. Try the following to see a profile of all the parts of QCARDS:

1. Press ALT+V, then press S to choose the View menu's SUBs command.

   The SUBs dialog box lets you access all parts of your program (see Figure 5.3). The first entry, `QCARDS.BAS`, is a file name. It represents the program's module-level code.

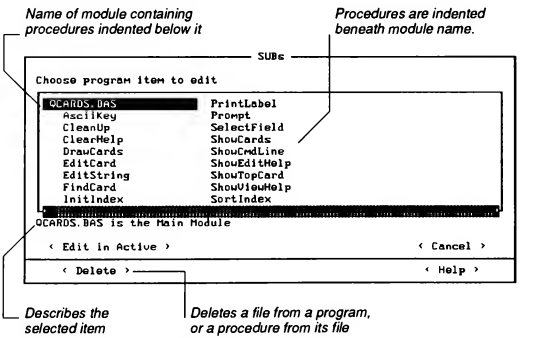


Figure 5.3 SUBs Dialog Box

2. Press DOWN to select the first entry beneath `QCARDS.BAS`.

   The first entry beneath `QCARDS.BAS` is `AsciiKey`. Each entry indented beneath `QCARDS.BAS` represents a **SUB** or **FUNCTION** procedure used by the program. All these procedures are saved in the QCARDS.BAS disk file.

3. Select the `DrawCards` procedure and press ENTER.

   QuickBASIC displays the code of the `DrawCards` procedure in the View window. Note that the View window's title bar contains

   ```
   QCARDS.BAS:DrawCards
   ```

   Use DIRECTION keys to scroll around in `DrawCards`. Its text is the only text in the View window.

4. Press ALT+V, then press S to choose the View Menu's SUBs command and select `QCARDS.BAS` if it is not already selected.

5. Press ENTER to return the module-level code to the View window.

QuickBASIC lets you define two types of procedures: **SUB** and **FUNCTION** procedures. To include the code of a procedure in your program's execution sequence, place its name wherever you want its task performed. You invoke a **SUB** procedure by using its name in a **CALL** statement. **FUNCTION** procedures are used the way QuickBASIC's intrinsic functions (like **TIME$**) are. Procedures are an effective way to code tasks that are performed repeatedly.

Defining a Procedure in QCARDS

You can start learning about procedures by writing simple ones. Do the following to see how QuickBASIC automatically segregates each procedure from the rest of the program. It also shows you how to define and call a simple procedure that you'll use later in QCARDS.

QCARDS Code Entry 1

1. Place the cursor at a blank line at the end of the declarations and definitions section in the View window (about line 102).

2. Press ENTER to create another blank line.

3. Type the words `SUB Alarm` and press ENTER.

   QuickBASIC opens a new window and starts the procedure with the statements `SUB Alarm` and `END SUB`. The statement `DEFINT A-Z` appears above the `SUB` statement because when you create a procedure, it inherits the default data type of the module-level code.

   The title bar in the View window now contains the name of the module (`QCARDS.BAS`) plus `Alarm`, the name of the procedure in the View window.

4. Type the following comment and code blocks, exactly as shown, between the `SUB` and `END SUB` statements:

```
' The Alarm procedure uses the SOUND statement to send signals
' to the computer's speaker and sound an alarm.
'
'
' Parameters: None
'
' Output: Sends an alarm to the user

' Change the numbers to vary the sound.
FOR Tone = 600 TO 2000 STEP 40
    SOUND Tone, Tone / 7000
NEXT Tone
```

   You can test the procedure and see what it does by calling it from the Immediate window.

5. Move the cursor to the Immediate window. Then type `CALL Alarm` and press ENTER.

Saving Edited Text

From now on, each time you save your work you will save the program with the name of the current chapter. When you save a program with a different name, the old version (QCARDS.BAS, in this case) still exists unchanged on disk. Therefore, if something makes a succeeding version of the program unusable, you can reload the previous version of the program and start over again.

QCARDS Code Entry 2

1. Press ALT+F, then press A to choose the File menu's Save As command.

   The Save As dialog box appears (see Figure 5.4).

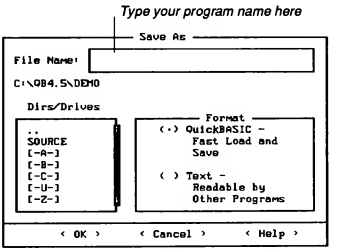


Figure 5.4 Save As Dialog Box

2. Type the name `chapter5` in the text box and press ENTER.

   This saves the program `QCARDS.BAS` as `CHAPTER5.BAS` on disk. The `Alarm` procedure that you created is saved as part of `CHAPTER5.BAS`.

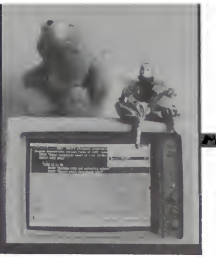
For More Information

For more information on the topics covered in this chapter, see the following:

Where to Look	What You'll Find
Chapter 11, "The File Menu"	Complete information on file management in QuickBASIC, including file-format options and special commands for opening types of files other than single-file programs.
Chapter 13, "The Edit Menu" and Chapter 14, "The View Menu"	Complete information on creating and viewing procedures in QuickBASIC.
Chapter 2, "SUB and FUNCTION Procedures," in *Programming in BASIC*	Explanations of BASIC's rules for creating all types of **SUB** and **FUNCTION** procedures.

CHAPTER 6

Editing in the View Window



QuickBASIC's editing features let you write and modify programs quickly, easily and accurately. As you edit, QuickBASIC checks your code to make sure it is valid BASIC syntax. In this chapter you will

- Discover QuickBASIC's smart editor
- Understand QuickBASIC's automatic text formatting
- Use on-line help to answer questions on the spot
- Use the Edit menu's Cut, Copy, and Paste commands
- Search for and replace text

You'll enter module-level code in QCARDS. If you have questions about any words you are typing, use on-line help (position the cursor on the word and press F1). On-line help includes complete descriptions and examples of each QuickBASIC statement and function. QuickBASIC also gives information on any variable, procedure, or symbolic constant at the cursor when you press F1. When you finish, press ESC to close the Help window.

This chapter requires one to two hours to complete.

The Smart Editor

One thing that makes editing in QuickBASIC different from using an ordinary word processor is its "smart editor." QuickBASIC recognizes language keywords and warns you when they are being used incorrectly. QuickBASIC also inserts certain types of text into the program for you. For example, when you created the `Alarm` procedure in the last chapter, QuickBASIC inserted the `DEFINT A-Z` and `END SUB` statements for you when you entered the `SUB Alarm` statement.

Automatic Formatting

QuickBASIC's smart editor capitalizes keywords and places uniform spacing between operators (such as plus and minus signs) and their operands (the values they operate on). This makes your program easier to read. The following steps illustrate this:

QCARDS Code Entry 3

1. Start QuickBASIC and open `CHAPTER5.BAS` (if it is not open).
2. Use PGDN and DOWN to move the cursor to the blank line beneath the following comment (at about line 106). Check the line counter on the reference bar to find the line.

```
' Open data file QCARDS.DAT for random access using file #1
```

3. Press ENTER to create a blank line, then type this statement exactly as shown, including the initial "comment delimiter" (the apostrophe preceding `open`):

```
'open "qcards.dat" for random as #1 len=len(card)
```

4. Press ENTER.

   QuickBASIC reads the line as though it were just a comment so no formatting is done.

5. Press UP to place the cursor at the comment delimiter.
6. Press DEL to delete the comment delimiter.
7. Press DOWN to move the cursor off the line.

   QuickBASIC converts the keywords (`OPEN`, `FOR`, `RANDOM`, `AS`, and `LEN`) to capital letters. Spaces are inserted on each side of the equal sign (=).

Since it is easier to type in lowercase letters, lines of code are shown in this tutorial with keywords in lowercase. When you type them and press ENTER, QuickBASIC capitalizes the keywords for you. Words that aren't keywords (such as symbolic constants, procedure names, and variable names) appear in mixed case or all uppercase letters.

The BASIC language is not "case sensitive." It doesn't distinguish between two versions of the same word by the case of the letters. For instance, the keyword `OPEN` is the same to the BASIC language as `open` or `OpEn`. When the smart editor capitalizes keywords, it is just for appearance. The variable name `Card` represents the same memory location as `CArd` or `CaRD`.

The smart editor automatically maintains consistency in the capitalization of specific words in a program listing. Conventions used in this manual and QuickBASIC's handling of text you enter are summarized in the following list:

Type of Word	Appearance in Manual	QuickBASIC Action
BASIC keywords	Lowercase	Converts them to all uppercase letters when you enter the line
Variable names and procedure names	Mixed case	Alters all instances of the name to most recent capitalization
Symbolic constants	Uppercase	Changes all instances of the name to most recent capitalization

The following steps illustrate the rules in the table above:

QCARDS Code Entry 4

1. Move the cursor to the line below the following comment (at about line 113):

```
' To count records in file, divide the length of the file by the
' length of a single record; use integer division (\) instead of
' normal division (/). Assign the resulting value to LastCard.
```

2. Press ENTER to make space, then type the following line exactly as shown:

```
LastCard=LoF(1)\Len(Card)
```

   Note that the backslash (\) operator in BASIC performs integer division, rather than normal division.

3. Press ENTER.

   Keywords are capitalized and spaces are inserted. The previous capitalization of `card` (in the `OPEN` statement) is updated to `Card`, to conform to the way you just typed it. QuickBASIC assumes that the last way you entered the word in the View window is the way you want it everywhere. This is done to *all* occurrences of the word except those within comments.

Syntax Checking

When you move the cursor off any code line you edit (either by pressing ENTER or a DIRECTION key), QuickBASIC checks its syntax. If the line is not valid BASIC code, QuickBASIC displays a dialog box describing the error.

The following steps show how the smart editor checks your code for syntax errors:

QCARDS Code Entry 5

1. Move the cursor to the line beneath the following comment block (about line 130):

```
' Redefine the Index array to hold the records in the file plus
' 20 extra (the extra records allow the user to add cards).
' This array is dynamic - this means the number of elements
' in Index() varies depending on the size of the file.
' Also, Index() is a shared procedure, so it is available to
' all SUB and FUNCTION procedures in the program.
'
' Note that an error trap lets QCARDS terminate with an error
' message if the memory available is not sufficient. If no
' error is detected, the error trap is turned off following the
' REDIM statement.
```

2. Press ENTER, then type the following lines exactly as shown (including errors):

```
on error goto MemoryErr
redim shared Index(1 to LastCard + 20) as PERSOR
on error goto
```

3. Press ENTER after the final line. QuickBASIC displays the following error message (see Figure 6.1):

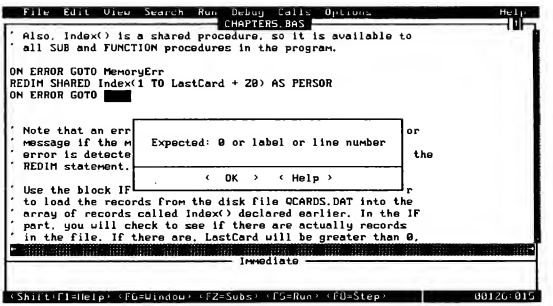


Figure 6.1 Error-Message Dialog Box

When you use **ON ERROR GOTO** to turn off an error trap (as is being done here) QuickBASIC expects it to be followed by the number 0.

4. Press ESC to clear the dialog box.

   QuickBASIC places the cursor at the place the error was detected.

5. Add a zero at the end of the line so it looks like this:

```
ON ERROR GOTO 0
```

Once you close an error-message dialog box, QuickBASIC doesn't check that statement's syntax again until you actually change the line. If you don't edit the line at all, QuickBASIC waits until you run the program before reminding you that the error still exists.

Errors Detected When You Run Your Program

Some errors are not apparent until you actually try to run your code. For example, in the `REDIM` statement you declared the `Index()` array as type `PERSOR`. This was a misspelling of the name of the user-defined type, `PERSON`, which you saw when you scrolled through the declarations and definitions section of QCARDS in Chapter 5 "The QCARDS Program." QuickBASIC did not detect this error on entry because it assumes that the `PERSOR` type is declared somewhere, even though it has not yet encountered the declaration.

The following step illustrates how these errors are detected:

➤ Press ALT+R, then press S to choose the Run menu's Start command to run the lines entered so far.

QuickBASIC displays the error message `Type not defined` and highlights the `AS` in the `REDIM` statement as shown in Figure 6.2.

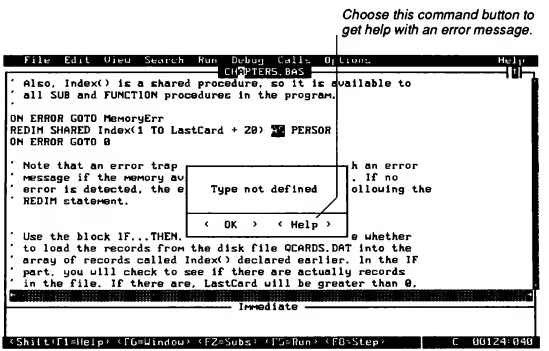


Figure 6.2 Error Message

Help for Error Messages

Error-message dialog boxes contain a Help button you can choose to get either a description of common causes for the error or the correct syntax plus examples of usage.

1. Press F1 to get on-line help for this error message, then choose the Help button in the dialog box.

   The following dialog box appears (see Figure 6.3):

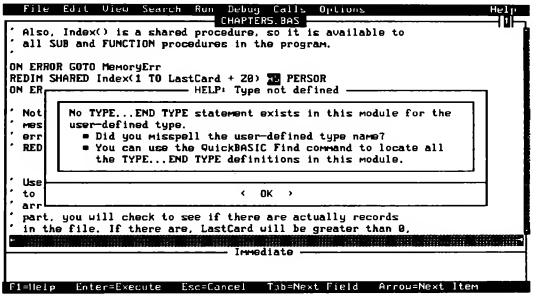


Figure 6.3 Help on Error Message

   The new dialog box indicates that no **TYPE...END TYPE** statement was found for PERSOR.

2. Press ESC as many times as necessary to clear the dialog boxes.

Overtyping the Error

Overtyping replaces the text at the cursor and moves the cursor right. Correct the error by overtyping, as follows:

QCARDS Code Entry 6

1. Place the cursor on the second `R` in `PERSOR`.
2. Press INS to change from insert to overtype.

   The INS key is a "toggle." It changes the shape of the cursor from the default (a thin blinking line) to a blinking box. When the cursor is a thin line, text you type is inserted at the cursor position. When the cursor has the box shape, whatever you type "overtypes" (replaces) the text at the current cursor position.
3. Overtype the second `R` in `PERSOR` with an `N` so the line looks like this:

   ```
   REDIM SHARED Index(1 TO LastCard + 20) AS PERSON
   ```
4. Press INS to reset the toggle to its default (the thin line).
5. Type the following on the next blank line to call your `Alarm` procedure:

   ```
   call Alarm
   ```
6. Press ALT+R, then press S to choose the Run menu's Start command to run the lines.

   QCARDS doesn't produce any visible output yet, but `Alarm` produces its sound.
7. Press a key to return to the QuickBASIC environment.

Automatic Procedure Declarations

When QuickBASIC encounters a procedure invocation in a program, it creates the **DECLARE** statement automatically. Then, it writes the declaration at the first line of the file (when you save your program). The following steps illustrate this:

QCARDS Code Entry 7

1. Press CTRL+HOME to move the cursor to the beginning of the program.
2. Press ALT+F, then press A to choose the File menu's Save As command. The Save As dialog box appears.

3. Type `chapter6` in the text box, then press ENTER to save the program as `CHAPTER6.BAS`. QuickBASIC inserts the declaration on the first line of the program, as follows:

```
DECLARE SUB Alarm()
```

This is a good point to take a break. In the next section you'll move the declaration just created and change some of the existing QCARDS code.

Working with Selected Text Blocks

"Selecting" is the way you highlight a block of text to work with. To select a text block directly, place the cursor at the beginning of a block, then hold down the SHIFT key while pressing the appropriate DIRECTION key.

Try the following to select a block of text:

QCARDS Code Entry 8

1. Start QuickBASIC (if you exited when you took a break) and open `CHAPTER6.BAS`.
2. Press CTRL+HOME to make sure the cursor is at the program's first line.

   The line should be `DECLARE SUB Alarm()`.
3. Press SHIFT+DOWN to select this line.

The only way you can remove the highlight from selected text and return it to its unselected state is by pressing one of the DIRECTION keys (UP, DOWN, LEFT, RIGHT), or by pressing F6 to move to a different window.

WARNING *Be cautious with selected text! If you type any character (even the SPACEBAR) while a line of text is selected, whatever you type replaces all selected text in the active window. The text that was selected is then permanently lost.*

Cutting or Copying Selected Text

The Edit menu has commands that let you move a block of selected text to and from a "Clipboard" (a dedicated space in memory). The Cut and Copy commands put selected text on the Clipboard. The Paste command copies whatever text is on

the Clipboard to the current cursor position. If a line (or more) of text is selected in the active window when you choose the Paste command, the text on the Clipboard replaces the selected text.

When a menu command is available, it is marked with a high-intensity letter. For example, if you open the Edit menu now, the Cut and Copy commands are available because there is text selected in the active window. However, the Paste command is not available unless you have already placed text on the Clipboard. The following steps use the Cut command to transfer the text from its current position to the Clipboard:

QCARDS Code Entry 9

1. Select the line `DECLARE SUB Alarm` if it isn't already selected.
2. Press ALT+E to open the Edit menu if it is not already open.
3. Press T to choose the Cut command.

The selected text is transferred to the Clipboard. You can only copy one block at a time to the Clipboard because each new block (regardless of its size) replaces everything that preceded it on the Clipboard.

IMPORTANT *The shortcut-key combination for Cut is SHIFT+DEL. If you just press DEL while text is selected, it is deleted, but not to the Clipboard. You cannot recover text deleted with DEL.*

The Copy command is similar to Cut, but it puts a duplicate of the selected text on the Clipboard and leaves the selected block as is.

NOTE *Copy leaves the original text block selected, so you have to press a DIRECTION key before typing anything or the block will be replaced by what you type.*

Pasting a Text Block

Any text you put on the Clipboard remains there until you either put something else on the Clipboard or until you exit from QuickBASIC. Follow these steps to move to the procedure-declarations section of QCARDS and paste in the declaration for `Alarm`:

QCARDS Code Entry 10

1. Press ALT+S, then press F to choose the Search menu's Find command.

   The Find dialog box appears (see Figure 6.4).

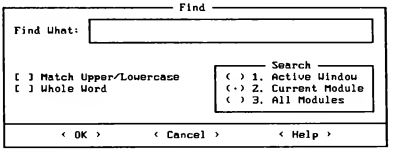


Figure 6.4 Find Dialog Box

2. Type the following line in the text box:

   ```
   This space reserved
   ```

3. Press ENTER to start the search.
4. Press HOME to place the cursor in column 1.
5. Press SHIFT+END to select the whole line.
6. Press ALT+E, then press P to choose the Edit menu's Paste command.

   The Paste command is active because there is text on the Clipboard. The text from the Clipboard replaces the selected text.

7. Save the program as `CHAPTER6.BAS`.

Manipulating Selected Text

You can manipulate selected text in other ways. For instance, you can indent selected text as a block rather than one line at a time. The following steps illustrate this:

QCARDS Code Entry 11

1. Press ALT+S, then press F to choose the Search menu's Find command. Type the following line in the text box, exactly as shown:

   ```
   ' Define temporary
   ```

   There may be text selected in the text box when the dialog box opens. If so, whatever you type replaces the selected text automatically. You can remove the selection from the text by pressing a DIRECTION key if you want to just change the text in the box rather than completely replacing it.

2. Press ENTER. The cursor should move to the following line:

   ```
   ' Define temporary Index() array to illustrate QCARDS screen
   ```

3. Press SHIFT+DOWN twice so both the following lines are selected:

   ```
   ' Define temporary Index() array to illustrate QCARDS screen
   REDIM SHARED Index(1) AS PERSON
   ```

4. Press TAB while this text block is selected. The whole block moves right by one tab stop.

5. Press SHIFT+TAB to move the block left by one indentation level.

6. Press DEL to delete the entire block permanently.

7. Save the program as usual as `CHAPTER6.BAS`.

Note that the text you just deleted was not placed on the Clipboard. This is because you deleted it with DEL rather than SHIFT+DEL (the shortcut key for the Cut command). The temporary `Index()` array definition is no longer needed because you defined it (with a `REDIM` statement) in the first part of this chapter.

Searching and Replacing Text

IMPORTANT *The remaining sections of this chapter provide practice using QuickBASIC's search and replace feature. The code you enter in the optional code-entry sections below is not necessary to make QCARDS run. You may want to read the rest of the chapter before deciding whether to try the optional code-entry sequences. If you make a mistake entering any of the optional sequences, the final program won't run properly. If you choose to do any of the optional code-entry sequences, you must do all three of them.*

In this section, you'll define the symbolic constant `DISKFILE$` to represent `"QCARDS.DAT"` the name of the disk file that contains QCARDS' data. Then you'll use the Search menu's Change command to replace all instances of the text string `"QCARDS.DAT"` with the symbolic constant. Once you define the constant `DISKFILE$`, you can just change the definition of `DISKFILE$` if you want to load a different data file in QCARDS (or create a new data file).

Defining the Symbolic Constant

The first thing to do is define the constant. There is already an appropriate place to do this in the declarations and definitions section. Do the following to find the place and then add the constant:

Optional QCARDS Code Entry 1

1. Save `CHAPTER6.BAS` as `OPTIONAL.BAS`.

   This way, if you do everything right you can start the next chapter by loading `OPTIONAL.BAS`, then saving it as `CHAPTER7.BAS`. If you make an error in one of the optional sections and find you can't correct it, you can start with `CHAPTER6.BAS` in the next chapter.

2. Press CTRL+HOME to move to the beginning of QCARDS.

3. Press ALT+S, then press F to choose the Search menu's Find command. Finally, type the following in the text box:

   ```
   TMPFILE
   ```

4. Press ENTER to place the cursor at the definition of `TMPFILE`. It looks like this:

   ```
   CONST TMPFILE$ = "$$$87y$.$5$"        ' Unlikely file name
   ```

5. Press DOWN to remove selection from `TMPFILE`.
6. Press HOME to place the cursor in the leftmost column, then type the following line, exactly as shown, but don't press ENTER:

```
const DISKFILE$ = "QCARDS.DAT"
```

7. Press SHIFT+LEFT until the string `"QCARDS.DAT"` is completely selected. Be sure both quotation marks are included in the selection.

It's important to make sure the quotation marks are selected because you want to replace `"QCARDS.DAT"` with `DISKFILE$`.

Replacing Multiple Occurrences of Text

Using the Change command is much like the Find command, except that QuickBASIC lets you substitute specified text when matching text is found before it searches for the next match. Next you will change `"QCARDS.DAT"` to `DISKFILE$` everywhere except in the definition you wrote in the last section. The Search menu's Change command lets you automate the process.

Optional QCARDS Code Entry 2

1. Press ALT+S, then press C to choose the Search menu's Change command. The Change dialog box's Change command appears (see Figure 6.5):

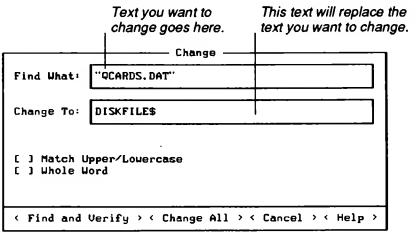


Figure 6.5 Change Dialog Box

Because `"QCARDS.DAT"` is selected in the View window, the Find What text box has `"QCARDS.DAT"` in it.

2. Press TAB to move the cursor to the Change To text box.

   This is where you enter the text you want to substitute for the search text.

3. Type `DISKFILE$` in this text box.

4. Press ENTER to choose the Find and Verify command button (the default).

WARNING *The second command button, Change All, makes all the changes without showing you the occurrences or waiting for confirmation. This option is risky unless you are sure you want to change all the occurrences. Even then, you may inadvertently change words that contain the search text in a way you didn't anticipate. If you have any doubts, use the default option Find and Verify. The Cancel button or ESC aborts the search and replace operation entirely.*

The Find and Verify command button highlights each occurrence of the search text, then displays the Change, Skip, Cancel dialog box (see Figure 6.6). In the next section you'll use it to make selective replacements. It contains buttons you can choose when you decide whether to replace the current occurrence:

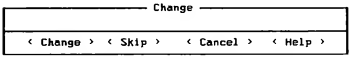


Figure 6.6 Change, Skip, Cancel Dialog Box

The Change, Skip, Cancel dialog box should still be on your screen. Replace occurrences of `"QCARDS.DAT"` with `DISKFILE$` as follows:

Optional QCARDS Code Entry 3

1. Press C to change the first occurrence. It is in the `OPEN` statement you entered earlier in this chapter.

   QuickBASIC moves to the next match. It is in a comment line in the `CleanUp` procedure (check the View window's title bar).

2. Press C to make the change.

   QuickBASIC moves to the next match. It is also in `CleanUp` in the line `KILL "QCARDS.DAT"`

3. Press C to make the change.

   QuickBASIC moves to the next match. It is also in `CleanUp` in the line `NAME TMPFILE AS "QCARDS.DAT"`

4. Press C to make the change.

   QuickBASIC moves to the next match. It is in a different procedure, `InitIndex`, in a comment line.

5. Press C to make the change.

   The final match is at `CONST DISKFILE$ = "QCARDS.DAT"` where you started. You *don't* want to change this one because this definition is the reason you just made all the other changes.

6. Press S to choose the Skip command button.

   When all occurrences have either been changed or skipped, QuickBASIC displays a dialog box containing `Change Complete`.

7. Press ESC to close the dialog box.

Checking Your Work

Whenever you make a series of changes throughout a program, you should check your work by running the program. If you've made a mistake during editing, you can use QuickBASIC's error messages to help you track down mistakes. Finish this session with the following procedure:

1. Press ALT+R, then press S to choose the Run menu's Start command to run the lines.

   If you get an error message, try to track down the error and fix it before you end the session. Go back over each of the optional QCARDS code-entry sections and compare your work with them.

2. Press a key to return to the QuickBASIC environment.

3. Save the program. If you renamed it `OPTIONAL.BAS`, you can just start with that file in the next chapter and rename it `CHAPTER7.BAS`.

The constant `DISKFILE$` now represents `QCARDS.DAT` in the program. Later you may want to modify the program to allow the user to load different files while the program is running. If you do, you can't represent the the file with the constant `DISKFILE$`, because the point of a constant is that it cannot be changed when the program is running. You would have to remove the `CONST` part of the definition and just make `DISKFILE$` an ordinary string variable.

For More Information

For more information on the topics covered in this chapter, see the following:

Chapter	Topic
Chapter 12, "Using the Editor," and Chapter 13, "The Edit Menu"	Complete information on QuickBASIC's editing features, including the commands found on Full Menus and editing shortcut-key combinations
Chapter 2, "SUB and FUNCTION Procedures," in *Programming in BASIC*	Defining and using procedures and placement of automatically generated procedure declarations.
Chapter 4, "String Processing," in *Programming in Basic*	Using strings as symbolic constants
Chapter 6, "Error and Event Trapping," in *Programming in BASIC*	Using the **ON ERROR** statement

CHAPTER 7

Programming with On-Line Help

You've seen Microsoft QuickBASIC's on-line help features in previous chapters. In this chapter you'll use on-line help to guide you in writing some code for the QCARDS program. You will

- Use alternative methods to get on-line help for QuickBASIC keywords
- Access information about on-line program symbols through help
- Use hyperlinks to get additional information within the Help window

This chapter takes about one hour to complete.

Using On-Line Help to Construct Statements

In this section you'll use the Help menu's Index command to find out how to use statements. Do this first:

QCARDS Code Entry 12

1. Start QuickBASIC if it is not running and open `CHAPTER6.BAS` (or `OPTIONAL.BAS`, whichever is appropriate).
2. Save the program immediately as `CHAPTER7.BAS`.

On-Line Help for Keywords

If you know the statement you want to use, you can find it quickly by choosing the Index command from the Help menu. You can get information on the first code block you'll enter in this chapter (the **IF...THEN...ELSE** statement) by doing the following steps:

1. Choose the Help menu's Index command (press ALT+H, then press I).

   Index lists all the QuickBASIC keywords. Scroll down through the list to see what it contains (use DIRECTION keys), or press the initial letter of the keyword you want more information on.

2. Select `IF... THEN` (press I).

   QuickBASIC places the cursor in the `IF...THEN` statement

3. Get help on `IF..THEN` (press F1).

   A QuickSCREEN appears (see Figure 7.1). It includes a brief description of the statement, the syntax for the block form of **IF...THEN...ELSE**, and a description of the terms used in the block.

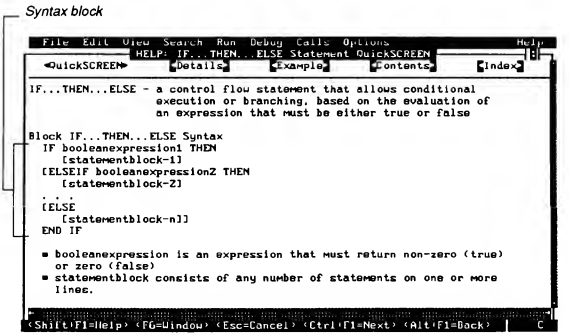


Figure 7.1 QuickSCREEN for IF...THEN...ELSE Statement

4. Make the Help window fill the whole screen (press CTRL+F10).
5. Move down to the syntax for the single-line form of `IF...THEN...ELSE` (press PGDN).
6. Move the cursor back to the top of the QuickSCREEN (press CTRL+HOME).
7. Restore the View and Immediate windows (press CTRL+F10).

If you've used the statement before, the QuickSCREEN help information is usually sufficient to help you write your code.

Hyperlinks in On-Line Help

The QuickSCREEN includes four hyperlinks across the top. You can move the cursor to any one by pressing the TAB key or the initial letter of the word in the hyperlink. The following list describes what is displayed by each of the hyperlinks:

Hyperlink	Information Displayed by Help
Details	Gives full explanation of each syntax element and describes all ways of using the statement
Example	Shows code examples that illustrate syntax use
Contents	Shows all possible categories of on-line help available in QuickBASIC
Index	Lists QuickBASIC keywords alphabetically

Try each of the available hyperlinks in turn, pressing ALT+F1 each time to return to the original syntax screen.

If you keep choosing several hyperlinks without returning to the previous screen, you can work your way backwards through the hyperlink sequence by pressing ALT+F1 repeatedly (up to a limit of 20 links).

Although you can't type in the Help window, you can size it with ALT+PLUS and ALT+MINUS so that the information you need is displayed while you enter text in the View window. When you understand the block **IF...THEN...ELSE** statement, implement the programming described in the comment block as follows:

QCARDS Code Entry 13

1. Place the cursor in the View window (press F6).
2. Scroll down to the line beneath the following comment block (about line 145) and create a line of space (press ENTER):

```
' Use the block IF...THEN...ELSE statement to decide whether
' to load the records from the disk file QCARDS.DAT into the
' array of records called Index() declared earlier. In the IF
' part, you will check to see if there are actually records
' in the file. If there are, LastCard will be greater than 0
' and you can call the InitIndex procedure to load the records
' into Index(). LastCard is 0 if there are no records in the
' file yet. If there are no records in the file, the ELSE
' clause is executed. The code between ELSE and END IF starts
' the Index() array at card 1.
```

3. Type the following code, exactly as shown:

```
if LastCard <> 0 then
    call InitIndex(LastCard)
else
    Card.CardNum = 1
    Index(1) = Card
    put #1, 1, Card
end if
```

4. Save the program as `CHAPTER7.BAS`. Then place the cursor back on the `ELSE` keyword.

5. Run your program up to the line on which the cursor rests (press F7).

   The line at the cursor appears in high-intensity video with the colors you chose for the Current Statement option in the Options menu's Display dialog box. This means that the program is still running, but execution is temporarily suspended.

On-Line Help for Program Symbols

QuickBASIC provides help for all "symbols" in your program. A symbol is anything that is defined, either explicitly or implicitly. For example, QuickBASIC keywords are symbols defined by the language. To get help for the symbols you define in your programs, place the cursor anywhere in the symbol, then press F1. QuickBASIC displays all the information it has about the symbol and where it is used in the program. Try this to find out about `Card.CardNum`.

1. Place the cursor in the word `CardNum` in `Card.CardNum` and press F1.

   The Help window shown in Figure 7.2 appears.

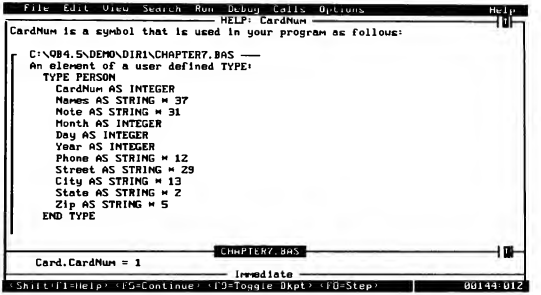


Figure 7.2 Symbol Help for CardNum

On-line help describes `CardNum` as an element of the `PERSON` user-defined type. The help displays the actual **TYPE...END TYPE** statement. `CardNum` is the first element.

2. Move the cursor to the left until it is in the word `Card` on the left side of the period, then press F1. On-line help shows that the name `Card` is used many places in QCARDS. The specific instance at the cursor when you pressed F1 is described first; it is described as a variable of type `PERSON`.

   Note that in several procedures, `Card` is also used as a variable name, but is a variable of type `INTEGER`. Also, `Card` is sometimes described as a parameter of type `PERSON`.

On-line help makes it easy to track the use of variable names throughout your programs. Although on-line help doesn't include the current value of a variable, QuickBASIC makes it easy to determine the current value of a variable in a suspended program. Here's how:

1. Close the Help window (press ESC).
2. Place the cursor anywhere in the word `LastCard` (about line 150).
3. Choose the Debug menu's Instant Watch command.

   QuickBASIC displays the Instant Watch dialog box (see Figure 7.3). It contains the name of the variable `LastCard` and its current value. (The actual value reflects the number of records in the `QCARDS.DAT` data file.)

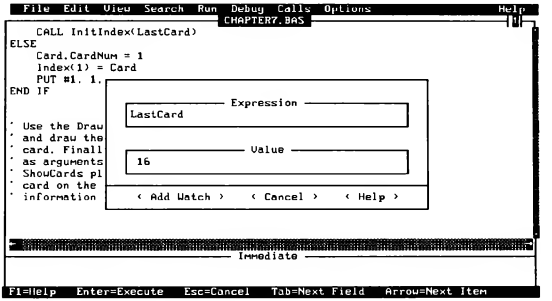


Figure 7.3 Instant Watch Dialog Box

4. Clear the Instant Watch dialog box (press ESC).

Perform the following steps to place the QCARDS cards on the screen and the data from QCARDS.DAT on the cards:

QCARDS Code Entry 14

1. Place the cursor on the line beneath the following comment (about line 163):

```
' Use the DrawCards procedure to initialize the screen
' and draw the cards. Then, set the first card as the top
' card. Finally, pass the variables TopCard and LastCard
' as arguments to the ShowCards procedure. The call to
' ShowCards places all the data for TopCard on the front
' card on the screen, then it places the top-line
' information (the person's name) on the remaining cards.
```

2. Create a line of space (press ENTER), then type the following code lines exactly as shown:

```
call DrawCards
TopCard = 1
call ShowCards(TopCard, LastCard)
```

3. Place the cursor back on the last line you entered (the one beginning with `CALL ShowCards`).
4. Execute the program to the line at the cursor (press F7).
5. Choose the View menu's Output Screen command to toggle back and forth between the QuickBASIC environment and the QCARDS output screen (press F4).

You can use the same techniques as before to get help for the **CALL** keyword that transfers control to a BASIC **SUB** procedure. You can also get help for the `DrawCards` and `ShowCards` procedures and for the variable named `TopCard`. Since the program is still running (though suspended) you can also check the values of its variables using the Debug menu's Instant Watch command.

Printing Screens from On-Line Help

The Print command on the File menu can make hard copies of text from on-line help. If you have a printer connected to the parallel port, you can print hard copy of any of the help screens as follows:

1. Choose the Help menu's Index command and move the cursor to the `DO...LOOP` entry. Then move that statement's on-line help into the Help window (press F1).

2. Choose the File menu's Print command (press ALT+F, then press P).

   When the Help window is the active window, you can choose the option to print either specifically selected text or the whole current help screen (see Figure 7.4).

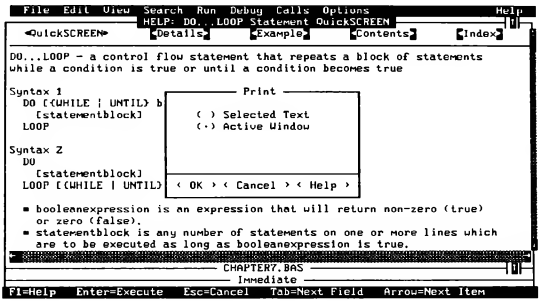


Figure 7.4 Print Dialog Box (Help)

3. Print this help screen if you have a printer (press ENTER), otherwise clear the dialog box (press ESC). (If you try to print when you don't have a printer, a `Device unavailable` dialog box may be displayed.)
4. Choose any hyperlinks you'd like to see, then print them out.
5. Close the Help window (press ESC).

Use the hard copy of the help text (or on-line help) to understand the programming. In the next steps, you'll insert the unconditional loop.

QCARDS Code Entry 15

1. Move the cursor to the line below this comment block (about line 178):

```
' Keep the picture on the screen forever with an unconditional
' DO...LOOP statement. The DO part of the statement goes on
' the next code line. The LOOP part goes just before the END
' statement. This loop encloses the central logic that lets
' a user interact with QCARDS.
```

2. Create a blank line (press ENTER).
3. Type the word `do` then press ENTER.
4. Move the cursor down to the line beneath this comment (at about line 226):

   ```
   ' This is the bottom of the unconditional DO loop.
   ```

5. Create a blank line (press ENTER), then type the word `loop`
6. Choose Run menu's Start command.

   You will need to break out of this unconditional **DO** loop.

7. Press CTRL+BREAK when you want to suspend execution of the unconditional **DO** loop.

   While execution is suspended, you can use commands like Instant Watch to examine the variables. Afterwards, you can continue program execution by choosing the Continue command from the Run menu (press ALT+R, then press N).

8. Save the program as `CHAPTER7.BAS`.

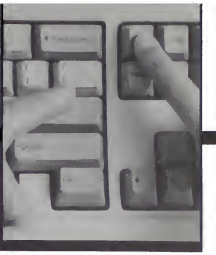
For More Information

For more information on on-line help, see the following:

Chapter	Topic
Section 10.8, "Using On-Line Help" and Chapter 21, "The Help Menu"	Information on QuickBASIC's Help menu commands and complete descriptions of all the Help features

CHAPTER 8

Using Example Code from On-Line Help



You've seen many facets of Microsoft QuickBASIC's on-line help in previous chapters. Each QuickSCREEN and Details screen for each statement and function contains a hyperlink to one or more code examples. In this chapter you will copy example code from on-line help to use in QCARDS. You'll use some of these examples as you

- Copy example code from on-line help and paste it into the View window
- Edit text copied from on-line help to work in a specific program

This chapter takes about one to one and one-half hours to complete.

Copying Example Code from On-Line Help

When you use on-line help for keywords, the QuickSCREEN describes the action and syntax of the keyword. If you have used the statement before, the syntax block may give all the information you need to write your code. However, if the statement is new to you, the Details and Example screens clarify how to use the syntax. In the next several code-entry sequences, you'll find an example in on-line help that you can copy and use in QCARDS. Example code from on-line help may not be a perfect match for QCARDS. Some code that you copy conflicts with code in the program. Don't worry—you'll fix it later in the tutorial.

QCARDS Code Entry 16

1. Place the cursor on the line following this comment block (about line 192):

```
' Get user keystroke with a conditional DO...LOOP statement.
' Within the loop, use the INKEY$ function to capture a user
' keystroke, which is then assigned to a string variable. The
' WHILE part of the LOOP line keeps testing the string
' variable. Until a key is pressed, INKEY$ keeps returning a
' null (that is a zero-length) string, represented by "".
' When a key is pressed, INKEY$ returns a string with a
' length greater than zero, and the loop terminates.
```

2. Create a blank line (press ENTER). Then, type do

3. Display the QuickSCREEN for the **DO** keyword (press F1).

   When you press F1 when the cursor is on a blank space, QuickBASIC displays context-sensitive help on the symbol at the left of the cursor. Note that the cursor remains in the View window.

4. Move the cursor into the Help window (press F6 twice).

5. Choose the Example hyperlink (press E, then press F1).

   The example screen appears in the Help window (see Figure 8.1):

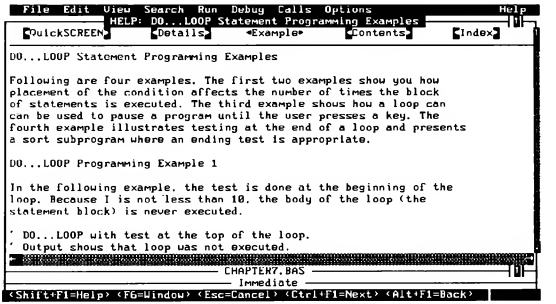


Figure 8.1 Example Screen for DO...LOOP Statement

The first paragraph of the screen describes the examples below. The description of the third example corresponds closely to the task you want to code into QCARDS at this point. Follow these steps to place the example code on the Clipboard:

QCARDS Code Entry 17

1. Scroll down to `Example 3`. It looks like this:

```
' DO...LOOP with test at the bottom of the loop.
  DO
      Choice$ = INKEY$
  LOOP WHILE Choice$ = ""
```

2. Select all four of the above lines.
3. Choose the Copy command from the Edit menu to place the lines on the Clipboard (press ALT+E, then press C).
4. Close the Help window (press ESC).
5. Select the `DO` you typed into the View window just before moving to the Help window (press SHIFT+HOME).
6. Choose the Paste command from the Edit menu (press ALT+E, then press P).

   The text from the Clipboard replaces the text that was selected.
7. Choose the Run menu's Start command to check for errors (press ALT+R, then press S), then return to QuickBASIC (press CTRL+BREAK).

You can use on-line help to check for potential conflicts between the code you just copied and QCARDS, as follows:

1. Place the cursor on `Choice$` and press F1.
2. On-line help shows you that variables named `Choice$` now occur twice in QCARDS, once in the module-level code and once in the `EditString` procedure. These don't conflict because variables local to a procedure are not known at the module level and vice versa.

Indenting a Block of Code

The loop you just entered is "nested" within the unconditional loop you entered earlier (in "QCARDS Code Entry 15," in Chapter 7, "Programming with On-Line Help"). This means that it is executed once each time the outer loop is executed. Nested code is usually indented one tab stop to the right of the code encompassing it to show visually the levels of nesting within your program. Here's a reminder of how to indent multiple text lines as a block:

1. Select the three code lines you just entered. They look like this:

```
DO
     Choice$ = INKEY$
LOOP WHILE Choice$ = ""
```

2. Move the whole block right one tab stop (press TAB).
3. Remove the selection from the block (press any DIRECTION key).

The rest of the code you enter in QCARDS is nested within the unconditional **DO** loop so each block should be similarly indented.

Copying Large Code Blocks from On-Line Help

The code you copied in the preceding section was short and simple. It contained only one variable name, so the likelihood of a conflict with an existing symbol in QCARDS was small.

You can copy larger blocks of example code from on-line help and paste them into programs you write. However, the code you copy may conflict with code you've already written into your program. In this section you'll copy more example code into QCARDS, then customize it significantly. Some of it will conflict, but you'll fix it later.

QCARDS Code Entry 18

1. Move the cursor to the line beneath the following comment block (about line 222):

```
' The ELSE clause is only executed if Choice$ is longer than a
' single character (and therefore not a command-line key).
' If Choice$ is not an ASCII key, it represents an "extended"
' key. (The extended keys include the DIRECTION keys on the
' numeric keypad, which is why QCARDS looks for them.) The
' RIGHT$ function is then used to trim away the extra byte,
' leaving a value that may correspond to one of the DIRECTION
' keys. Use a SELECT CASE construction to respond to those key-
' presses that represent numeric-keypad DIRECTION keys.
```

2. Create a blank line (press ENTER), then type

```
select case Choice$
```

3. Press HOME, then display help for the SELECT CASE statement (press F1).

4. The help screen shown in Figure 8.2 appears.

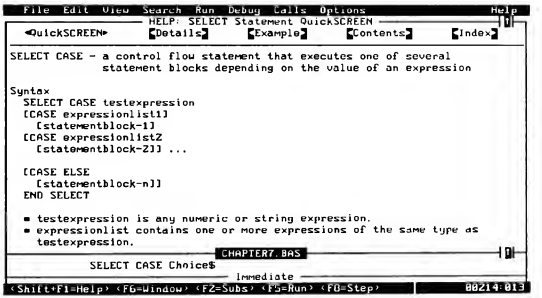


Figure 8.2 QuickSCREEN for SELECT CASE Statement

The **SELECT CASE** statement is commonly used like a switchboard in situations where there are numerous possible inputs, and many of them must be handled in unique ways. You don't want to take a unique action for each key a user might press, but with this kind of structure you can add **CASE** clauses for each key you want to handle. Then you can treat the rest of the possible keys as "none of the above" with **CASE ELSE**.

In the following steps you'll copy a large piece of code from on-line help:

QCARDS Code Entry 19

1. Move the cursor into the help window (press SHIFT+F6 twice).
2. Choose the Example hyperlink at the top of the screen to move the examples into the Help window (press E, then press F1).
3. Choose the Search menu's Find command (press ALT+S, then press F), type `Example 2` in the text box, then press ENTER.
4. Scroll down (about 15 lines) to the line: `IF LEN(Choice$) = 1 THEN`
5. Select this line, and the next 43 lines, all the way down to `END IF`. (Be careful *not* to include the final `LOOP` statement in the selected text.)
6. Choose the Copy command from the Edit menu (press ALT+E, then press C).
7. Close the Help window (press ESC). This puts the cursor back on `SELECT CASE` in the View window.
8. Select the line `SELECT CASE Choice$` (press SHIFT+END).
9. Choose the Paste command from the Edit menu (press ALT+E, then press P).

   The code pasted after the comment block should appear as follows:

```
IF LEN(Choice$) = 1 THEN
    ' Handle ASCII keys
       SELECT CASE ASC(Choice$)
           CASE ESC
               PRINT "Escape key"
               END
           CASE IS  32, 127
               PRINT "Control code"
           CASE 30 TO 29
               PRINT "Digit: "; Choice$
           CASE 65 TO 90
               PRINT "Uppercase letter: "; Choice$
           CASE 97 TO 122
               PRINT "Lowercase letter: "; Choice$
           CASE ELSE
               PRINT "Punctuation: "; Choice$
           END SELECT

ELSE
    ' Convert 2-byte extended code to 1-byte ASCII code and
    ' handle
       Choice$ = RIGHT$(Choice$, 1)

       SELECT CASE Choice$
           CASE CHR$(DOWN)
               PRINT "DOWN arrow key"
           CASE CHR$(UP)
               PRINT "UP arrow key"
           CASE CHR$(PGDN)
               PRINT "PGDN key"
           CASE CHR$(PGUP)
               PRINT "PGUP key"
           CASE CHR$(HOME)
               PRINT "HOME key"
           CASE CHR$(ENDKEY)
               PRINT "END key"
           CASE CHR$(RIGHT)
               PRINT "RIGHT arrow key"
           CASE CHR$(LEFT)
               PRINT "LEFT arrow key"
           CASE ELSE
               BEEP
       END SELECT
END IF
```

Now test your work, then save it:

QCARDS Code Entry 20

1. Choose the Run menu's Start command (press ALT+R, then press S).

   As you press keys, their names are printed on the QCARDS output screen.

2. Return to the QuickBASIC environment (press ESC).

   The `ESC` key case in the code you copied terminates the program.

3. Save the program as `CHAPTER8.BAS`.

This example code works into QCARDS easily because handling keyboard input is a common task. You can copy, then adapt, this type of code for your programs.

Editing the Block Copied from On-Line Help

Examining and analyzing code from other programs is a great way to extend your programming skills. Modifying code from on-line help to work in your programs can also save you time. Each case in the `SELECT CASE...END SELECT` blocks deals with a key code representing a command the user can choose while using QCARDS. The following steps show you how to modify this block:

QCARDS Code Entry 21

1. Select the whole first block (below the `' Handle ASCII keys` comment, about line 225). It looks like this:

```
SELECT CASE ASC(Choice$)
    CASE ESC
        PRINT "Escape key"
        END
    CASE IS < 32, 127
        PRINT "Control code"
    CASE 30 TO 29
        PRINT "Digit: "; Choice$
    CASE 65 TO 90
        PRINT "Uppercase letter: "; Choice$
    CASE 97 TO 122
        PRINT "Lowercase letter: "; Choice$
    CASE ELSE
        PRINT  "Punctuation "; Choice$
END SELECT
```

This block displays the category of ASCII key the user presses. QCARDS already has a procedure named `AsciiKey` that examines the code of the key the user presses, then calls other procedures to carry out the user's commands.

When you type in the call to `AsciiKey` (in the next step), what you type replaces this selected block.

2. Type the following line at the cursor:

```
call AsciiKey(Choice$, TopCard, LastCard)
```

The `AsciiKey` procedure is called with three arguments, which convey information `AsciiKey` needs to carry out its tasks. `Choice$` contains the code for the key the user presses, `TopCard` contains the number of the card at the front of the screen, and `LastCard` holds the number representing the last record in the `Index()` array.

3. Indent the line you just entered two tab stops.

The edited code should look like this (see Figure 8.3):

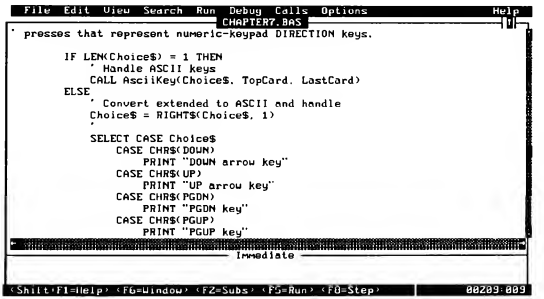


Figure 8.3 Code Showing Call to AsciiKey Procedure

4. Choose the Run menu's Start command (press ALT+R, then press S).

Try issuing some of the QCARDS commands. You should be able to edit the card at the front of the screen. However, if you press the DIRECTION keys, you still just get a description of the key overprinted on the QCARDS screen.

To begin to make QCARDS' DIRECTION keys functional, you'll need to edit more of the text you copied from on-line help, as follows:

QCARDS Code Entry 22

1. Choose the QCARDS Quit command and return to QuickBASIC (press Q).
2. Place the cursor in the first column of the line `PRINT "DOWN arrow key"`
3. Select the whole line (press SHIFT+END), then replace it by typing this line:

```
TopCard = TopCard - 1
```

   This line is executed when the user presses the DOWN key. It decreases the value of the card representing the data at the front of the screen by one. This is the user's way of telling QCARDS to show the record whose number is one less than the current top card.

4. Indent the line, then edit the rest of the cases so the final block looks exactly as follows:

```
SELECT CASE Choice$
           CASE CHR$(DOWN)
                TopCard = TopCard - 1
           CASE CHR$(UP)
                TopCard = TopCard + 1
           CASE CHR$(PGDN)
                TopCard = TopCard - CARDSPERSCREEN
           CASE CHR$(PGUP)
                TopCard = TopCard + CARDSPERSCREEN
           CASE CHR$(HOME)
                TopCard = LastCard
           CASE CHR$(ENDKEY)
                TopCard = 1
           CASE ELSE
                BEEP
       END SELECT
```

   The final two DIRECTION key cases (`RIGHT` and `LEFT`) of the block you copied from on-line help are not needed in QCARDS (since the cards cannot move sideways), so you can delete those cases.

   The `CASE ELSE` part of the statement is executed if the QCARDS user presses a key with an extended character code for which no other case is provided.

5. Save the program as `CHAPTER8.BAS`.

You can replace the BEEP statement used in the CASE ELSE case of the Help example with a call to the Alarm procedure you created in Chapter 5, "The QCARDS Program." The following steps use the Find command to place an earlier call to the Alarm procedure on the Clipboard, then select this BEEP statement and replace it with CALL Alarm.

Optional QCARDS Code Entry 4

1. Choose the Search menu's Find command (press ALT+S, then press F), type CALL Alarm in the text box, and press ENTER.
2. QuickBASIC searches for and then selects CALL Alarm, which was entered in the program in Chapter 5, "The QCARDS Program."
3. Choose the Edit menu's Cut command (press ALT+E, then press T). (The words CALL Alarm are selected, so you don't need to highlight them.)
4. Choose the Find command again, type BEEP in the text box, and press ENTER.

   The match is made with BEEP in the CASE ELSE statement.
5. Choose the Edit menu's Paste command to replace BEEP with CALL Alarm (press ALT+E, then press P).

Finishing the QCARDS Code

The **SELECT CASE** block changes the value of TopCard when the user presses DIRECTION keys, but doesn't do anything to actually change the information displayed on the cards.

The following code lines take care of three eventualities that can arise from certain combinations of DIRECTION key presses, then shows the correct data on the cards:

QCARDS Code Entry 23

1. Place the cursor on the line beneath the following comment (about line 252):

```
' Adjust the cards according to the key pressed by the user,
' then call the ShowCards procedure to show adjusted stack.
```

2. Create an empty line (press ENTER), then type the following code, exactly as shown:

```
if TopCard < 1 then TopCard = LastCard + TopCard
if TopCard > LastCard then TopCard = TopCard - LastCard
if TopCard <= 0 then TopCard = 1
```

   These three lines use the single line form of **IF...THEN** to "rotate" the card stack. They prevent `TopCard` from being passed to `ShowCards` when it has a value that is beyond the range of the `Index()` array.

3. Now type your final code line:

```
CALL ShowCards(TopCard, LastCard)
```

   The `ShowCards` procedure places the proper data on the front card, and the proper name on each of the cards behind it.

4. Save the program now as `CHAPTER8.BAS`.

5. Choose the Run menu's Start command to run QCARDS (press ALT+R, then press S).

Using QCARDS

The commands on the command line at the bottom of the QCARDS output screen are now fully functional. Take a few minutes to try each feature. The commands are summarized in the following list:

QCARDS Command	Result
Edit Top	Lets you change any or all fields of the card at the front of the screen.
Add New	Places a "blank card" on the front of the screen for you to fill in.
Copy to New	Duplicates all fields of the current top card on a new card.
Delete	Marks the current top card for deletion. Note that it is not actually deleted until you choose Quit. Therefore, if you decide not to delete the card, you can use CTRL+BREAK to break out of the program. When you restart, the card will still be there.

Find	Lets you move to one of the fields, then type what you want to search for. If an exact match is found, the card with the exact match is placed in the first position. If no exact match is found, QCARDS beeps.
Sort	Orders the cards alphabetically (or numerically) by the field in which you place the cursor. For example, if the cursor is in the zip code (Zip) field, Sort orders the cards in increasing zip code order. If you place the cursor in the state (ST) field, the cards are ordered alphabetically by state.
Print	Prints out just the name and address fields (for use on a mailing label).
Quit	Writes the current Index array to the random-access disk file. If any cards have been marked as deleted, they are removed from the file at this time. Edits to cards and additions to the file are also made when you quit QCARDS. Quit then terminates the QCARDS program.

Try the DIRECTION keys too. Most of the DIRECTION keys work. However, by copying code from Help, you inadvertently incorporated a naming inconsistency that caused a bug in your code. In the next chapter, you'll use QuickBASIC's debugging features to remove the bug.

For More Information

For more information on on-line help, see the following:

Chapter	Topic
Section 10.8, "Using On-Line Help" and Chapter 21, "The Help Menu"	Information on QuickBASIC's Help menu commands and complete descriptions of all the Help features.

CHAPTER 9

Debugging While You Program

Microsoft QuickBASIC's smart editor keeps your code free of syntax errors as you enter individual statements. But QuickBASIC has other features that help you isolate "bugs"—errors in the general logic or specific details of your program that cause it to behave in unanticipated ways. Some of these features are on the Debug menu, while others are on the Run menu. Still others are controlled completely by function keys. These features let you suspend program execution at any point, control which statements of your program execute, and monitor the values of variables.

In this chapter, you'll find a real bug and fix it. You'll practice

- Using QuickBASIC's debugging features
- Defining and calling a procedure that has parameters
- Creating a stand-alone executable file from within QuickBASIC

This chapter takes about one to one and one-half hours to complete.

Debugging Commands

Most QuickBASIC debugging commands are on the Debug menu. However, some of the most useful debugging features are on the Run menu, or are accessible only through function keys.

Debug-Menu Commands

Commands on the Debug menu let you control program flow and monitor the value of variables in a running program.

Command (and Shortcut Key)	Action
Add Watch	Places a variable you specify into the Watch window, then continually updates its value.
Instant Watch (SHIFT+F9)	Gives current value of a highlighted variable; optionally, places the variable in Watch window.
Delete Watch	Deletes the specified variable from Watch window.
Toggle Breakpoint (F9)	Sets a "breakpoint" at the cursor if one is not currently set there. A breakpoint is a specified location where program execution will halt. If there is a breakpoint currently set at the cursor, choosing Toggle Breakpoint turns it off.
Clear All Breakpoints	Turns off all breakpoints everywhere in the program.

Debugging Commands on the Run Menu

Commands from the Run menu control program execution during debugging.

Command (and Shortcut Key)	Action
Start (SHIFT+F5)	Runs the program in the View window beginning with the first statement.
Restart	Resets all variables to zero (or zero-length strings), compiles declarations, and sets the first executable statement as the current statement.
Continue (F5)	Continues execution of a suspended program from the current statement. In a program that is not running, the effect is identical to the Start command.

Function Keys Used in Debugging

The following table shows the debugging commands that are controlled only by function keys:

Command	Action
F7	Executes program to statement at the cursor.
F8	Single steps through your program. Each time you press F8, the execution sequence progresses one statement. If the current statement is a procedure invocation, F8 moves the procedure definition into the View window. If the program is not already running, F8 acts the same as the Run menu's Restart command: pressing F8 restarts the program, making the first executable statement the current statement. In a suspended program, F8 executes the current statement in the execution sequence.
F10	Single steps through your program. F10 steps over or executes all statements of a procedure as though they were a single statement without moving the procedure definition into the View window. Otherwise, F10 acts the same as F8.

Debugging a Procedure

In this section you'll use everything you've learned so far to turn the `SELECT CASE` block that handles DIRECTION keys into a procedure. In Chapter 5, "The QCARDS Program," you saw how easy it is to define procedures with QuickBASIC. The procedure you create in the next code-entry sequence is defined with parameters. When it is called, it is called with arguments.

The following steps tell you how to create a **SUB** procedure with the name `DirectionKey`. As noted in Chapter 5, when QuickBASIC's smart editor

processes the word `sub` followed by a procedure name, it opens a window and starts the procedure with the **SUB** and **END SUB** statements.

QCARDS Code Entry 24

1. Start QuickBASIC if it is not already running, and open `CHAPTER8.BAS`.
2. Place the cursor in the leftmost column of the line containing `SELECT CASE Choice$` (about line 231), then select the whole block (a total of 16 lines) down to and including `END SELECT`. It looks like this (see Figure 9.1):

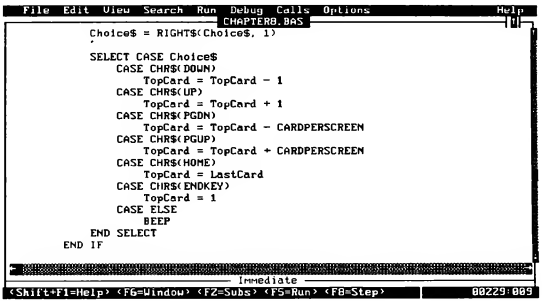


Figure 9.1 Selecting the SELECT CASE Choice$ Block

3. Choose the Edit menu's Cut command to delete the text and place it on the Clipboard (press ALT+E, then press T).
4. Create a blank line above the `END IF` statement, then move the cursor to column 1 of the new blank line.
5. Type

   ```
   sub DirectionKey(Choice$, TopCard%, LastCard%)
   ```

   then press ENTER.

   Notice that QuickBASIC has started a **SUB...END SUB** procedure with the name you specified, and has moved it into the View window.

6. Put the cursor on the blank line immediately above the `END SUB` statement (press DOWN).

7. Choose the Edit menu's Paste command to paste the text block from the Clipboard. The procedure should look like Figure 9.2.

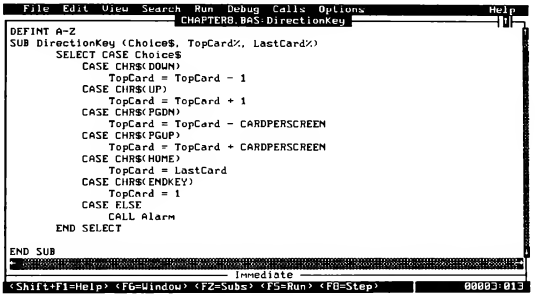


Figure 9.2 DirectionKey Procedure in View Window

Now that the procedure is defined, you have to put a **CALL** statement at the place where you deleted the text. The **CALL** statement makes the statements in a **SUB** procedure part of the program's execution sequence. Do the following:

QCARDS Code Entry 25

1. Choose the View menu's SUBs command (press F2).

   Note that the fifth entry under `CHAPTER8.BAS` is now `DirectionKey`.

2. Select `CHAPTER8.BAS` if it isn't already selected. Press ENTER to return the cursor to the module-level code.

3. Move the cursor to the end of the following line (at about line 229):

   ```
   Choice$ = RIGHT$(Choice$, 1)
   ```

4. Create a blank line (press ENTER). Then, type the following line at the cursor but *do not* press ENTER:

   ```
   call DirectionKey(Choice$, TopCard, LastCard)
   ```

The `DirectionKey` procedure is more complicated than `Alarm` because it is defined with parameters (the words in parentheses in the `SUB` statement you created earlier) and called with arguments (the words in parentheses in the `CALL` statement). You don't have to use the same names for the parameters as you use for the arguments, but it is all right to do so.

The `DirectionKey` procedure makes QCARDS' module-level code easier to follow because the arithmetic for handling DIRECTION keys is now represented by a single statement, rather than 15 lines of statements. Using procedures costs almost nothing in terms of execution speed or program size, so the added program readability is worth the small effort of programming them. Procedures are especially beneficial for tasks that are performed repeatedly in a program.

Learning about Procedures

You can use QuickBASIC's debugging features to learn about procedures. The cursor should still be on the call you just typed to `Direction Key` (about line 230). Try the following:

1. Execute the program up to the call to `DirectionKey` (press F7).

   QCARDS starts to run. In the following step you will press a DIRECTION key while QCARDS is running, causing QuickBASIC to suspend execution at the call to `DirectionKey`.

2. Press a DIRECTION key (preferably PGUP).

   This returns control to the QuickBASIC environment. The statement `CALL DirectionKey` appears in high-intensity video. It hasn't been executed yet—it is the current statement.

3. Execute the `CALL DirectionKey` procedure as a single statement (press F10). The `END IF` statement appears in high-intensity video. It is now the current statement.

Continuing a Suspended Program

You don't always have to start suspended programs over from the beginning after making edits. Usually you can just choose the Continue command from the Run menu to continue execution from the point of suspension. QCARDS should run as before. When you press DIRECTION keys, the data on the cards should change. Try the following:

1. Continue program execution (press F5).
2. Press the END key on the numeric keypad to move the data on the cards to the last item. The END key doesn't work. If you did the optional code entry in Chapter 8, the `Alarm` procedure sounds off because the END key isn't recognized as one of the cases in the `SELECT CASE` block. (If you didn't do the optional code entry in Chapter 8, QCARDS beeps.) You have a bug.
3. Press the UP, DOWN, PGDN, and PGUP keys. They should all work. So there is a bug in the code dealing with the END key.
4. Return to QuickBASIC (press CTRL+BREAK).

   QuickBASIC returns you to the environment and places the cursor at either the `Choice$ = INKEY$` statement, or `LOOP WHILE Choice$ =` statement, whichever was executing when you pressed CTRL+BREAK. Right now you are stuck in this `DO` loop. You cannot step past it with F8 or F10. You will have to place the cursor on the statement following the loop, then use F7 to execute to it.
5. Move the cursor down (about 27 lines) past the big comment block to the following line (at about line 224):

   ```
   IF LEN(Choice$) = 1 THEN
   ```
6. Execute the program to the statement at the cursor (press F7). The QCARDS screen is displayed.
7. Press the END key again.

   QuickBASIC returns you to the environment. The line you were on when you pressed F7 is highlighted:

   ```
   IF LEN(Choice$) = 1 THEN
   ```

Usually when you suspend program execution you can press F5 to continue execution from the point of suspension. However, some edits you make to a suspended program prevent simple continuation. In the next few QCARDS code-entry sequences, QuickBASIC may display a dialog box indicating that one of the edits you have made requires restarting the program (see Figure 9.3). That is normal. Just press ENTER and continue editing.

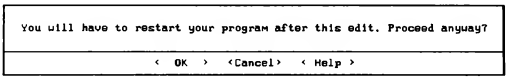


Figure 9.3 Restart Program Error Message

Isolating a Bug

QuickBASIC's debugging features make coding and logic errors easy to trace and fix because they make it easy to understand the line-by-line workings of your code. In the next QCARDS code-entry sequence, you'll use single stepping to find the bug. You know you pressed a DIRECTION key to suspend the program so you can anticipate that execution will move to the `ELSE` block when you execute the statement at the cursor.

Follow these steps to track down the bug:

QCARDS Code Entry 26

1. Execute the `IF` statement and the one that follows and move the cursor to the call to `DirectionKey` (press F8 twice to single step up to the call).
2. Single step into the `DirectionKey` procedure (press F8).

   The cursor is placed on the `SELECT CASE` statement.

3. Set a breakpoint (see Figure 9.4) at the current line (press F9).

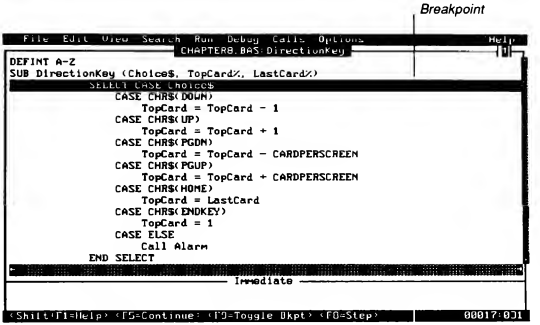


Figure 9.4 Setting a Breakpoint

4. Continue execution of QCARDS (press F5 twice). When the QCARDS output screen appears, press the END key to activate the suspected bug.

   QuickBASIC returns the cursor to the line on which you set the breakpoint.

5. Single step to the `CASE ELSE` statement (press F8). The `CALL Alarm` statement appears in reverse video.

6. Move the cursor into the word HOME in the CASE CHR$(HOME) statement, then get help (press F1).

   The Help window opens and describes HOME as a symbolic constant with a value of 71 (see Figure 9.5).

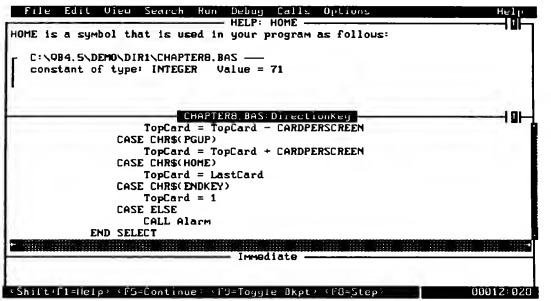


Figure 9.5 Symbol Help for Symbolic Constant HOME

7. Move the cursor down to the CASE CHR$(ENDKEY) statement, place the cursor in the word ENDKEY, and get help (press F1).

   The Help window opens and describes ENDKEY as a variable of type INTEGER. But ENDKEY should be a symbolic constant, just like HOME.

8. Choose the Instant Watch command from the Debug menu.

   The value of ENDKEY is shown as zero.

9. Press ENTER to place ENDKEY in the Watch window.

In the program, ENDKEY should be a symbolic constant defined with a **CONST** statement in the declarations and definitions section of the program (just like HOME). Try this:

QCARDS Code Entry 27

1. Choose the View menu's SUBs command (press ALT+V, then press S), then choose the first entry (CHAPTER8.BAS) in the list box to move the module-level code of the program into the View window.
2. Move the cursor to the top of the module-level code (press CTRL+HOME).
3. Choose the Search menu's Find command, then type HOME in the text box and press ENTER.

HOME is selected in the declarations and definitions section. Beside it, you can see the cause of the bug. The symbolic constant used to represent the END key is ENDK, whereas the **SELECT CASE** code you copied from on-line help used ENDKEY to represent the END key (see Figure 9.6).

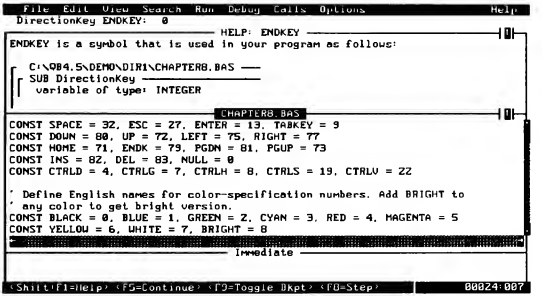


Figure 9.6 Error in Naming Symbolic Constant for the END Key

Most of the symbolic-constant names used to represent the DIRECTION keys are the same as those shown on the keys themselves, but you can't represent the END key with the symbolic constant END, because **END** is a BASIC keyword. The on-line help example used ENDKEY to deal with this problem, QCARDS uses ENDK. It doesn't matter which is used, ENDKEY is just as good as ENDK. What

is important is that the word used in the definition be the same as that used elsewhere in the program. Do the following to change ENDKEY to ENDK:

QCARDS Code Entry 28

1. Choose the Search menu's Change command (press ALT+S, then press C). (If you get a message saying you will have to restart your program, press ENTER.)
2. Type ENDKEY in the first text box, then press TAB and type ENDK in the second text box.
3. Start the search (press ENTER). The first match is in the DirectionKey procedure.
4. Make the change (press C). The search ends. That was the only occurrence of ENDKEY.

Closing the Watch Window

When the Watch window is open, QuickBASIC executes more slowly than when it is closed. You close the Watch window by repeatedly choosing the Delete Watch command from the Debug menu, then choosing variables from the resulting dialog box until the Watch window closes. You may also want to occasionally choose the Clear All Breakpoints command too.

1. Choose the Debug menu's Clear All Breakpoints command (press ALT+D, then press C).
2. Choose the Debug menu's Delete Watch command (press ALT+D, then press D).
3. Delete the entry (press ENTER).

The next step is to run your program and save it. Do the following:

QCARDS Code Entry 29

1. Choose the Run menu's Start command. QCARDS should run and recognize all the DIRECTION keys as well as all the command-line keys.
2. Choose Quit from the QCARDS command line, then press a key to continue.
3. Choose the File menu's Save As command and save the program as `FINALQ.BAS`.

   QCARDS should now be fully functional.

Automatic Procedure Declarations

When you save a program, QuickBASIC creates a **DECLARE** statement for any new procedure. Generally, you should move such **DECLARE** statements to the declarations and definitions section of your program. QuickBASIC uses procedure declarations to check the number, order, and types of the arguments you pass to a procedure when you call it. **DECLARE** statements should be toward the end of the declarations and definitions below statements such as **DEF***type* and **TYPE...END TYPE**. You should copy the `DECLARE` statement for the procedure `DirectionKey` into the procedure-declarations portion of the declarations and definitions (as you did with the declaration for `Alarm` in Chapter 5). Do the following:

QCARDS Code Entry 30

1. Move to the beginning of the program (press CTRL+HOME).
2. Select the entire `DECLARE DirectionKey` statement (press SHIFT+END).
3. Delete the selected statement from QCARDS and place it on the Clipboard (press ALT+E, then press T).
4. Move to the **SUB** procedure declarations section of the program (about line 74).

5. Create a blank line (if necessary) and insert the `DECLARE DirectionKey` statement (press ALT+E, then press P).

6. Save the program again as `FINALQ.BAS`.

Creating a Stand-Alone Executable File

You can create a version of QCARDS that you can execute directly from the DOS command line without starting QuickBASIC at all. Follow these steps:

1. Choose the Run menu's Make EXE File command (press ALT+R, then press X).

   If a dialog box appears asking if you want to save your files, press ENTER. The Make EXE File dialog box appears (see Figure 9.7):

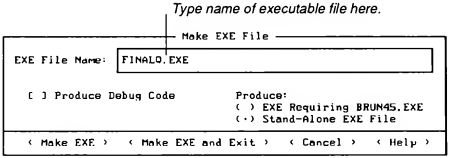


Figure 9.7 Make EXE File Dialog Box

2. Type `QCARDS` in the text box.

3. Choose the Make EXE and Exit option (press ALT+E).

   QuickBASIC creates the EXE file and exits to DOS.

4. Run QCARDS (type `qcards` at the DOS prompt and press ENTER).

NOTE *If you don't have a hard disk, QuickBASIC prompts you to insert a disk containing BC.EXE, and later prompts you for a path for libraries it needs to finish making the executable file. BC.EXE, LINK.EXE and BRUN45.LIB are on disk 3 (Utilities 1). BCOM45.LIB is located on disk 4 (Utilities 2). When asked for a path, type in a full path including both drive name and file name.*

Learning about QuickBASIC's Other Menu Items

The preceding chapters have familiarized you with most of the commands on QuickBASIC's Easy Menus. Two commands were not covered. The Options menu's Set Paths command lets you change some settings that were made automatically by the Setup program you used to install QuickBASIC. The View menu's Included Lines command is used only in programs that rely on include files. See Sections 20.2, "Set Paths Command," and 14.7, "Included Lines Command," for full discussions of these commands.

Easy Menus provides a very functional programming environment for the novice or intermediate BASIC user. QuickBASIC's Full Menus provide additional commands needed for advanced or professional programming. For example, the Full Menus Debug menu contains commands useful for debugging large and complex programs. The Full Menus File menu lets you load multiple files simultaneously, so you can create programs using multiple, separately compiled source files (modules). The uses of these advanced commands are covered in Chapters 10–21 of this manual. The programming techniques are discussed in *Programming in BASIC*.

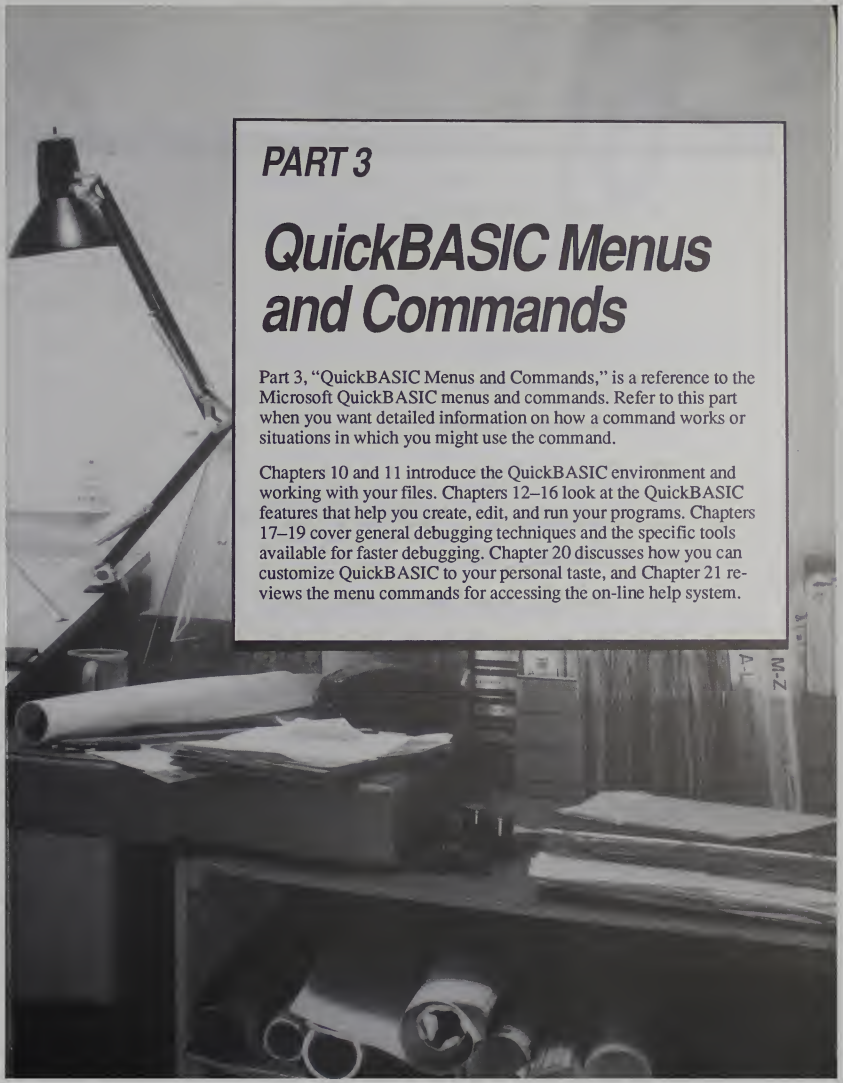
For More Information

For more information on the topics discussed in these chapters, see the following:

Chapter	Topic
Chapter 2, "SUB and FUNCTION Procedures," in *Programming in BASIC*	Complete information on programming with procedures, including rules for parameters and arguments.
Chapter 17, "Debugging Concepts and Procedures," Chapter 18, "The Debug Menu," and Chapter 19, "The Calls Menu"	Complete rules for using all QuickBASIC's debugging features, including advanced features such as watchpoints and the Calls menu.

PART 3

QuickBASIC Menus and Commands



PART 3

QuickBASIC Menus and Commands

Part 3, "QuickBASIC Menus and Commands," is a reference to the Microsoft QuickBASIC menus and commands. Refer to this part when you want detailed information on how a command works or situations in which you might use the command.

Chapters 10 and 11 introduce the QuickBASIC environment and working with your files. Chapters 12–16 look at the QuickBASIC features that help you create, edit, and run your programs. Chapters 17–19 cover general debugging techniques and the specific tools available for faster debugging. Chapter 20 discusses how you can customize QuickBASIC to your personal taste, and Chapter 21 reviews the menu commands for accessing the on-line help system.

10.1 Starting QuickBASIC

This section discusses starting QuickBASIC with and without special options. It also describes the parts of the QuickBASIC screen. If you have not already done so, read Chapter 1, "Setting Up QuickBASIC," to set up QuickBASIC on your computer.

10.1.1 The QB Command

To start QuickBASIC, type `QB` at the DOS prompt.

The full syntax for the `QB` command is the following:

QB [[/RUN]] [[*programname*]] [[/B]] [[/G]] [[/H]] [[/NOHI]] [[/C:*buffersize*]] [[/L[[*libraryname*]]] [[MBF]] [[/AH]] [[/CMD *string*]]]

NOTE *In this manual, the QB command and its options appear in uppercase letters. However, because DOS is not case sensitive, you may use lowercase letters as well.*

The following list describes QuickBASIC's command options. These options can be typed on the DOS command line following the `QB` command and have the effects described. Use the options in the order listed.

Option	Description
/RUN *programname*	Causes QuickBASIC to load and run *programname* before displaying it.
programname	Names the file to be loaded when QuickBASIC starts.
/B	Allows the use of a composite (black-and-white) monitor with a color graphics card. The /B option displays QuickBASIC in black and white if you have a color monitor.

/G	Sets QuickBASIC to update a CGA screen as fast as possible. The option works only with machines using CGA monitors. If you see "snow" (dots flickering on the screen) when QuickBASIC updates your screen, then your hardware cannot fully support the /G option. If you prefer a "clean" screen, restart QuickBASIC without the option.
/H	Displays the highest resolution possible on your hardware.
/NOHI	Allows the use of a monitor that does not support high intensity. See Section 10.1.2 for a description of this option.
/C:*buffersize*	Sets the size of the buffer receiving data. This option works only with an asynchronous communications card. The default buffer size is 512 bytes; the maximum size is 32,767 bytes.
/L *libraryname*	Loads the Quick library that is specified by *libraryname*. If *libraryname* is not specified, the default Quick library, QB.QLB, is loaded.
/MBF	Causes the QuickBASIC conversion functions to treat IEEE-format numbers as Microsoft Binary format numbers.
/AH	Allows dynamic arrays of records, fixed-length strings, and numeric data to be larger than 64K each.
/CMD *string*	Passes *string* to the COMMAND$ function. This option must be the last option on the line.

For example, type

```
QB /RUN GRAPHIX /G /AH
```

if you want to run a program named `GRAPHIX` with the following options:

- Load and run `Graphix` before displaying it
- Quickly update CGA screen
- Allow dynamic arrays to exceed 64K

Generally, however, you start QuickBASIC by typing `QB` and pressing ENTER.

10.1.2 The /NOHI Option

If you type `QB` at the DOS level, QuickBASIC assumes you have a monitor that can display high intensity. However, if you use a monitor that does not support high intensity, you will need to tell QuickBASIC how to display on your system. Use the list below to determine any options you may need.

Monitor Display	Invocation Command
16 colors (CGA, EGA or VGA)	`QB`
4-color monochrome (MDA)	`QB`
8 colors (CGA, EGA or VGA)	`QB /NOHI`
4-color black-and-white composite (CGA, EGA or VGA)	`QB /B`
2-color black-and-white composite (CGA, EGA or VGA)	`QB /B /NOHI`

Laptop computers frequently use liquid crystal displays (LCDs) that are considered 2-color black-and-white composite displays; they require the `QB /B /NOHI` command.

10.1.3 The QuickBASIC Screen

The first time you type `QB` and press ENTER from DOS, a dialog box offers you the opportunity to review the QuickBASIC Survival Guide. If you press ESC to clear the dialog box, you now see the QuickBASIC screen, shown in Figures 10.1 and 10.2. Figure 10.1 shows the top half of the QuickBASIC invocation screen.

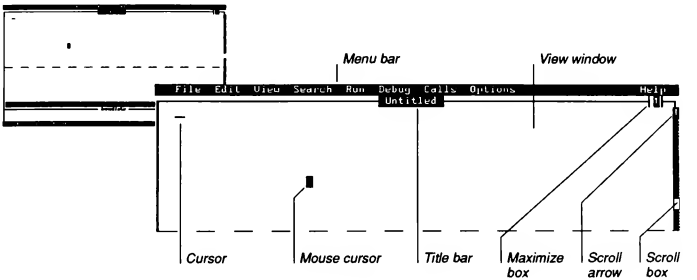


Figure 10.1 Top Half of QuickBASIC Invocation Screen

The following list describes the items shown in Figure 10.1.

Item	Description
Menu bar	Names each menu. When you press the ALT key, the highlighted letter indicates which key "pulls down" that menu.
View window	Displays your program's text.
Cursor	Shows where the text you type will appear. The cursor appears in the active window.
Mouse cursor	Shows current on-screen position of mouse (use with mouse only).
Title bar	Shows the name of the program or procedure.
Maximize box	Expands the active window to fill the screen (use with mouse only).
Scroll arrow	Scrolls the text one character or one line at a time (use with mouse only).
Scroll box	Shows cursor's relative position within the file or procedure.

Figure 10.2 shows the bottom half of the QuickBASIC invocation screen.

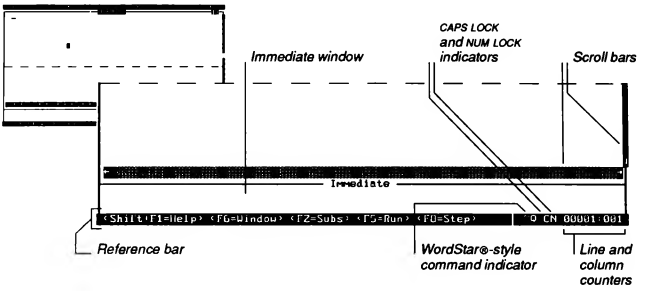


Figure 10.2 Bottom Half of QuickBASIC Invocation Screen

Item	Description
Immediate window	Executes QuickBASIC statements directly; they need not be entered as part of a program.
CAPS LOCK and NUM LOCK indicators	C appears when the CAPS LOCK key is toggled on. N appears when the NUM LOCK key is toggled on.
Scroll bars	Scrolls text in the currently active window (use with mouse only).
Reference bar	Contains reference information. The first five items are buttons showing a frequently used shortcut key and the corresponding menu or option. Clicking a button with the mouse gives the same result as pressing the shortcut key shown in the button. Pressing ALT displays four different reference items.
WordStar-style command indicator	^Q appears when you enter CTRL+Q, a WordStar-style command. ^K appears here when you are setting a place marker (see Section 12.4, "Using Placemarkers in Text"). ^P appears here when you are entering a literal character (see Section 12.6, "Entering Special Characters").
Line and column counters	Give current position of cursor within the text in the active window.

When you start QuickBASIC without specifying a program name, you can begin to write a program, or you can ask for on-line help. To use the QuickBASIC on-line help, press F1 for a general Help window, or use the Help menu for more detailed information (see Section 10.8, "Using On-Line Help," and Chapter 21, "The Help Menu," for more details).

To clear a Help window, press the ESC key. Pressing ESC also clears menus, dialog boxes, and error messages from the QuickBASIC screen.

10.2 Opening Menus and Choosing Commands

QuickBASIC commands are organized in pull-down menus on the menu bar. Figure 10.3 shows one of these, the File menu.

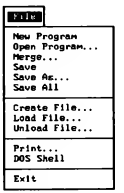


Figure 10.3 The File Menu

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

- Open menus and choose commands with the keyboard or a mouse. (See Section 10.7, "Using the Mouse," for a summary of mouse techniques.)
- Execute a command directly with a "shortcut" key, which is one or two keystrokes that perform the same task as a menu command.

Menu commands followed by three dots (. . .) indicate that more information is required before the command can execute. In these cases, a "dialog box"—a box that asks you for additional information—appears on the screen.

10.2.1 Keyboard Technique

You can choose any command on a QuickBASIC menu by using the keyboard.

To open a menu, follow these steps:

1. Press and release the ALT key. (Notice that after you press the ALT key, the first letter of each menu is highlighted.)
2. Press the first letter of the menu's name.

NOTE *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 active only within menus. When using key combinations in other circumstances, you must hold down the first key while pressing the rest of the sequence.*

This "selects" (highlights) the entire menu and displays the menu's commands. A command name contains a highlighted letter whenever it is "active" (available).

If no letter in a command is highlighted and it appears dimmed, you cannot choose that command until you take some other action. For example, if you are editing a program but have not selected any text to copy or delete, the commands Cut and Copy can have no effect. You can use Cut and Copy only after you select text. (See Section 12.2 to learn how to select text for copying or deleting.)

To move from one menu to another, do either of the following:

- Close the menu by pressing ESC, then repeat steps 1 and 2 above.
- Press the LEFT or RIGHT direction keys.

To choose a command, do one of the following:

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

Some commands take effect immediately and cannot be canceled once chosen. However, commands followed by three dots (. . .) cause QuickBASIC to display a dialog box so you can supply additional information. To cancel a dialog box, press ESC. See Section 10.3 for a discussion of dialog boxes.

See Also

Section 10.7, "Using the Mouse"

10.2.2 Using Shortcut Keys

In QuickBASIC the function keys (F1-F10) serve as shortcut keys for many of the menu functions. Shortcut keys allow you to substitute one or two keystrokes for the process of choosing a command through on-screen menus. Shortcut keys for menu commands are listed on the menu, next to the command. If you are new to programming you may prefer to use only the on-screen menus. When you feel comfortable with QuickBASIC commands, you may want to use the shortcut keys.

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. Table 10.1 lists and explains all of the QuickBASIC shortcut keys and equivalent menu commands.

Table 10.1 QuickBASIC Shortcut Keys

Programming Shortcut Keys	Action	Menu Command
SHIFT+F1	Displays general help information	Help on Help command on Help menu
F1	Displays help information on the keyword or variable nearest to the left of the cursor	Topic command on Help menu
ALT+F1	Displays up to 20 previous help screens	None
F2	Displays a list of loaded **SUB** or **FUNCTION** procedures, modules, include files, or document files	SUBs command on View menu
SHIFT+F2	Displays next procedure in active window	Next SUB command on View menu
CTRL+F2	Displays previous procedure in active window	None
F3	Finds next occurrence of previously specified text	Repeat Last Find command on Search menu
F4	Toggles display of output screen	Output Screen command on View menu
F5	Continues program execution from current statement	Continue command on Run menu

Table 10.1 (*continued*)

Programming Shortcut Keys	Action	Menu Command
SHIFT+F5	Starts program execution from beginning	Start command on Run menu
F6	Makes next window the active window	None
SHIFT+F6	Makes previous window the active window	None
F7	Executes program to current cursor position	None
F8	Executes next program statement, traces through procedure	None
SHIFT+F8	Steps back in your program through the last 20 program statements recorded by the History On or Trace On command	None
F9	Toggles breakpoint	Toggle Breakpoint command on Debug menu
SHIFT+F9	Displays Instant Watch dialog box	Instant Watch command on Debug menu
F10	Executes next program statement, traces around procedure	None
SHIFT+F10	Steps forward in your program through the last 20 program statements recorded by the History On or Trace On command	None
CTRL+F10	Toggles between multiple windows and full screen for active window	None

See Also

Section 12.7, "Summary of Editing Commands"

10.3 Using Dialog Boxes

QuickBASIC displays a dialog box when it needs additional information before it can carry out an action. For example, a dialog box might do the following:

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

Figures 10.4 and 10.5 illustrate the parts of a dialog box.

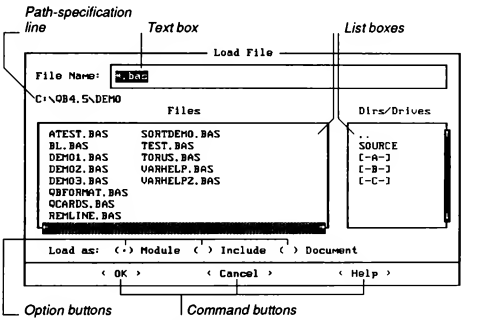


Figure 10.4 Load File Dialog Box

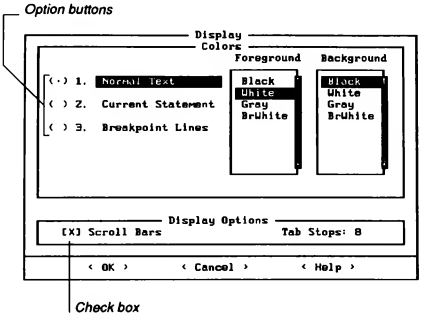


Figure 10.5 Display Dialog Box

To move between different parts of a dialog box, use one of the following methods:

- Press the TAB key.
- Hold down the ALT key while pressing the key corresponding to the highlighted letter in the item you want selected. When a dialog box opens, it usually contains any options set previously.

The different dialog box components are described in the following list:

Component	Description
Path-specification line	Displays the path to the current directory. Change the path by either typing a new path name in the text box or by selecting the appropriate directory in the Dirs/Drives list box.
Text box	Displays typed text.

List box	Lists a group of similar items, such as files or procedures available, from which you may choose one. Use the DIRECTION keys to move within list boxes.
Check box	Toggles an option on or off. Once the cursor is in the checkbox area, press SPACEBAR, or press ALT and then the highlighted letter of the item you want to turn on or off. When an option is on, an X appears in the check box; when it is off, the check box is empty.
Option button	Selects one of a group of options. Only one of the group may be chosen at a time. Use the DIRECTION keys to move among option buttons.
Command button	Executes a command. Once the cursor is in the command button area, use TAB to alter selections. Press either SPACEBAR or ENTER to execute commands.

Note that within dialog boxes you must keep the ALT key pressed while pressing the high-intensity letter. This procedure is slightly different from the one for choosing menu commands, which does not require you to hold the ALT key down while pressing the highlighted letter.

10.4 Using Windows

Microsoft QuickBASIC uses windows to contain different types of information. There are four window types—View, Immediate, Help, and Watch. This section introduces QuickBASIC's windows and tells how to do the following:

- 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

10.4.1 Window Types

QuickBASIC uses different windows to perform different functions. 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 10.1). When you load a program, the code that is outside of any **FUNCTION** or **SUB** procedure—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 at the bottom of the screen when you start QuickBASIC. The Immediate window allows you to execute BASIC statements immediately. (See Section 17.3.5 for more information on the Immediate window.)
Help	The window that contains on-line help. You can copy examples from Help windows and paste them into your program.
Watch	The window that opens at the top of the screen when you choose certain commands from the Debug menu. It displays the values of variables as your program runs. (See Section 17.3.4 for more information on using the Watch window for debugging.)

10.4.2 Splitting the View Window (Full Menus Only)

You can split the View window into upper and lower windows; this allows you to view or edit two parts of a program simultaneously. Because the View window can be split in two (Full Menus only), you can have a total of five windows open at one time (two View windows, the Immediate window, the Help window, and the Watch window), as shown in Figure10.6.

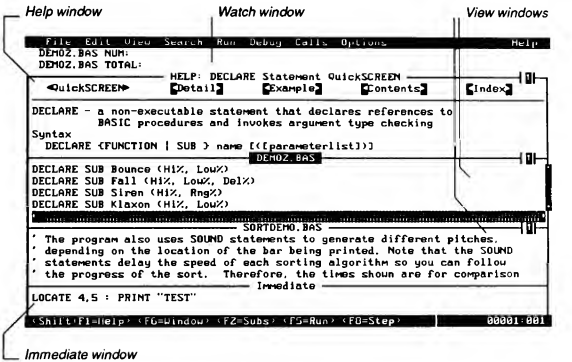


Figure 10.6 QuickBASIC Screen with Five Windows Open

To split the View window, choose the Split command from the View menu. The screen is now divided into two parts. Repeat the process to restore the original screen.

See Also

Section 14.3 "Split Command"

10.4.3 Changing the Active Window

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

To make another window active, follow these steps:

- Press F6 to cycle downward through the windows on the screen.
- Press SHIFT+F6 to cycle upward.

10.4.4 Changing Window Size

You can increase or decrease the size of a window 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 (hold down the first key while pressing the second) to increase or decrease the window's size. (Use the PLUS and MINUS keys on the keypad.)

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 occupies the whole screen, returns the active window 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.

10.4.5 Scrolling in the Active Window

To look at the parts of a file that do not fit within the boundaries of a View window, you can move the text ("scroll") up, down, right, or left.

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

Refer to Table 10.2 for information on scrolling more than one character at a time. The rightmost column in this table shows keystrokes that you may find more convenient if you prefer WordStar-style commands.

Table 10.2 Scrolling

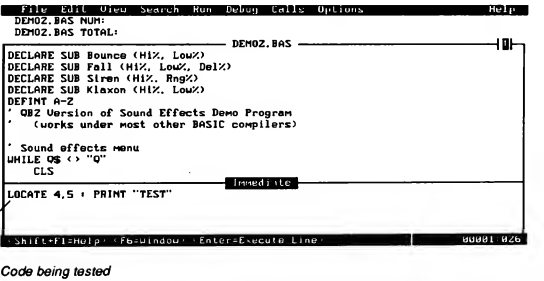
Scrolling Action	Keystrokes	WordStar-Style Equivalents
Beginning of line	HOME	CTRL+Q+S
End of line	END	CTRL+Q+D
Page up	PGUP	CTRL+R
Page down	PGDN	CTRL+C
Left one window	CTRL+PGUP	---
Right one window	CTRL+PGDN	---
Beginning of file	CTRL+HOME	CTRL+Q+R
End of file	CTRL+END	CTRL+Q+C

You can also set place markers anywhere in your program and jump between them while editing. See Chapter 12, "Using the Editor," for more information.

10.5 Using the Immediate Window

The Immediate window, the bottom window on the initial QuickBASIC screen, allows direct execution of QuickBASIC statements. Use it to refine short pieces of program code and see the effect immediately. When you are satisfied with the results, you can copy the code into your program.

Figure 10.7 shows code being tested in the Immediate window.



Code being tested

Figure 10.7 Immediate Window

You can enter up to ten 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 one line, but each complete statement must be separated from the next by a colon (:). When you place the cursor on a line and press ENTER, only the statements on that line execute.

As you write your program, you can make the Immediate window active, type and edit statements as you would in the Edit window, then execute them immediately. By testing your ideas before incorporating them into a larger program, you verify that they function properly.

The Immediate window is similar to "direct mode" in BASICA, and has the following characteristics:

- Any statement or group of statements on a line will be executed when you place the cursor anywhere on the line and press ENTER.
- As many as ten lines can be entered in the Immediate window; you can execute the lines in any order.

  After you have entered ten 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, changes in one line 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
  ```
- The Immediate window is expanded when you repeatedly press 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 appear, only the first ten are available for use. Press CTRL+F10 again to return the window to its previous size.
- Code written in the Immediate window is not saved to disk by the Save command. Use the Cut or Copy commands from the Edit menu to move the contents of the Immediate window into a View window if you want to save them with the rest of the file.

See Also

Section 17.2, "Preventing Bugs with QuickBASIC"; Section 17.3.5, "Immediate Window"

10.5.1 Statements Not Allowed

Most QuickBASIC statements and functions are allowed in the Immediate window. However, an error message will appear if you use the following QuickBASIC keywords 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**	**$DYNAMIC**

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.*

10.5.2 Doing Calculations

Use the Immediate window to calculate complicated expressions, then display results with the **PRINT** statement. Use any of BASIC's intrinsic functions (such as **SIN** or **EXP**) in these calculations. Similarly, you can use the Immediate window to print the value returned by **FRE(–1)**. This tells you the amount of memory available after both QuickBASIC and your program are loaded.

10.5.3 Testing Screen Output

As a program grows in length and complexity, it is useful to test your code before incorporating it into your program. Typing code in the Immediate window and testing it there first avoids having to run a program from the beginning each time you want to modify a small part of code.

Examples

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

```
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
```

10.5.4 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** and **END SUB** or **FUNCTION** and **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
```

10.5.5 Changing the Values of Variables

Use the Immediate window to assign a new value to a variable in a running program. Stop the program, assign the variable a new value in the Immediate window, and continue execution. The value you assign in the Immediate window becomes the variable's value when the program continues running.

Example

The following steps show how you can use the Immediate window to change a variables 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. Use the Start command on the Run menu to start the program, then use CTRL+BREAK to interrupt it.
3. Type

   ```
   i% = 9900
   ```

   in the Immediate window and press ENTER.
4. Press F5 to continue the program.

   The loop now executes from 9900 to 10,000.

The Immediate window can access variable values only after those variables have been assigned values; this occurs after statements containing the variables have been executed in the View window. For example, suppose the statement you just executed in the View window gave a variable called `x%` a value of `3`. You then enter `PRINT x%` in the Immediate window. The number `3` will be 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.

10.5.6 Simulating Run-Time Errors

Run-time errors are those that occur while a program is running. Each run-time error is associated with a numeric code. If your program uses error handling (**ON ERROR** statements), QuickBASIC returns this numeric code to your program.

One way to find out which error message is associated with a given code is to refer to Appendix I, "Error Messages," in *Programming in BASIC*. 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. Figure 10.8 shows an error-message dialog box on the QuickBASIC screen.

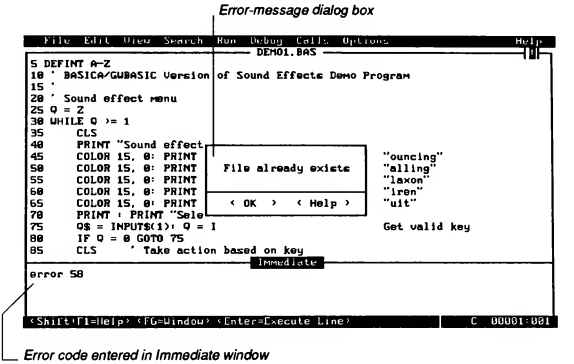


Figure 10.8 Simulated Run-Time Error

10.6 Using the Watch Window

The Watch window is one of QuickBASIC's advanced debugging features. It displays variables tracked for debugging purposes. The Watch window opens at the top of the screen when you set a watchpoint or add a watch expression. (See Section 17.3.2, "Breakpoints and Watchpoints," and Section 17.3.3, "Watch Expressions.")

Figure 10.9 shows the QuickBASIC screen with entries in the Watch window.

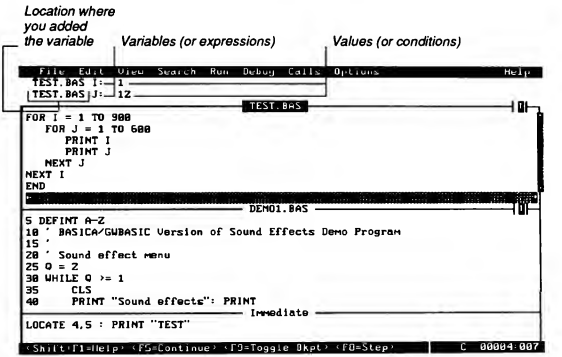


Figure 10.9 Watch Window

The Watch window opens whenever you choose the Add Watch or Watchpoint command, or when you choose Add from the Instant Watch dialog box. Add Watch, Watchpoint, and Instant Watch are all on the Debug menu. Each Watch window entry contains the following information:

- Location where you added the variable.

  This part of the entry names the module or procedure from which you added an expression to the Watch window.

- Variable or expression.

  The variable or expression you want to track.

- Value or condition.

  In Figure 10.9, the variables `I` and `J` have values `1` and `12`, respectively. Since the program is currently executing in `TEST.BAS` and both `I` and `J` were added to the Watch window when `TEST.BAS` was in the View window (indicated by the `TEST.BAS` entry in front of both `I` and `J`), both of their values appear. If they had been added from a different location such as a procedure, then the procedure name would appear in the Watch window and `Not watchable` would appear in the value location while the program executed in `TEST.BAS`.

See Also

Section 17.3.4, "Watch Window"; Chapter 18, "The Debug Menu"

10.7 Using the Mouse

You can use QuickBASIC with a mouse. Terms associated with the mouse and used in this manual include the following:

Mouse Term	Description
Mouse cursor	Block cursor that moves on the screen as you move the mouse.
Click	Placing the mouse cursor on an item, and pressing and releasing the left mouse button.
Double click	Placing the cursor on an item and pressing the left mouse button twice in a row.
Drag	Placing the mouse cursor at the beginning of what you wish to select. Press the left mouse button and hold it down. Move the pointer to the end of your desired selection and release the button. Use this technique to highlight words, manipulate window sizes, and use scroll bars.

To choose commands with the mouse follow these steps:

1. Point to the menu name and click the left mouse button.
2. Point to the command you want to select and click the left mouse button.

NOTE *Use only the left mouse button to choose QuickBASIC menus or commands. Clicking the right mouse button is the same as pressing* `F1` *(accessing on-line help). For information on how to set the right mouse button to execute your program to the line containing the mouse cursor, see Section 20.3, "Right Mouse Command."*

Table 10.3 explains mouse techniques for accomplishing other tasks.

Table 10.3 Mouse Commands

Task	Mouse Technique
Close a menu	Move the mouse cursor off the menu and click the left button.
Make a window active	Click anywhere in the window.
Expand or shrink a View or Immediate window	Move the mouse cursor to the title bar and drag the title bar up or down.
Expand the active window to occupy the full screen	Click the maximize box, at the right of the title bar, or double click the mouse anywhere on the title bar.
Scroll text	Place the cursor on 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 one page at a time, place the mouse cursor 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.
Select programs and files	Double click the file name, click the file name and OK in a list box (use this technique with the Open Program, Merge, and Load File commands), or single click to highlight the file name and press ENTER.
Change directories	Double click a directory name to view its contents. (In most dialog boxes, directories and drives appear in a separate list to the right of the file list.) If the current directory is a subdirectory, two periods (. .) appear in the list. Double click the two periods to move up one directory level.
Display Instant Watch dialog box	Press and hold the SHIFT key while clicking the right mouse button.

NOTE *To open a program with the mouse, double click the file name in the list box. Clicking an item with a mouse selects that item. Double clicking an item in a list box selects the item, confirms all previous choices, and closes the dialog box.*

10.8 Using On-Line Help

QuickBASIC has an extensive on-line help system. Through on-line help, you can get information on virtually any QuickBASIC topic, menu, command, or keyword.

See Also

Chapter 21, "The Help Menu"

10.8.1 Help Features

QuickBASIC's on-line help consists of two parts: environment help (for error messages, menus, and commands) and the Microsoft QB Advisor (for language-oriented topics). Press ESC to clear either type of on-line help from your screen.

Environment help appears in dialog boxes; the QB Advisor appears in the Help window at the top fo the screen. The following special features are available within the QB Advisor:

- Hyperlinks connect related topics; they give you immediate access to all related information on a particular topic (see Section 10.8.2).
- Examples can be copied or pasted directly into your program. You may resize the Help window, use commands such as PGUP and the DIRECTION keys, and scroll. You can even use Find from the Search menu to locate specific information within a particular help screen.
- Placemarkers can be used. If you use a particular screen frequently, set a placemarker there as you would in program text. Later, you can quickly return to the same screen. (See Section 12.4, "Using Placemarkers in Text.")
- QuickBASIC remembers up to the last 20 hyperlinks you accessed. Rather than repeatedly search through on-line help, you can press ALT+F1 to trace back through as many as 20 help screens.

You can access on-line help in four ways:

1. Move the cursor to the word in your program you want help on, and press F1.
2. Select the Help button or press F1 when a help dialog box is displayed.
3. Use hyperlinks from within Help windows to call up more information on related topics (see Section 10.8.2).
4. Select a command from the Help menu.

10.8.2 Hyperlinks

The QB Advisor provides interconnections called "hyperlinks" between related subjects. With a few keystrokes, you can access all of the information and related material on a particular topic.

Hyperlinks appear at the top and bottom of the Help window between highlighted (on monochrome monitors) or green (on most color monitors) triangles. Any BASIC keyword also forms a hyperlink to that keyword's on-line help.

To use a hyperlink, make the Help window active (press SHIFT+F6), and press the TAB key to move the cursor to the hyperlink you want to activate. Alternatively, you can enter the first letter of a hyperlink's name and the cursor will jump to that hyperlink. Press F1 to call up the screen associated with that hyperlink.

NOTE *Mouse users can place the mouse cursor on a hyperlink and click the right button, provided the right button function is set to invoke context-sensitive help (the default setting).*

Figure 10.10 shows examples of hyperlinks in the **PRINT** Help screen. The `Remarks` and `Example` hyperlinks provide additional remarks and examples.

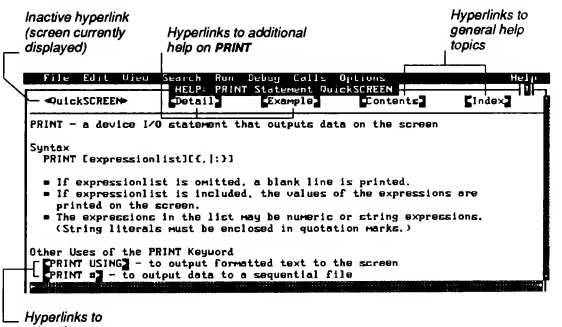


Figure 10.10 Help on the PRINT Statement

10.8.3 Moving in Help Windows

You can use the keyboard to move in Help windows. For example, you can use the DIRECTION keys. You can also use the key sequences described in Table 10.4.

Table 10.4 Help Keys

Key Sequence	Description
SHIFT+F1	Displays Help on Help
ESC	Clears help from the screen
TAB	Moves to next hyperlink
SHIFT+TAB	Moves to previous hyperlink
character	Moves to next hyperlink starting with the letter *character*
SHIFT+*character*	Moves to previous hyperlink starting with the letter *character*
PGUP	Displays previous screen (if one exists) of current topic
PGDN	Displays next screen (if one exists) of current topic
CTRL+F1	Displays help for the topic stored just after the current help topic
SHIFT+CTRL+F1	Displays help for the topic stored just before the current help topic

10.8.4 Help Files

When you choose any help command, QuickBASIC searches for the appropriate help file. QuickBASIC searches the current working directory, then the directory specified by the Set Paths command (Options menu), and finally any removable disks present in your system. If QuickBASIC cannot find an appropriate file, it displays the dialog box shown in Figure 10.11. Copy the missing file into the appropriate directory.

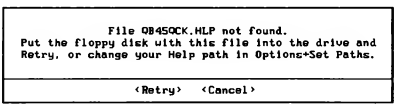


Figure 10.11 Dialog Box for Missing Help File

If you cannot invoke any form of on-line help, you may be missing all of the help files; in this case, you need to check the contents of your QuickBASIC directory or rerun the setup procedure. Or you may have inadequate memory to display on-line help. This situation can occur if you try to run QuickBASIC concurrently with other memory-resident files. In this case, remove one or more memory-resident files and start QuickBASIC again. You may also have inadequate memory if your machine doesn't meet the minimum system requirements (see the introduction to this manual).

10.8.5 Hard-Disk System

If you use a hard disk, you installed all of the help files in a single directory during your setup (see Chapter 1, "Setting Up Microsoft QuickBASIC," for more information). The Set Paths command on the Options menu specifies the directory path to the help files.

When you invoke language-oriented help, you will automatically receive the QB Advisor.

10.8.6 Removable-Disk System

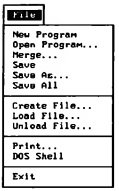
Microsoft QuickBASIC provides some language help on the same disk that contains QuickBASIC. This file contains statement descriptions and syntax, but not the detailed examples available in the QB Advisor.

To use the QB Advisor on a dual removable-disk system, press either the `Remarks` or `Example` hyperlinks at the top of the screen. A dialog box asks you to insert the Microsoft QB Advisor disk.

Once you insert the Microsoft QB Advisor disk, you can access more detailed help on QuickBASIC's keywords. You may leave the Microsoft QB Advisor disk in the drive until QuickBASIC prompts you for another disk.

CHAPTER 11

11 The File Menu

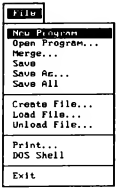


The File menu lets you work with files in QuickBASIC. From the File menu you can create new files, load existing files, modify files, or remove files. You also use the File menu to print files and exit from QuickBASIC.

The File menu has the following commands:

- New Program. Clears any previously loaded program and lets you begin a new program.
- Open Program. Opens a program; lists system files and directories.
- Merge. Merges the contents of two files (Full Menus only).
- Save. Writes the contents of the file in the active View window to a disk file (Full Menus only).
- Save As. Saves the current file with the name you specify.
- Save All. Saves all currently loaded files (Full Menus only).
- Create File. Begins either a new program module, an include file, or a document file as part of the current program (Full Menus only).
- Load File. Loads an existing file—either a program module, an include file or a document file—into memory (Full Menus only).
- Unload File. Removes an entire file from memory (Full Menus only).
- Print. Lets you print all or part of your program.
- DOS Shell. Returns temporarily to the DOS command level (Full Menus only).
- Exit. Removes program from memory and returns to the DOS prompt.

11.1 New Program Command



The New Program command from the File menu clears all previously loaded files so you can begin typing a completely new program. If a loaded program contains unsaved text, QuickBASIC asks if you want to save it before clearing it from memory.

A program contains one or more BASIC statements that QuickBASIC will translate into instructions for your computer. BASIC programs can be contained in a single file. However, QuickBASIC also allows you to build programs containing multiple self-contained parts ("modules"). Each module usually contains procedures that are used to perform specific tasks. One such module may be called into several programs. The New Program command creates the main module of a multiple-module program. In a single-module program, the main module is the only module.

A program can have four types of files:

- One main module (required)
- Other modules
- Include files
- Document files (including .MAK files)

Modules can contain the following:

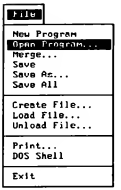
- Module-level code
- SUB procedures
- FUNCTION procedures

NOTE *When you are writing a completely new program, use the New Program command. When you are calling up a program that already exists on disk, use the Open Program command.*

See Also

Section 11.2, "Open Program Command"; Section 11.7, "Create File Command"

11.2 Open Program Command



The Open Program command opens an existing program. The dialog box displayed initially lists the files with .BAS extensions in the current working directory. You can also list files in other directories and on other disks on your system.

When you choose the Open Program command, the dialog box shown in Figure 11.1 appears.

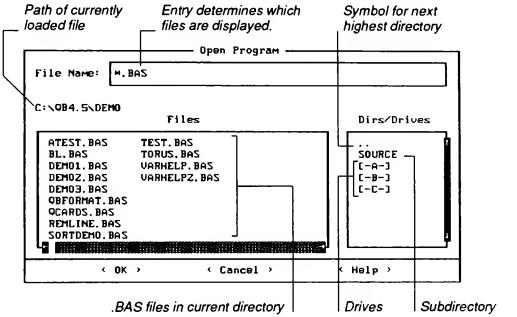


Figure 11.1 Open Program Dialog Box

If a program consists of several modules, all of the modules belonging to the program are loaded. If a program uses include files (see Section 2.5.4.3, "Using Include Files for Declarations," and Appendix F, "Metacommands," in *Programming in BASIC*), the include files are compiled, but not loaded or displayed.

11.2.1 Specifying a File

Files are displayed in columns in the Files list box. Directories and available drives appear in the Dirs/Drives list to the right of the Files list. Use the DIRECTION keys or the scroll bars to scroll in either of the list boxes. Note that the Files list box scrolls left and right, and the Dirs/Drives list box scrolls up and down.

The Open Program dialog box provides two ways to specify the file you want to load (see Section 10.7, "Using the Mouse," for mouse techniques). Use either of the following two methods:

1. Type the name of the program in the text box and press ENTER.

   If you enter a file name with no extension, QuickBASIC assumes your file has the .BAS extension. If you want to load a file that has no extension, type a period (.) immediately after the file name.

2. Press the TAB key to move to the list box, and use the DIRECTION keys to move through the list box until the desired file is highlighted. Then press ENTER.

   You can also highlight the file name by moving to the list box and pressing the first letter of the name of the file you want to load.

In either case, the text of the program appears in a window with the file name in the title bar.

11.2.2 Listing Directory Contents

You can use the Open Program dialog box to list the contents of any directory on your system. When you choose a directory name in the Dirs/Drives list box, QuickBASIC lists all the subdirectories and .BAS files in the directory you choose. When you enter or choose a file name from the Files list box, that file is loaded into memory. The following list describes several methods you can use to list contents of directories:

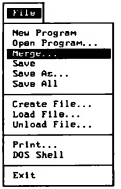
Task	Action
Listing all files in the current directory	Type `*.*` in the text box.
Listing files in root directory of drive A	Type `A:\*.*` in the text box.
Listing all files in a subdirectory named BIN	Highlight `BIN` in the Dirs/Drives list box and press ENTER. Type `*.*` in the text box.
Listing all files in the current directory with 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, press a DIRECTION key or the period (.) to select `..` if you are already in the Dirs/Drives list box. Then type `*.*` in the text box.

NOTE *The "current working directory" is the directory shown by the CD command from DOS. In 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.*

See Also

Section 11.1, "New Program Command"; Section 11.8, "Load File Command"

11.3 The Merge Command (Full Menus Only)



The Merge command on the File menu inserts the contents of a file at the beginning of the line the cursor is on. When you choose the Merge command from the File menu, the dialog box shown in Figure 11.2 appears.

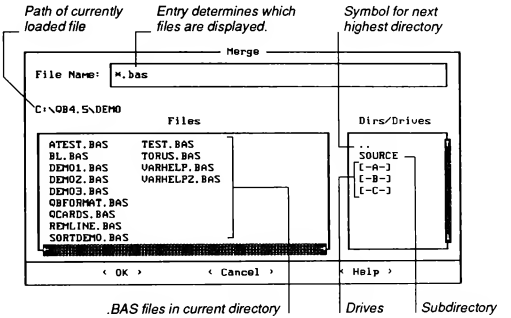


Figure 11.2 Merge Dialog Box

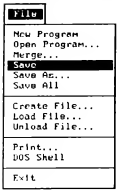
The Merge dialog box operates the same way as the Open Program dialog box. However, the Merge command inserts the specified file into the current file, whereas the Open Program command clears whatever is currently in memory, then loads the program.

NOTE *QuickBASIC cannot merge binary files; the file brought in with the Merge command must be text format.*

See Also

Section 11.2, "Open Program Command," and Section 14.1, "SUBs Command," in this manual; Chapter 7, "Programming with Modules," and Appendix B, "Differences from Previous Versions of QuickBASIC," in *Programming in BASIC*.

11.4 Save Command (Full Menus Only)



The Save command saves the contents of the current file (the file being displayed in the active View window) to a disk file.

If the file you are saving already has a name, Save overwrites the version on disk. If the file does not have a name, the Save dialog box in Figure 11.3 appears to ask you for a name.

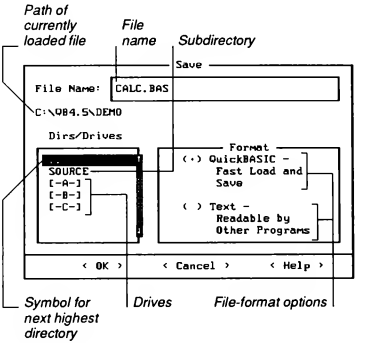
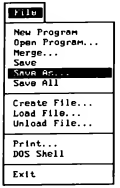


Figure 11.3 Save Dialog Box

See Also

Section 11.5, "Save As Command"; Section 11.6, "Save All Command"

11.5 Save As Command



The Save As command saves the current file with the name you specify. It is useful for saving a file 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, a dialog box like that shown in Figure 11.3 appears. The Save As dialog box lists the existing name of the file you want to save. Enter a new name in the text box to replace the old name. The next time you save the file, the new file name will appear in the text box.

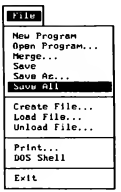
The following list describes the file-format options in the Save dialog box:

Option	Purpose
QuickBASIC—Fast Load and Save	Saves a program in QuickBASIC format. A program saved this way loads faster than one saved as a text file but can be edited only using QuickBASIC. Because it is a non-ASCII file, you will not be able to change it using another text editor. This is QuickBASIC's default format for saving a program. Files classified as document or include files with the Create File or Load File command cannot be saved in this format.
Text—Readable by Other Programs	Saves your file to disk as a text (ASCII) file. Text files can be read, modified, or printed by any editor or word processing program that reads ASCII files. Files classified as document or include files are always stored in this format.

See Also

Section 11.4, "Save Command"; Section 11.6, "Save All Command"

11.6 Save All Command (Full Menus Only)



The Save All command saves all currently loaded files that have changed since the last time you saved a file. A currently loaded file is one whose name appears in the dialog box of the View menu's SUBs command. Save All is useful when you are working with multiple files in memory.

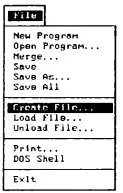
If the main module you are saving already has a name, Save All overwrites the version on disk. If the main module does not have a name, a dialog box like that shown in Figure 11.3 asks you for a name. Section 11.5 explains each of the file-format options.

When you save a multiple-module program, QuickBASIC creates a special file on your disk that contains the names of all the modules in the program. The file is given the base name of the main module plus the extension .MAK. QuickBASIC uses the .MAK file as a roadmap to the various program modules the next time you load the program. For more information on .MAK files, see Section 16.7.2.

See Also

Section 11.4, "Save Command"; Section 14.1, "SUBs Command"

11.7 Create File Command (Full Menus Only)



The Create File command on the File menu allows you to begin a new file as part of the program currently in memory.

When you choose the Create File command from the File menu, a dialog box appears. To create a new file, type the file name in the text box, select the file type, and press ENTER. See Figure 11.4.

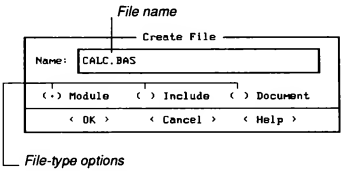


Figure 11.4 Create File Dialog Box

Select the appropriate file type from the following:

File Type	Description
Module	A discrete program component. A program 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 long programs or use the same groups of procedures over and over in different programs. QuickBASIC's smart editor checks syntax and formatting in modules. See Section 12.5 in this manual for information on the smart editor. See Chapter 7 in *Programming in BASIC* for more information on modules.
Include	A text file whose statements are compiled into your program when QuickBASIC encounters the file's name following a **$INCLUDE** metacommand. QuickBASIC's smart editor checks syntax and formatting in include files as it does in modules. Although the contents of the include file are used in your program, the include file itself is not a part of the program. Its name does not appear in the .MAK file of a multiple-module program.
Document	A text file. QuickBASIC does not treat a file opened as a document as BASIC code, so the smart editor does not check document files for syntax and formatting. However, you can edit a document file with the QuickBASIC editor just as you would with a normal word processor. Document file names never appear in the .MAK file of a multiple-module program.

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

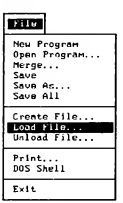
While a file's name appears in this list box, you cannot load it again, even with a different classification. For example, if you create a file as a document, you cannot load it again as an include file until you unload the current copy with the Unload File Command (see Section 11.9, "Unload File Command").

NOTE *Use the Create File command to add a new module to the program currently in memory. Create File differs from Open Program, which loads a file as a module after unloading all other files prior to loading the program. If no other modules are loaded when you use Create File to create a module, it is considered the main module of the program.*

See Also

Section 11.1, "New Program Command"; Section 11.2, "Open Program Command"; Section 11.8, "Load File Command"; Section 14.1, "SUBs Command"

11.8 Load File Command (Full Menus Only)



The Load File command on the File menu loads single program modules—as well as include files and document files—from disk into memory. When you choose Load File, a dialog box appears. To load a file, type the file name in the text box, select the file type, and press ENTER. See Figure 11.5.

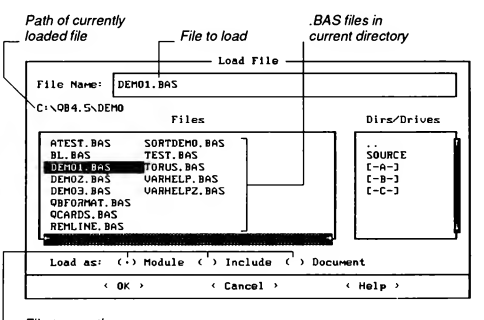


Figure 11.5 Load File Dialog Box

You must select the correct file type: Module, Include, or Document. See Section 11.7, "Create File Command," for more detail on specific file types.

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

While a file 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 it with the Unload File command. See Section 11.9 for more information on Unload File.

If you have procedures in other modules (program files) that you want to use in the current program, you can load those modules using Load File. Once the file is loaded, you can reference the procedures it contains anywhere in the current program.

Modules from other programs can be added into the program currently in memory as follows:

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.
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.

Save the program using the Save All command once all the modules are loaded.

See Also

Section 11.1, "New Program Command"; Section 11.7, "Create File Command"

11.9 Unload File Command (Full Menus Only)

The Unload File command removes an include file, document, or entire module from memory. After you unload a module, the disk file containing the module still exists; however, when you save your program, the module is no longer a part of the program.

If your program contains more than one module, the name of the unloaded module is removed from the program's .MAK file. If you unload the main module of a multiple-module program, QuickBASIC prompts you to set a new main module before proceeding. Unloading a document or include file leaves any other loaded files in memory.

When you choose the Unload File command, the dialog box shown in Figure 11.6 appears.

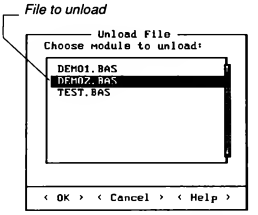


Figure 11.6 Unload File Dialog Box

Follow these steps to unload a file:

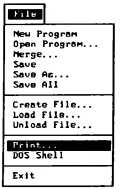
1. Choose the Unload File command from the File menu.
2. Select the name of the module, include file, or document that you no longer want in the program.
3. Press ENTER.

The module is unloaded but still exists as a disk file. You can use Unload File to unload any type of file—a program module, include file, or document file—from memory.

See Also

Section 11.1, "New Program Command"; Section 16.7.2, "The .MAK File"

11.10 The Print Command



The Print command on the File menu lets you print selected text, text in the active window, the entire current module, or all currently loaded program modules. To use the Print command, your printer must be connected to your computer's LPT1 port.

When you choose the Print command from the File menu, the dialog box shown in Figure 11.7 opens. Select the appropriate option from the Print options box to print your file.

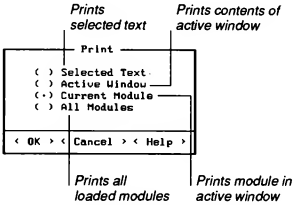
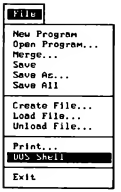


Figure 11.7 Print Dialog Box

11.11 DOS Shell Command (Full Menus Only)



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.

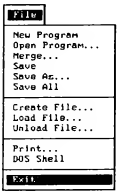
QuickBASIC needs to find the COMMAND.COM file before it can execute the Shell command. QuickBASIC looks for COMMAND.COM first in the directory specified in the COMSPEC environment variable, then in the current directory. See your DOS documentation for more information about COMMAND.COM and COMSPEC.

Follow these steps to return to QuickBASIC from the DOS command level:

1. Type `exit`
2. Press ENTER

The QuickBASIC screen reappears as you left it.

11.12 Exit Command



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 the dialog box shown in Figure 11.8.

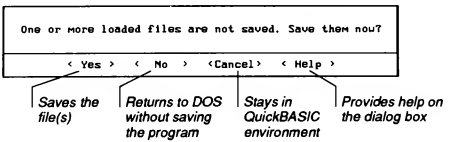


Figure 11.8 Exit Dialog Box

To exit from QuickBASIC after the Exit dialog box has been called up, use one of these procedures:

1. To save the program in its current state and then exit, choose Yes or press ENTER.

   If this is a new program that you haven't named yet, QuickBASIC displays a dialog box similar to the Save dialog box. See Section 11.4 for more information on this dialog box.

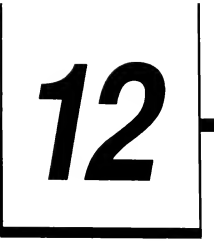
2. To return to DOS without saving the file in its current state, choose No or type N.

To stay in the QuickBASIC environment (that is, not exit to DOS after all), press ESC.

See Also

Section 11.5, "Save As Command"; Section 11.11, "DOS Shell Command"

CHAPTER 12
Using the Editor



This chapter describes how to enter and edit program text using QuickBASIC editing commands. Before beginning this chapter, you should be familiar with the QuickBASIC screen and the techniques described in Chapter 10, "Getting Around in QuickBASIC."

This chapter includes information on

- Performing simple editing functions such as entering text and moving the cursor
- Deleting and inserting text
- Moving and copying blocks of text
- Searching for and replacing characters, words, or a group of words
- Copying text from other files

12.1 Entering Text

New characters are added to the text either by inserting or overtyping. In "insert mode," the QuickBASIC editor inserts each new character to the left of the cursor position. In "overtype mode," each new character replaces the character under the cursor. You switch between insert mode and overtype mode by pressing the INS key or CTRL+V. The cursor is a blinking underscore in insert mode and a blinking box in overtype mode.

When typing a QuickBASIC program, you can enter a line (tell QuickBASIC that you have completed one or more BASIC statements) either by pressing the ENTER key at the end of the line or by moving the cursor off the line. If the Syntax Checking command is on (Options menu, Full Menus only), QuickBASIC checks the line for proper statement syntax.

12.2 Selecting Text

Before you can perform editing functions that manipulate blocks of text (copying and deleting), you must indicate the portion of text you wish to edit by selecting (highlighting) it. Selecting tells both you and the computer which text will be changed by the next command.

Hold down the SHIFT key while using the DIRECTION keys or any of the editing shortcut keys (see Section 10.2.2, "Using Shortcut Keys") to select text. For example, press SHIFT+END to select everything from the cursor to the end of a line. To remove highlighting, press any DIRECTION key.

IMPORTANT *If you begin typing while the text is still selected, what you type replaces the selected text.*

12.3 Indenting text

Indent text to improve the readability and structured appearance of your program. Appropriate use of indenting can aid debugging. 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.

Indenting is an integral part of well-structured programs—programs that are highly readable, logical and self-explanatory. QuickBASIC offers several indentation controls, listed in Table 12.1

Table 12.1 QuickBASIC Indentation Controls

Keystrokes	Description
(Automatic)	Indents each new line automatically, maintaining the current indentation level.
HOME	Cancels automatic indentation on a blank line and moves cursor to the left margin.
BACKSPACE	Moves indentation back one level.
CTRL+Q+S	Moves cursor to the left margin of a line containing indented text.
SHIFT+TAB	Removes leading spaces for one indentation level. If a block of text is selected, the whole block is moved to the left.

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

1. Choose the Display command from the Options menu.
2. Select 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 Option menu's Display dialog box sets the number of spaces per indentation level.

Some text editors use literal tab characters to represent multiple spaces when storing text files. Refer to Appendix B, "Differences from Previous Versions of BASIC," in *Programming in BASIC* if you work with files originally created with such an editor or if you suspect that your text editor uses literal tab characters.

See Also

Section 20.1, "Display Command"

12.4 Using Placemarkers in Text

If you are working on different parts of a large program, you can tell QuickBASIC to remember where in the program you are working (set a "placemarker"). This makes it possible to jump between different markers as needed. You can set up to four placemarkers, numbered 0–3, anywhere in your program.

Follow these steps to use placemarkers:

- Put the cursor on the line you want to mark. Press CTRL+K and then *n* to set placemarker number *n*.
- Press CTRL+Q and then *n* to move to placemarker number *n*.

12.5 The Smart Editor

QuickBASIC Version 4.5 combines a text editor, a compiler, and a debugger into a single "smart editor." The smart editor includes special features 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.

12.5.1 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 begin typing, QuickBASIC assumes that you want to write BASIC statements and turns on the smart editor. The smart editor is also on when you do either of the following:

- Choose New Program or Open Program from the File menu.
- Choose Create File or Load File from the File menu, then select the Module or Include option in the dialog box.

You may occasionally need to edit a text file consisting of something other than BASIC statements, such as a QuickBASIC .MAK file or document file. For this purpose, you can turn off the smart-editor features and use the environment as an ordinary word processor. Do this by choosing Create File or Load File from the File menu, then selecting the Document option in the dialog box.

12.5.2 Automatic Syntax Checking

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

```
PRINT GOTO
```

When an error occurs, QuickBASIC displays an error message. Press ESC or SPACEBAR to clear the message from the screen and position the cursor where the error occurred. You must correct syntax errors before your program will run.

NOTE *When you clear a syntax-error message, the cursor is placed on the statement that caused the error. Press 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 enter the following line instead of the **PRINT** keyword:

```
pront
```

Although this does not look like a valid BASIC statement, QuickBASIC does not flag the error until you try to run the program. Until then, QuickBASIC interprets the statement as a call to a **SUB** procedure named `pront`.

Syntax checking is always on unless you choose Create File or Load File commands and choose the Document option. You can toggle syntax checking off and on with the Syntax Checking command from the Options menu. A bullet (•) appears next to the command when syntax checking is toggled on.

12.5.3 Error Messages

QuickBASIC translates statements to executable code as they are entered, if there are no syntax errors. If there are syntax errors, QuickBASIC displays a dialog box containing the appropriate error message. If QuickBASIC displays an error message that puzzles you, you can get further information. For more information on the error message, choose the Help command button in the dialog box. To get more information on the use of the statement that caused the error, choose OK in the dialog box and do either of the following steps:

- Choose the Topic command on the Help menu.
- Press F1

Often you will see the error message `Syntax error`, but some error messages are more specific. For example, suppose you enter the following line, which is missing a file name:

```
OPEN, I, 1,
```

QuickBASIC generates the syntax-error message `Expected: expression` as shown in Figure 12.1:

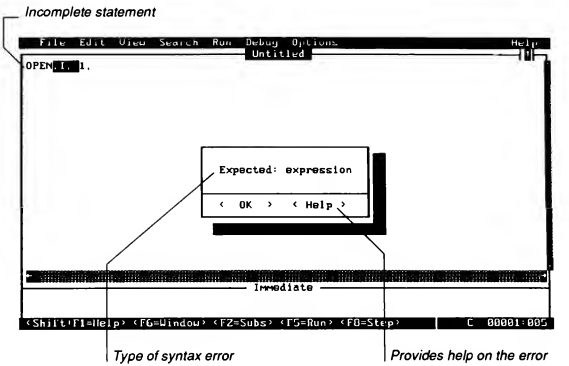


Figure 12.1 Syntax Error Message

In the case above, do the following to get on-line help on the `OPEN` statement:

1. Choose OK in the dialog box.
2. Place the cursor in the `OPEN` keyword.
3. Press F1.

The initial on-line help screen is displayed, as shown in Figure 12.2.

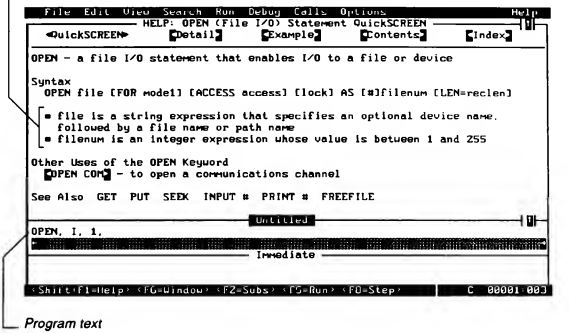


Figure 12.2 Initial Help Screen on the OPEN Keyword

In the screen shown in Figure 12.2, choose the usage you are interested in by putting the cursor anywhere on the appropriate line and pressing F1. Notice that your program text remains visible on the screen when you are using on-line help. To exit on-line help, press ESC.

See Also

Chapter 8, "Statement and Function Summary," and Appendix I, "Error Messages," in *Programming in BASIC*.

12.5.4 Automatic Formatting

In addition to checking for syntax errors, the smart editor automatically standardizes the format of each line as you enter it. The editor makes the following changes:

- Changes all BASIC keywords to uppercase letters.
- Inserts spaces before and after all BASIC operators.
- Keeps capitalization consistent in all references to variables or procedures.
- Adds punctuation where appropriate. For instance, the smart editor inserts semicolons between consecutive items in a **PRINT** statement if you do not include them yourself. It also 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. It changes the capitalization of all occurrences of a variable or procedure name to the reflect the most recently entered version. For example, if you first name a variable `MyVar` (uppercase `M` and `V`), and later enter it as `myvar` (all lowercase), QuickBASIC changes all instances of `MyVar` to `myvar`.

12.6 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 literal characters makes your programs more readable but less portable. For portability, use the **CHR$** function. See Section 9.5, "Summary of String Processing Statements and Functions," in *Programming in BASIC*.)

High-order ASCII characters are those corresponding to decimal 128–255. You can enter them directly in your QuickBASIC program by holding down the ALT key while you type 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.

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. See Appendix D, "Keyboard Scan Codes and ASCII Character Codes," in *Programming in BASIC* for decimal equivalents of ASCII characters. 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 CTRL-key sequence to produce its literal character.

2. Hold down the CTRL key again, this time pressing 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 reference bar, your line would be deleted. Note that you cannot use this technique for entering the characters corresponding to the following: ^J, ^L, ^M, ^U. See Appendix D, "Keyboard Scan Codes and ASCII Character Codes," in *Programming in BASIC* for a complete list of the ASCII characters.

12.7 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 12.2 summarizes all the QuickBASIC editing commands.

Table 12.2 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 file	---	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*†
Inserting		
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, saving in Clipboard	SHIFT+DEL	---

Table 12.2 *(continued)*

Keypad Moving Cursor	Keys	WordStar-Style Key Combinations
Selected text, not saving in Clipboard	DEL	CTRL+G
Leading spaces for one indentation level of selected lines	SHIFT+TAB	
Copying		
Save selected text to Clipboard	CTRL+INS	---
Scrolling		
Up one line	CTRL+UP	CTRL+W
Down one line	CTRL+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	---

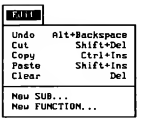
† Replace *n* with a number from 0–3 to identify the desired marker. (Release CTRL+K before typing *n*.)

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

§ Selects only the current line if the cursor is on the first character. Otherwise it selects the current line and the line below.

CHAPTER 13

13 The Edit Menu



The Edit menu controls the functions used to build and manipulate your program code (text). Use the Edit menu to cut, copy, and paste text, to undo your last edit, and to create **SUB** and **FUNCTION** procedures.

The Edit menu has the following commands:

- Undo. Reverses your last command or action, restoring the current line to its previous state (Full Menus only).
- Cut. Deletes selected text and places it on the Clipboard.
- Copy. Places a copy of selected text on the Clipboard.
- Paste. Inserts the contents of the Clipboard at the cursor.
- Clear. Deletes selected text permanently (Full Menus only).
- New SUB. Creates a new **SUB** procedure in its own window (Full Menus only).
- New FUNCTION. Creates a new **FUNCTION** procedure in its own window (Full Menus only).

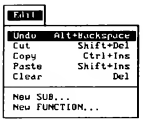
13.1 Understanding the Clipboard

The editor in QuickBASIC uses a Clipboard, which is the part of your computer's memory used to temporarily store blocks of text you have cut or copied. If you have used a word processor to cut, copy, and paste text, you are familiar with the Clipboard concept.

When you use the Cut or Copy command to move, remove, or duplicate a section of text, that text is stored in the Clipboard. You can then use the Paste command to insert the Clipboard's contents as many times as you like; Paste does

not alter the contents of the Clipboard. But any time you use the Cut or Copy command, the contents of the Clipboard are replaced by new contents.

13.2 Undo Command (Full Menus Only)



The Undo command on the Edit menu reverses any changes made to the current line. Once the cursor moves off the line, however, Undo cannot restore the line to its previous state.

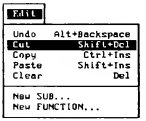
Undo works on the current line immediately after you make changes to it. Choosing Undo on an unaltered line has no effect.

NOTE *Undo cannot restore text deleted with CTRL+Y, which removes an entire line. However, a copy of the line is placed on the Clipboard, so you can restore it with the Paste command.*

Shortcut Key

ALT+BKSP

13.3 Cut Command



The Cut command removes selected text from the screen and places it on the Clipboard. When no text is selected, the Cut command is not available.

Use Cut and Paste together to relocate lines or blocks of text as follows:

1. Select the text you wish to move (it may be as many lines as you wish).
2. Choose the Cut command from the Edit menu.
3. Move the cursor to the point where you wish to insert the text.
4. Choose the Paste command from the Edit menu.

You can use this sequence to move entire blocks of a program from one section to another, reducing the time it takes you to write new programs.

NOTE *The Clipboard contains only the most recently cut or copied text.*

Shortcut Key

SHIFT+DEL

See Also

Section 13.4, "Copy Command"; Section 13.5, "Paste Command"; Section 13.6, "Clear Command"

13.4 Copy Command

The Copy command places a copy of selected text on the Clipboard, leaving the original text unchanged. Copying replaces any previous contents on the Clipboard.

Use the Copy and Paste commands together to duplicate sections or lines of program code. For example, you can use the following sequence to speed the creation of similar **SUB** procedures:

1. Select the text you wish to duplicate (it may be as many lines as you wish).
2. Choose the Copy command from the Edit menu.
3. Move the cursor to the point at which you wish to insert the text.
4. Choose the Paste command from the Edit menu.
5. Customize the copied text for a new line, procedure or section of program text.

NOTE *The Clipboard contains only the most recently cut or copied text.*

Shortcut Key

CTRL+INS

See Also

Section 13.3, "Cut Command"; Section 13.5, "Paste Command"

13.5 Paste Command

The Paste command inserts a copy of the Clipboard's contents to the right of the cursor.

Paste works only when the Clipboard contains text. If you have text selected, the Paste command replaces the selected text with the contents of the Clipboard. If no text is selected, Paste inserts the contents of the Clipboard to the right of the cursor, or above the cursor if the Clipboard contents exceed one line.

Shortcut Key

SHIFT+INS

See Also

Section 13.3, "Cut Command"; Section 13.4, "Copy Command"

13.6 Clear Command (Full Menus Only)

The Clear command completely removes selected text from your file.

Cleared text is not transferred onto the Clipboard and cannot be replaced by using the Paste command. Pressing the DEL key without text selected removes the character to the right of the cursor.

WARNING *Use the Clear command with care; you can restore cleared text only by using the Undo command immediately.*

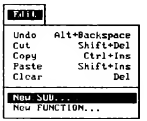
Shortcut Key

DEL

See Also

Section 13.2, "Undo Command"; Section 13.3, "Cut Command"

13.7 New SUB Command (Full Menus Only)



The Edit menu's New SUB command creates and names a new SUB procedure as part of the module in the active window. The next time you save your program, all new procedures are added to the program's module.

SUB procedures can be defined in the main module or other modules, and can be moved from one module to another by using the SUBs command from the View menu.

Use SUB procedures for repetitive operations, including ones which manipulate information (arguments) you give the SUB. For example, you might define a SUB procedure `CenterText`, which centers text on the screen. If you enter the following line in your program:

```
CALL CenterText("Hourly wage")
```

the phrase `Hourly wage` appears in the center of your screen (the actual SUB code is omitted here).

IMPORTANT *A procedure name can appear only once in a program. An error occurs if the same name appears more than once.*

See Also

Chapter 5, "The QCARDS Program," in this manual; Chapter 2, "SUB and FUNCTION Procedures," in *Programming in BASIC*.

13.7.1 Creating a New SUB Procedure

Create a new **SUB** procedure by following these steps:

1. Choose New SUB from the Edit menu. A dialog box appears.
2. Type the name for the new **SUB** procedure in the text box, as shown in Figure 13.1.

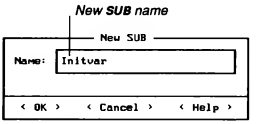


Figure 13.1 New SUB Dialog Box

3. Press ENTER.

The New SUB command opens a new window with the **SUB** and **END SUB** statements already in place, ready for you to enter the procedure. When you complete the procedure, use the SUBs command from the View menu to return to the module-level code, another procedure, or another module.

Note that if you type `SUB` *subname*, QuickBASIC starts the procedure immediately, bypassing the need to open the menu or dialog box.

When you start a new "procedure" (the collective term for **SUB** and **FUNCTION** procedures) either by typing **SUB** or **FUNCTION** followed by the procedure name or by using the New SUB or New FUNCTION command from the Edit Menu, QuickBASIC displays the procedure in the View window.

13.7.2 Default Data Types for Procedures

Data type, determined by a **DEF***type* statement, refers to the type of data a particular range of names will have. The **DEF***type* statement is a generic term that collectively refers to the data definition statements **DEFINT** (define integer), **DEFSNG** (define single precision), **DEFDBL** (define double precision), **DEFLNG** (define long integer), and **DEFSTR** (define string). A range of letters follows the **DEF***type* statement, indicating the range of names affected by the **DEF***type* statement. For example, `DEFDBL M-Q` defines all variables with names beginning with letters in the range M–Q to be double precision.

In a new procedure, QuickBASIC automatically inserts a **DEF***type* statement at the top of the View window, ahead of the line containing the **SUB** keyword:

```
DEFtype letter1-letter2
SUB SubName
END SUB
```

or

```
DEFtype letter1-letter2
FUNCTION FunctionName
END FUNCTION
```

In the preceding example, `letter1` and `letter2` indicate the letter range the **DEF***type* statement affects. Note that you may omit `letter2` if you want to define only a single letter as a specific data type.

Which **DEF***type* is automatically added to the procedure depends on whether there are **DEF***type* statements at the module level. If there are no **DEF***type* statements in the main module, QuickBASIC creates the procedure without a **DEF***type* statement, and the new procedure defaults all variables to single precision. If there are one or more **DEF***type* statements at the module level, QuickBASIC includes all **DEF***type* statements other than **DEFSNG** in the new procedure. Any variable that falls outside of the letter range indicated by any of the other **DEF***type* statements defaults to single precision.

13.7.3 Changing a Procedure's Default Type

If QuickBASIC inserts a **DEF***type* statement you do not want, you can remove it (in which case the default for that procedure becomes single precision), or you can change it by overtyping.

When you have no **DEF***type* statements in your main module, QuickBASIC omits the **DEF***type* statement in new procedures (and procedure variables default to single precision). You can add new **DEF***type* statements to the procedure by placing them in front of the **SUB** or **FUNCTION** statement (in insert mode) and pressing ENTER.

13.7.4 Saving and Naming Procedures

When you save a program, QuickBASIC creates a declaration for each procedure at the beginning of the module containing the procedure. A declaration consists of the **DECLARE** keyword, followed by the **SUB** or **FUNCTION** keyword, then the name of the procedure, and a list of its formal parameters.

Procedures are not saved as independent files the way modules are. Therefore procedure names are governed by the naming conventions of QuickBASIC rather than DOS. They are saved within the module in which they are created, or to which they are moved.

The rules for naming procedures are the same as those for naming variables. Procedure names can be any valid BASIC name up to 40 characters long. They must be unique. If you give a procedure the same name as a variable, the error message `Duplicate definition` appears.

See Also

Section 13.8, "New FUNCTION Command"; Section 14.1, "SUBs Command"

13.8 New FUNCTION Command (Full Menus Only)

The New FUNCTION command defines (creates and names) a new **FUNCTION** procedure as a part of the module or program in the active window. When you save the module, any new procedures become part of that module.

Create your own **FUNCTION** procedures whenever you repeatedly calculate similar quantities. For example, you might define a **FUNCTION** procedure `Volume (X)` that calculates the volume of a sphere of radius `X`. Your program then finds the volume of a sphere of any radius very easily:

```
SphereVol = Volume (rad)    ' find volume of a sphere of
                            ' radius rad
```

QuickBASIC sets the value of `X` equal to `rad` when it calculates the volume of the sphere. If you need to calculate the volume of a sphere several times, such a **FUNCTION** procedure could save you space, time, and debugging efforts.

IMPORTANT *A procedure name can appear only once in a program. An error occurs if the same name appears more than once.*

You can define a new **FUNCTION** procedure by following these steps:

1. Choose New FUNCTION from the Edit menu. A dialog box appears.
2. Type the name for the new **FUNCTION** procedure in the text box. See Figure 13.2.

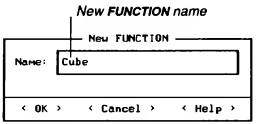


Figure 13.2 New FUNCTION Dialog Box

3. Press ENTER.

QuickBASIC opens a new window with the **FUNCTION** and **END FUNCTION** statements already in place. When you complete the **FUNCTION** procedure, use the SUBs command from the View menu to return to the previous module.

Keyboard Shortcut

Type `FUNCTION` followed by the name of the new procedure to perform steps 1–3 in the previous list automatically.

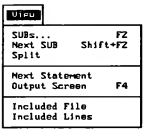
See Also

Chapter 5, "The QCARDS Program"; Section 13.7.2, "Default Data Types for Procedures"; Section 13.7.3, "Changing a Procedure's Default Type"; Section 13.7.4, "Saving and Naming Procedures"; Section 14.1, "SUBs Command." Also see Chapter 2, "SUB and FUNCTIONS Procedures," in *Programming in BASIC*."

CHAPTER 14

The View Menu

14

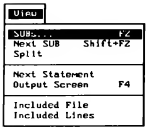


The View menu gives the available options for viewing program components. You also use the View menu to move SUB and FUNCTION procedures, modules, include files, and document files in and out of the View window for viewing and editing.

The View menu includes the following commands:

- SUBs. Moves procedures, modules, include files, and document files into and out of windows and moves procedures among modules.
- Next SUB. Places the next procedure (alphabetical by name) in the active window (Full Menus only).
- Split. Divides the View window horizontally (Full Menus only).
- Next Statement. Moves the cursor to the next statement to be executed (Full Menus only).
- Output Screen. Toggles the screen display between the program output and the QuickBASIC editor.
- Included File. Loads referenced include files for viewing and editing (Full Menus only).
- Included Lines. Displays the contents of referenced include files for viewing only.

14.1 SUBs Command



The SUBs command lets you select among the currently loaded files, their procedures (**SUB** and **FUNCTION**), include files, and document files. When you choose a file or procedure, QuickBASIC displays it in the View window.

The SUBs dialog box controls file and procedure handling.

To use the SUBs command, follow these steps:

1. Choose the SUBs command from the View menu. The SUBs dialog box appears, as shown in Figure 14.1.

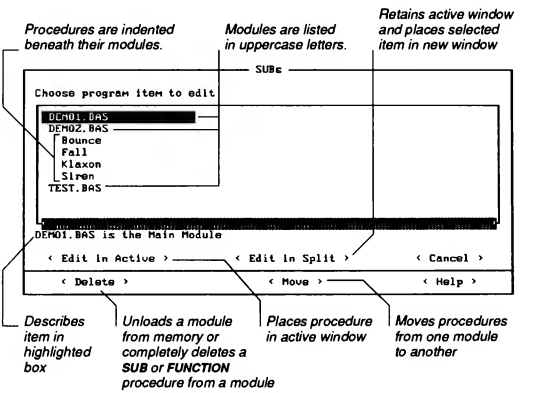


Figure 14.1 SUBs Dialog Box

2. Highlight the module, procedure, or include file you want to work with by pressing the DIRECTION keys. Then you can do one of the following:

   - Choose the Edit in Active button to place your selection in the active window.

- Choose the Edit in Split button to split the window horizontally; the lower (active) window displays the new selection and the upper window contains the contents of the previous window. Use F6 to cycle down through the windows or SHIFT+F6 to cycle up. To close the extra window choose the Split command from the View menu again.
- Choose the Move button to move the selected procedure to a different module. A dialog box appears with a list of destination modules. See Figure 14.2. (Moving a procedure in a single-module program has no effect.)

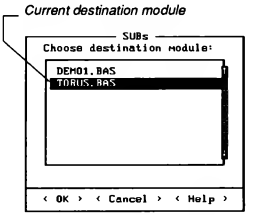


Figure 14.2 Move Dialog Box

- Choose the Delete button to remove the selected procedure completely from the module. (Note that deleting a module removes that module and all of its associated procedures from memory but keeps the module on disk.)
- Choose Cancel to return to the previous active window.
- Choose Help for more information on using the SUBs dialog box.

You are now ready to work with the module, procedure, or include file you selected.

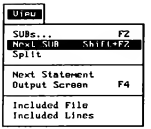
Shortcut Key

F2

See Also

Section 11.9, "Unload File Command"; Section 14.2, "Next SUB Command"; Section 16.7, "Set Main Module Command"

14.2 Next SUB Command (Full Menus Only)



The Next SUB command displays the contents of the next procedure (alphabetically by name) in the active window. If the current module contains no procedures, choosing Next SUB has no effect. You may cycle either forward to the next procedure of backward to the previous one.

Next SUB is of particular value when working with modules that contain only a few procedures. For programs with many procedures, the SUBs command is generally faster for moving between program parts.

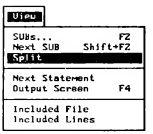
Shortcut Keys

SHIFT+F2 (forward) and CTRL+F2 (backward)

See Also

Section 14.1, "SUBs Command"

14.3 Split Command (Full Menus Only)



The Split command divides the View window horizontally, allowing you to work with two parts of a program simultaneously.

To use the Split command follow these steps:

1. Choose Split from the View menu.
2. Press F6 to cycle down through the windows, making the different windows active, or SHIFT+F6 to cycle up through the windows.
3. Choose Split again to fill the screen with the active window and close the inactive window.

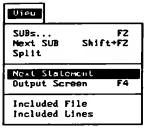
You can increase or decrease the size of a window or expand it to fill the entire screen. To change the size of a window, make it the active window and use the following key combinations (use the PLUS and MINUS keys on the numeric keypad):

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. If it already fills the screen, the window is returned to its former size.

See Also

Chapter 10, "Getting Around in QuickBASIC"; Section 14.1, "SUBs Command"

14.4 Next Statement Command (Full Menus Only)



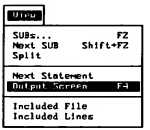
The Next Statement command places the cursor at the next statement to be executed but does not execute the statement.

The Next Statement command is typically used after pausing a program for debugging purposes; it allows you to see which statement the Continue command will execute first when you choose it. Next Statement is useful after moving around in a suspended program to return to the original point of suspension.

See Also

Section 15.1, "Find Command"; Section 16.3, "Continue Command"; Chapter 18, "The Debug Menu"

14.5 Output Screen Command



The Output Screen command causes QuickBASIC to switch between the environment and output screens created by your program.

This command is functional at any time during editing, but is most useful after the program sends data to the screen. For example, when you run a program that has screen output, QuickBASIC prints the message

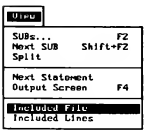
```
Press any key to continue
```

on the bottom of the output screen when the program ends. After returning to the QuickBASIC editor, you can switch back and forth between the editor and output screens by choosing the Output Screen command. This allows you to refer to previous output while editing or debugging your program.

Shortcut Key

F4

14.6 Included File Command (Full Menus Only)



The Included File command loads a file referenced with the **$INCLUDE** metacommand for viewing and editing.

When the cursor is on a **$INCLUDE** metacommand line, choosing the Included File command loads the text of the include file into the active window for editing. QuickBASIC keeps the 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. However, QuickBASIC considers an include file separate from the rest of the program.

You may wish to edit an include file without moving the cursor from the line in your program containing the **$INCLUDE** metacommand. To do so, follow these steps:

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

Save the include file to incorporate your changes. QuickBASIC prompts you if you forget.

14.6.1 Nesting Include Files

You can have any number of **$INCLUDE** metacommands in a program. However, the "nesting" of include files is limited.

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.

14.6.2 Finding Include Files

When QuickBASIC processes a **$INCLUDE** metacommand, it searches first for the include file in any directory specified in the metacommand, then the working directory, and finally any include-file directory indicated by the Set Paths command. If you refer to a directory in the **$INCLUDE** metacommand itself, you

need to include a full or relative path name. For example, the following 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:

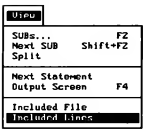
```
' $INCLUDE: '..\include\declares.bi'
```

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.

See Also

Section 14.7, "Included Lines Command"; Section 20.2, "Set Paths Command"

14.7 Included Lines Command



The Included Lines command lets you view, but not edit, the contents of referenced include files. This command is a toggle; a bullet (•) appears next to the option when it is on. Choosing it again turns it off.

The Included Lines command 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 include file after choosing Included Lines, you cannot edit the include file as long as this option is on. If you try to edit the include file while Included Lines is on, 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 the program file while the Included Lines command is on, a dialog box appears asking if you want to turn the Included Lines command off. If you press ENTER, the include file disappears, and you can then edit the source file in the usual way.

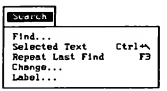
For editing files referenced with the **$INCLUDE** metacommand, use the Included File command or the Load File command on the File menu (remember to choose the Include option).

See Also

Section 14.6, "Included File Command"

CHAPTER 15

15 The Search Menu



The Search menu lets you find specific text—"target" text—anywhere in a program, and replaces it with new text. The Search menu has several uses, particularly in long or complex programs. For instance, to move quickly to a distant part of your program, you can search for a line label or other identifying text you know to be in that section.

The Search menu has the following commands:

- Find. Searches for and locates the nearest occurrence of designated text.
- Selected Text. Searches for text that matches the currently highlighted text (Full Menus only).
- Repeat Last Find. Locates the next occurrence of the text specified in the previous search (Full Menus only).
- Change. Searches for and then replaces target text with new text.
- Label. Searches for a specified line label (Full Menus only).

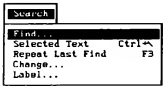
15.1 Defining Target Text

Search procedures require you to specify your target text—the text you want to search for. QuickBASIC uses several methods to specify target text. The word containing the cursor or closest to the left of the cursor becomes the default target text. If the cursor is on a blank line, the target text from the previous search becomes the default. You can alter default target text by typing new text in the Find or Change dialog boxes.

All searches begin at the cursor and move forward. If QuickBASIC reaches the end of the active window without finding a match, it returns to the top of the

document and continues the search until it either locates the text or reaches the original cursor location. You can also select options that extend the search to the entire current module or all modules loaded.

15.2 Find Command



The Find command searches for and places the cursor on specified text. The text may consist of a single character, a word, or any combination of words and characters.

Enter the text you want to find in the text box of the Find dialog box, shown in Figure 15.1.

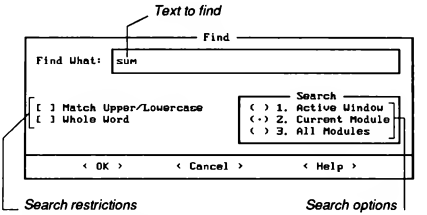


Figure 15.1 Find Dialog Box

The dialog box also controls the extent of the search (whether the search includes the active window, the current module, or all files currently loaded in QuickBASIC) and any restrictions on the search.

To perform a search, do the following:

1. Choose Find from the Search menu. A dialog box appears.
2. Enter the text you want to find in the Find What text box. (Any default target text appears highlighted.)
3. Select any search option buttons by pressing the TAB and DIRECTION keys. (See Table 15.1 for a list of available options.)

4. Set any restrictions by pressing the TAB key to select the option and SPACEBAR to turn the option on or off. (See Table 15.2 for a list of restrictions.)
5. Press ENTER to begin the search.

If it cannot locate the target text, QuickBASIC displays the message `Match not found` and leaves the cursor where it was when you started the search.

Press ENTER or ESC to remove the message and continue.

The options available with search commands are listed in the following table:

Table 15.1 Search Options

Option	Description
Active Window	Searches only in the current, active window
Current Module	Searches only in the current module (including procedures)
All Modules	Searches all parts of all loaded files

The following table lists the restrictions you can set with the Find command:

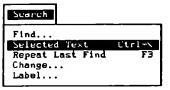
Table 15.2 Search Restrictions

Restriction	Description
Match Upper/Lowercase	Searches for text that matches exactly. For example, if `CHR` is the specified text, the Match Upper/Lowercase option finds `CHR` but not `Chr`.
Whole Word	Locates the target as a separate word. Whole words include words surrounded by spaces, punctuation marks, or other characters not considered parts of words. Word parts include A–Z, a–z, 0–9, and !, #, $, %, and & (commonly used in declarations). For example, a whole word search for `CHR` would locate `CHR:` but not `CHR$` or `CHARACTER`.

See Also

Section 15.5, "Change Command"

15.3 Selected Text Command (Full Menus Only)



The Selected Text command on the Search menu searches for text that matches text selected in the active window.

Follow these steps to use the Selected Text command:

1. Select the text you want to find. The text must fit on one line (256 characters).
2. Choose the Selected Text command from the Search Menu.

QuickBASIC highlights the next occurrence of the selected text and puts the cursor at the first character of the string. If there are no other occurrences, the screen remains the same.

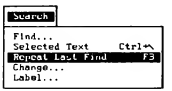
Shortcut Key

CTRL +\

See Also

Section 15.2, "Find Command"

15.4 Repeat Last Find Command (Full Menus Only)



The Repeat Last Find command searches for the next occurrence of the text specified in the last search. Text specified in the previous Find or Change command becomes the target text of a Repeat Last Find search.

If you have not used the Find or Change commands and you choose the Repeat Last Find command, QuickBASIC uses the word containing the cursor, or the word nearest to the left of the cursor, as the default target text.

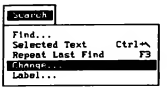
Shortcut Key

F3

See Also

Section 15.2, "Find Command"

15.5 Change Command



The Change command searches for one text string and replaces it with another. Either string may consist of a single character, a single word, or any combination of words and characters.

Use the Change dialog box, shown in Figure 15.2, to enter the target text, the replacement text, and search attributes and restrictions.

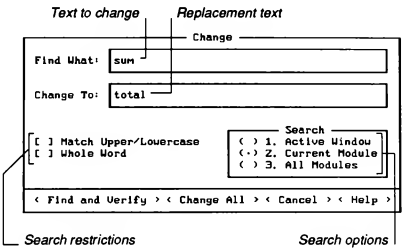


Figure 15.2 Change Dialog Box

To change text, do the following:

1. Choose Change from the Search menu.
2. Type the text you want to replace in the Find What text box.
3. Type the replacement text in the Change To text box.
4. Select any search options you desire (see Table 15.1).
5. Set any search restrictions (see Table 15.2).
6. Choose Find and Verify or Change All to begin replacing text, or Cancel to cancel the command.

   - Find and Verify finds the target text, then displays four more buttons: Change, Skip, Cancel, and Help. Change executes the change, Skip searches for the next occurrence (without making any changes), and

Cancel stops the search and returns to the point at which the search began. Help provides more information about the Find and Verify dialog box. See Figure 15.3.

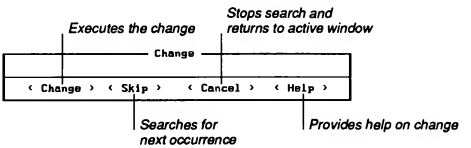


Figure 15.3 Change, Skip, Cancel Dialog Box

- Change All finds and replaces all occurrences of the target text without asking for confirmation.
- Cancel returns control to the active window.
- If it cannot locate the target text, QuickBASIC displays the message

  `Match not found`

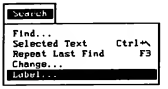
  Press ENTER to remove the message and continue.

IMPORTANT *The Undo command on the Edit menu cannot undo a change made with the Change command. Use particular caution with Change All since this command can drastically alter your program.*

See Also

Section 15.2, "Find Command"

15.6 Label Command (Full Menus Only)



The Label command searches for a line label. Since line labels require a colon, QuickBASIC searches for your text-box entry plus a colon. QuickBASIC searches only for the target text followed by a colon. For example, if you enter `mylab` as your target text in the Label dialog box, QuickBASIC searches for `mylab:`

The Label dialog box is shown in Figure 15.4.

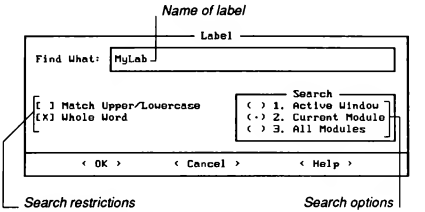


Figure 15.4 Label Dialog Box

See Also

Section 15.2, "Find Command"; Section 15.3, "Selected Text Command"

CHAPTER 16

16 The Run Menu

The Run menu contains commands that let you control program execution within QuickBASIC, compile stand-alone programs with the BC compiler, and create Quick libraries. This allows you to create and debug your program within QuickBASIC, then compile your running program for greater speed. As stand-alone applications, your programs can be run from the DOS command line—without being loaded into QuickBASIC. You also use the Run menu to define main program modules.

The Run menu has the following commands:

- Start. Runs the currently loaded program from within the QuickBASIC environment.
- Restart. Resets all variables to zero and all string expressions to zero-length strings, and prepares to execute the first statement in the program.
- Continue. Resumes execution of a suspended program at the statement where execution was stopped (does not reset variables or expressions).
- Modify COMMAND$. Accesses the string passed to a program via the **COMMAND$** function from the DOS command line (Full Menus only).
- Make EXE File. Creates an executable file from the currently loaded program.
- Make Library. Creates a custom Quick library (Full Menus only).
- Set Main Module. Sets a selected module as the main module (Full Menus only).

16.1 Start Command

The Start command on the Run menu runs the currently loaded program.

Once you start a program you may stop it by pressing CTRL+BREAK. To continue running the program from where you stopped, choose the Continue command from the Run menu. To start from the beginning again, choose Start.

NOTE *If you load a BASICA program, some statements require modification to run properly with QuickBASIC. They are described in Appendix A, "Converting BASICA Programs to QuickBASIC," in Programming in BASIC.*

Shortcut Key

SHIFT+F5

See Also

Section 16.2, "Restart Command"; Section16.3, "Continue Command"; Section 16.5, "Make EXE File Command"

16.2 Restart Command

The Restart command on the Run menu resets all variables to zero and all string expressions to zero-length strings. It prepares your program to run, but does not begin execution. Restart also highlights the first executable statement in your program.

Restart is generally used during debugging when you do not want to continue execution. It is especially useful if you want to restart a program from the beginning and step through program code (using the F8 key) prior to the first breakpoint or watchpoint. (Breakpoints and watchpoints are devices used to halt program execution. See Section 18.3, "Watchpoint Command," and Section 18.8, "Toggle Breakpoint Command.") If you want to execute to the first breakpoint or watchpoint without stepping through code, use the Start command.

To run your program from a specific location without resetting any variables or expressions, use the Continue command on the Run menu or the Set Next Statement command on the Debug menu followed by the Continue command.

See Also

Section 16.1, "Start Command"; Section 16.3, "Continue Command"; Section 18.11, "Set Next Statement Command"

16.3 Continue Command

The Continue command continues an interrupted program from the last statement executed or starts a program from the beginning.

The Continue command is particularly useful for debugging. Use Continue to run a program from one breakpoint or watchpoint to another. When execution stops, you can analyze the code and make any corrections. Once you complete the corrections, choose Continue to execute from that point forward. If you make a change that prevents continuation from the current statement, QuickBASIC stops and asks if you want to continue without the change or if you prefer to include the change and restart the program. (If you do not want to restart, you can frequently use the Immediate window to make temporary changes that will allow the program to continue.)

Shortcut Key

F5

See Also

Section 16.1, "Start Command"; Section 16.2, "Restart Command"

16.4 Modify COMMAND$ Command (Full Menus Only)

The Modify COMMAND$ command is a feature helpful to advanced programmers who want to access text strings passed to their programs from the DOS command line via the **COMMAND$** function. You only need the Modify COMMAND$ command if your program uses the **COMMAND$** function.

In some applications, you may wish to pass information from the DOS environment to the application. For example, typing `QB DEMO1` lets QuickBASIC know that you wish to work within the program DEMO1 in the QuickBASIC environment. `DEMO1` is information that is passed to QuickBASIC from the command line.

You can pass information to your program from the DOS command-line using the **COMMAND$** function. Memory is cleared every time the program starts, and the information from the DOS command line is lost. In the development and debugging process, you will want to test the program without exiting to DOS each time you alter the program. The Modify COMMAND$ command tells QuickBASIC to substitute what you have entered in the dialog box for the string the **COMMAND$** function would normally return.

In the Modify COMMAND$ dialog box, you indicate the string to be returned by the COMMAND$ function. See Figure 16.1.

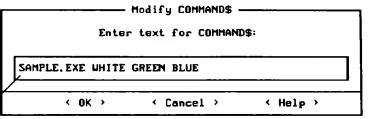


Figure 16.1 Modify COMMAND$ Dialog Box

16.5 Make EXE File Command

Once your program runs within QuickBASIC, you can use the Make EXE File command to make a version of the program that runs directly from the DOS command line without being loaded into QuickBASIC. The file for such a program is known as an executable file, and has the extension .EXE. Entering the base name (name with no extension) 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 (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 BRUN45.LIB or BCOM45.LIB.

16.5.1 Creating Executable Files

The files in the following list 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. You can also use the Options menu's Set Paths command to tell QuickBASIC where to search for these files. (See Section 20.2, "Set Paths Command," 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.

BRUN45.LIB (Full Menus only)	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 BRUN45.LIB library is the run-time-module library. Programs linked with this library have small executable files, but can only run in combination with BRUN45.EXE, the run-time module.
BCOM45.LIB	Provides supporting routines like BRUN45.LIB, except that executable files created with this library do not require BRUN45.EXE (run-time module) support. Files created with BCOM45.LIB are larger than those that use BRUN45.EXE, but will run as stand-alone programs.

Take the following steps to create an executable file from a program loaded in QuickBASIC:

1. Choose the Make EXE File command from the Run menu. The Make EXE File dialog box appears. See Figure 16.2.

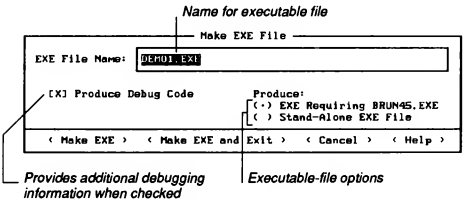


Figure 16.2 Make EXE File Dialog Box

2. Type a different base name in the text box if you want to rename your file; otherwise, leave the text box alone.
3. Choose either the EXE Requiring BRUN45.EXE option or the Stand-Alone EXE File option.

   See Section 16.5.3, "Types of Executable Files," for a discussion of these alternatives.
4. Choose Produce Debug Code if you want your executable file to include additional error-checking and to report error locations at run time. Also choose Produce Debug Code if you want your program to respond to CTRL+BREAK. Note that Produce Debug Code does not produce CodeView®-compatible

files and results in larger and slower programs. See Section 16.5.4, "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.

16.5.2 Quick Libraries and Executable Files

If your program uses routines from a Quick library, you must load the Quick library when you load your program. If a Quick library is loaded when you choose the Make EXE file command, the BC compiler searches for a stand-alone (.LIB) library that corresponds to the loaded 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 needs 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. You can use the Options menu's Set Paths command to tell QuickBASIC where to look for any .LIB or Quick libraries it needs when creating an executable file. (The Make Library command on the Run menu makes both kinds of libraries automatically. See Section 16.6, "Make Library," in this manual and Appendix H in *Programming in BASIC* for more information on creating and using Quick libraries.)

16.5.3 Types of Executable Files

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 BRUN45.EXE.

16.5.3.1 Programs that Use the Run-Time Module

Programs compiled with the EXE Requiring BRUN45.EXE option need access to the run-time module BRUN45.EXE when they run. The run-time module contains code needed to implement the BASIC language. Run-time modules provide the following advantages over stand-alone programs:

- The executable file is much smaller.
- COMMON variables and open files are preserved across CHAIN statements, allowing programs to share data. Stand-alone programs do not preserve open files or variables listed in COMMON statements when a CHAIN statement transfers control to another program.
- The BRUN45.EXE run-time module resides in memory, so it does not need to be reloaded for each program in a series of chained programs.

16.5.3.2 Stand-Alone Programs

When you use the Stand-Alone EXE File option, the executable file does not require access to the run-time module BRUN45.EXE. However, since files created with this option include the support routines found in BRUN45.EXE, they are larger than files created with the EXE Requiring BRUN45.EXE option. Also, files created with the EXE Requiring BRUN45.EXE option do not preserve open files or variables listed in COMMON statements when a CHAIN statement transfers control to another program.

Stand-alone files have the following advantages:

- Slightly faster execution than run-time programs.
- RAM space may be saved if you have small, simple programs that do not require all the routines in the run-time module.
- The program does not require the run-time module to run. This allows the program to run by itself on any DOS computer.

16.5.4 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 you make an executable file, the following conditions are checked when the program runs:

- Arithmetic overflow. All arithmetic operations, both integer and floating-point, are checked for overflow and underflow.
- Array bounds. Array subscripts are checked to ensure they are within the bounds specified in DIM statements.
- Line locations. The generated binary code includes additional tables so that run-time error messages 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 pressed CTRL+BREAK. If so, the program stops.

  Note that you must compile a program with the Produce Debug Code option turned on if you want it to respond to CTRL+BREAK. Otherwise, CTRL+BREAK halts a program only when one of the following conditions is met:

  1. A user enters data in response to an **INPUT** statement's prompt.
  2. The program's code explicitly checks for CTRL+BREAK.

WARNING *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.*

16.5.5 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. Executable files optimize for speed and accuracy by changing the order in which they do certain arithmetic operations. (They also make greater use of a numeric coprocessor chip or emulation of a coprocessor's function.)

One side effect of this extra accuracy in the results of relational operations is that you may notice a difference when comparing single- or double-precision values. For example, the following code fragment prints `Equal` when run in the QuickBASIC environment but `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 chip, 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"
```

16.6 Make Library Command (Full Menus Only)

The Make Library command creates a custom Quick library for use with compiled programs. A Quick library can contain procedures that you wish to use in more than one program.

When you make a library with QuickBASIC, consider whether the library is new or an update of an existing library. If it is an update, start QuickBASIC with the /L command-line option, supplying the name of the library to be updated as a command-line argument (for example, `QB /L Mylib`). 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 it is a new library, start QuickBASIC first, and use Load File to load the files you want to convert into a library.

If you load your program when starting QuickBASIC, be sure to unload any modules you do not want in the Quick library, including the main module (unless it contains procedures you want in the library).

16.6.1 Unloading and Loading Modules

To keep your Quick libraries as compact as possible, remove any nonessential modules currently loaded. To remove ("unload") the unwanted modules, follow these steps:

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. The module is unloaded, but still exists as a disk file.
3. Repeat steps 1 and 2 until you have unloaded all unwanted modules.

Alternatively, start QuickBASIC with or without a library specification, and load only those modules you want. In this case, you load each module using the Load File command from the File menu.

To load one module at a time, follow these steps:

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 you have loaded all the modules you want.

16.6.2 Creating Libraries

Once you load the previous library (if any) and 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 16.3 appears.

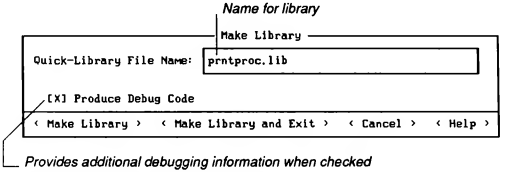


Figure 16.3 Make Library Dialog Box

To create a Quick library after you have chosen the Make Library command, do the following:

1. Enter the name of the library you wish to create in the Quick-Library File Name text box.

   If you enter a base name (that is, a file name with no extension), QuickBASIC appends the extension .QLB when it creates the Quick library. If you want your library to have no extension, add a terminating period (.) to the base name. Otherwise, you may enter any unique base name and extension you like, as long as it is 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 you are updating. It makes your library larger, your program slower, and gives only a small amount of error information.
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 *A Quick library that needs to trap events such as keystrokes must contain at least one event-trapping statement in one of its modules. This statement can be as simple as* ***TIMER OFF****, but without it, events are not trapped correctly in the Quick library.*

16.7 Set Main Module Command (Full Menus Only)

The Set Main Module command lets you select a main module from a list of the currently loaded modules.

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—it is the first entry in any list box (such as the one in the SUBs command dialog box) that shows the modules in a program.

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.

16.7.1 Changing the Main Module

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).

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, follow these steps:

1. Choose the Set Main Module command from the Run menu. The Set Main Module dialog box contains a list of currently loaded modules.
2. Select the module you want to designate as the main module, and press ENTER. See Figure 16.4.

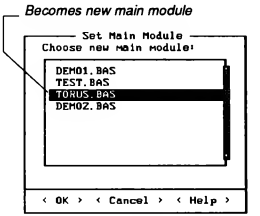


Figure 16.4 Set Main Module Dialog Box

16.7.2 The .MAK File

When you save a multiple-module program QuickBASIC creates a special file containing the names of all the program modules. This file has the base name of the main module plus the extension .MAK.

When you reload the program, QuickBASIC uses the .MAK file to locate all of the modules. For this reason, do not delete a .MAK file. If you move a multiple-module program to another directory, you also need to move the .MAK file.

Only program modules are included in a .MAK file. The main program module is the first .MAK file entry. QuickBASIC does not add other types of files to the .MAK list. If your program relies on include files, the program must specify each file as an argument to a **$INCLUDE** metacommand.

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

You can remove modules from a program's .MAK file without editing the .MAK file directly. With the program loaded, follow the module-unloading procedure outlined in Section 16.6.1, "Unloading and Loading Modules." When you save the program, the names of any unloaded modules are removed from the .MAK file.

CHAPTER 17

17 Debugging Concepts and Procedures

This chapter introduces the topic of debugging your programs with QuickBASIC. QuickBASIC can help you track down errors that result from flaws in the logic of a program.

This chapter provides the following information on debugging:

- Using QuickBASIC to prevent bugs as you are writing your program
- Debugging terms and concepts

See Chapter 18, "The Debug Menu," for more information on debugging.

17.1 Debugging with QuickBASIC

QuickBASIC simplifies debugging with advanced features such as breakpoints, watchpoints, instant watch, and watch expressions. Other fundamental tools include the **STOP**, **PRINT**, **CONT**, **TRON**, and **TROFF** statements.

With QuickBASIC, you do not need to use separate tools or invoke any special options to debug your program. When you notice a bug, you can start tracking it immediately. The QuickBASIC environment contains everything you need.

Additionally, QuickBASIC does not require you to recompile or relink a program after making a change. You can usually run a program to the point where a problem occurs, suspend execution, fix the problem, and continue running the program as though it never stopped. In QuickBASIC, you debug as you program.

17.2 Preventing Bugs with QuickBASIC

Preventing bugs is the best way to have a bug-free program. Here are three ways to prevent bugs:

1. Design your program carefully.

   An important rule in good program design is to isolate different tasks by writing **SUB** or **FUNCTION** procedures to perform those tasks. A small procedure is much easier to debug than a large program without procedures.

   An additional benefit of using procedures is that you can save groups of procedures in a separate file called a module, then use them in other programs that perform the same tasks. (See Chapter 2, "SUB and FUNCTION Procedures," in *Programming in BASIC* for more information on procedures and modules.)

2. Use the Immediate window.

   As you program, use the Immediate window at the bottom of the QuickBASIC screen to isolate and test small pieces of the program. See Section 17.3.5 for more information on the Immediate window. When these pieces work on their own, move them into the View window.

3. Run your program often.

   Because QuickBASIC checks and translates each statement to executable code (instructions understood by the computer) after you type a line and press ENTER, you can continually test-run the program as you add new pieces. This helps catch simple bugs that could be difficult to track in a finished program.

Suppose you have written a program that contains the following code:

```
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
```

You believe you have designed a logical loop. Now you can test the code by doing the following:

1. Execute the loop and enter text when prompted. (To end the loop, press ENTER instead of entering text.)
2. Type the following in the Immediate window:

```
For I% = 1 to 20: PRINT Array$(I%): NEXT
```

Your output will alert you to a flaw in your program—the variable `I%` is not incrementing. Upon closer inspection, you notice that you must change the line

```
I% = I + 1
```

to

```
I% = I% + 1
```

Along with designing your program carefully, using the Immediate window, and running your program often, another debugging technique is setting a "watch expression." This allows you to "watch" (monitor) the value of the variable `I%` as you trace through the loop. Section 17.3.3, "Watch Expressions," and Section 17.3.4, "Watch Window," tell how to watch an expression while a program is executing.

17.3 QuickBASIC's Debugging Features

This section defines some key debugging terms and concepts. See Chapter 18, "The Debug Menu," for information on the Debug menu and how to use QuickBASIC's debugging commands.

17.3.1 Tracing (Full Menus Only)

The Trace On command highlights each statement as it executes. If the program manipulates the screen, QuickBASIC switches back and forth between the highlighted statements and the output screen (this is referred to as "animated execution"). For example, with tracing on you can follow branching in an **IF...THEN...ELSE** statement. QuickBASIC's tracing options let you run the entire program at once (normal tracing), one statement at a time (single stepping), or one procedure at a time (procedure stepping). See Section 17.3.6, "Other Debugging Features," for a discussion of single stepping and procedure stepping.

Turning on tracing automatically enables QuickBASIC's History On command. History records the last 20 statements executed so you can stop execution and

trace backward and forward through the 20 previous statements with History Back and History Forward. The history feature allows you to answer the question, "How did I get here?"

See Also

Section 18.6, "Trace On Command"; Section 18.7, "History On Command"

17.3.2 Breakpoints and Watchpoints

A breakpoint is a location in your program where you want execution to pause. With breakpoints, you can set your program to execute only as far as you want, letting you test one segment of your program at a time. You can also use breakpoints to stop the program at strategic places and then examine a variable's value.

A watchpoint is an expression that stops a program when the expression becomes true (nonzero). You might use a watchpoint, for example, if your program crashes in a loop that increments `X` from `1` to `50`. A watchpoint of `X=49` stops the program when `X` becomes `49`. You can then single step the program through the forty-ninth loop to locate the particular statement that causes the crash.

Watchpoints appear in the Watch window. The Watch window displays all of the current watchpoints and watch expressions.

See Also

Section 17.3.4, "Watch Window"; Section 18.1, "Add Watch Command"; Section 18.8, "Toggle Breakpoint Command"

17.3.3 Watch Expressions

Watch expressions, which are entered with the Add Watch command, let you observe the value of a variable (a number or text string) or the condition of an expression (true or false) in an executing program. For example, if you want to watch the value of a variable `MyVar` and an expression `X=1`, QuickBASIC places `MyVar` and `X=1` in the Watch window. As the program runs, QuickBASIC displays the changing values of `MyVar` and the condition of `X=1` (–1 for true, 0 for false) in the Watch window. If a program `MYPROG.BAS` initially assigns `MyVar` a value of `10` and `X` a value of `1`, then the Watch window displays

```
MYPROG.BAS  MyVar:  10
MYPROG.BAS  X=1:    -1
```

If the program later changes `MyVar` to `99` and `X` to `3`, the Watch window will then display

```
MYPROG.BAS   MyVar:   99
MYPROG.BAS   X=1:     0
```

The Instant Watch command displays the value of the variable or expression containing the cursor in a dialog box—you do not need to open the Watch window. However, the Instant Watch dialog box does give you the option of adding the variable or expression to the Watch window.

See Also

Section 17.3.4, "Watch Window"; Section 18.1, "Add Watch Command"; Section 18.2, "Instant Watch Command"

17.3.4 Watch Window

The Watch window is the window that opens at the top of the QuickBASIC screen to let you track watchpoint conditions or watched expressions during program execution.

The current value or condition (true or false) of a variable or expression can be displayed in the Watch window only when the program is executing in the part of the program in which the watchpoint or watch expression was added. The message `Not watchable` is displayed at all other times.

For example, suppose you create a watch expression to monitor the value of a variable `MyVal` while you are working on a procedure called `MySub`. The value of `MyVal` will appear in the Watch window only when the program executes the `MySub` procedure. At any other time—when execution is in the module-level code or other procedures—the message `Not watchable` appears after `MyVal` in the Watch window.

Even if a 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 was active when the variable was added to the Watch window.

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

See Also

Section 17.3.3, "Watch Expressions"; Section 18.1, "Add Watch Command"; Section 18.2, "Instant Watch Command"

17.3.5 Immediate Window

The Immediate window is the window at the bottom of the screen. In it you can execute BASIC statements directly to gain information useful in debugging.

Note, however, that the Immediate window can access information only about those variables in the part of the program (module or procedure) that is currently active.

For example, if you suspend a program in a procedure, you may use the Immediate window to access local and global variables and expressions (indicated by the **SHARED** statement) but not ones local only to the module-level code or another procedure.

See also

Section 10.5, "Using the Immediate Window"

17.3.6 Other Debugging Features

Table 17.1 describes several additional debugging features available in QuickBASIC Version 4.5. Some are intended to be used together, while others are generally used by themselves.

Table 17.1 Additional Debugging Features

Feature	How to Execute	Description
Break on Errors command	Choose from the Debug menu	Locates errors in programs using error handling. Break on Errors disables error handling and halts execution when an error occurs. Press SHIFT + F8 (History Back) to see the error-causing statement.
Execute to cursor	Press F7	Executes from the beginning of a program up to and including the line containing the cursor, and then halts execution. Use this feature to quickly execute to a problem-causing section of code, and then single step to find the error.
Set Next Statement command	Choose from the Debug menu	Changes the execution sequence of an interrupted program so that the next statement to execute will be the one at the cursor. If your program encounters an error that you can correct immediately, use Set Next Statement to tell QuickBASIC to begin executing from the corrected line when you choose Continue.

Table 17.1 *(continued)*

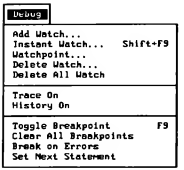
Feature	How to Execute	Description
Single stepping	Press F8	Executes a single statement. This is a powerful tool for debugging code. By single stepping and using the Watch window together, you can gain a thorough understanding of how each statement affects your variables.
Procedure stepping	Press F10	Executes a procedure call and the procedure as a single statement. If you are single stepping and encounter a procedure you know works properly, procedure stepping lets you execute the entire procedure at once (rather than single stepping through it).

See Also

Section 18.10, "Break On Errors Command"; Section 18.11, "Set Next Statement Command"

CHAPTER 18

18 The Debug Menu

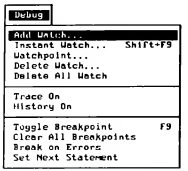


The Debug menu controls features that speed program debugging. These features let you watch variables and expressions as they change, trace program execution, stop the program at specified locations or on specified conditions, and alter the restart location of a suspended program.

The Debug menu contains the following commands:

- Add Watch. Displays the values of variables and expressions in the Watch window as a program executes.
- Instant Watch. Displays the value of a variable or expression.
- Watchpoint. Allows you to halt program execution when a condition you specify becomes true (Full Menus only).
- Delete Watch. Removes an item from the Watch window.
- Delete All Watch. Removes all entries in the Watch window and closes the window (Full Menus only).
- Trace On. Steps through the program in slow motion, highlighting the currently executing statement (Full Menus only).
- History On. Records the last 20 lines executed by the program (Full Menus only).
- Toggle Breakpoint. Turns breakpoints on and off.
- Clear All Breakpoints. Removes all previously set breakpoints.
- Break on Errors. Sets an implicit breakpoint at the most recent error-handling statement (Full Menus only).
- Set Next Statement. Changes the program execution sequence so that the next statement executed is the one at the cursor (Full Menus only).

18.1 Add Watch Command



The Add Watch command displays the values of variables and expressions as the program executes. QuickBASIC opens a Watch window at the top of the screen to accommodate the variables.

To use the command, choose Add Watch on the Debug Menu. A dialog box appears. Type in the name of the variable or expression you want to watch. See Figure 18.1.

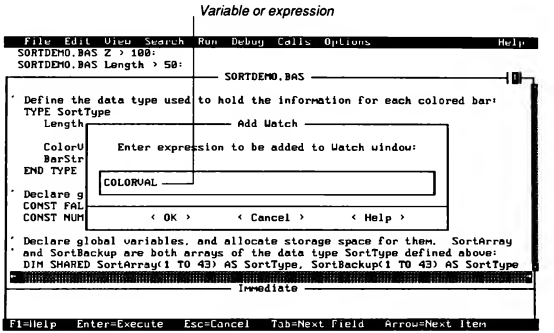


Figure 18.1 Add Watch Dialog Box

Watching a variable in the Watch window avoids repeated use of **PRINT** statements for tracking the variable's value. You can also watch the condition (true or false) of an expression in the Watch window. True is represented by `-1`; false by `0`. The variables or expressions you are watching are easily removed from the Watch window when debugging is complete.

During program execution, a value in the Watch window changes when its value in the program changes. Sometimes the message `Not watchable` may appear rather than a value. This means that the debugger cannot currently access the variable because program execution is not in the part of the program (the procedure- or module-level code) from which the variable was added to the Watch window. A variable is watchable only when the program is executing in the part of the program from which the variable was added to the Watch window.

Note that if you are interested in only one or two instances of a variable's value, it may be easier to use the Instant Watch command, described in Section 18.2.

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, beginning with 1 and 2. The first 10 Fibonacci numbers are 1, 2, 3, 5, 8, 13, 21, 34, 55, and 89. This example uses the Add Watch command and animated tracing (the process of switching back and forth between highlighted statements in your program and the output screen).

1. Start QuickBASIC or choose the New Program command from the File menu, and type:

```
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 and typing `Fib` in the dialog box. (`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 enter the procedure 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 on the Debug menu, and again add `n` to the Watch window. This displays the value of `n` within the `Fib` procedure.

5. Choose the Trace On command on the Debug menu.

6. Choose Start on the Run menu to execute the program.

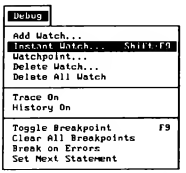
7. Enter `8` after the input prompt. Observe the Watch window to see how `n` changes as the procedure recursively calls itself to calculate the eighth number in the Fibonacci sequence.

8. Choose the Output Screen command (or press F4) to see the program results.

See Also

Section 17.3.3, "Watch Expressions"; Section 17.3.4, "Watch Window"

18.2 Instant Watch Command



The Instant Watch command displays the value of a variable or the condition of an expression (true or false) in a dialog box.

Use the command by moving the cursor within (or to the right of) a variable or selecting an expression and then choosing Instant Watch on the Debug menu. QuickBASIC displays a dialog box with the variable or expression in the upper text box and the value or condition (–1 for true, 0 for false) in the lower text box. See Figure 18.2.

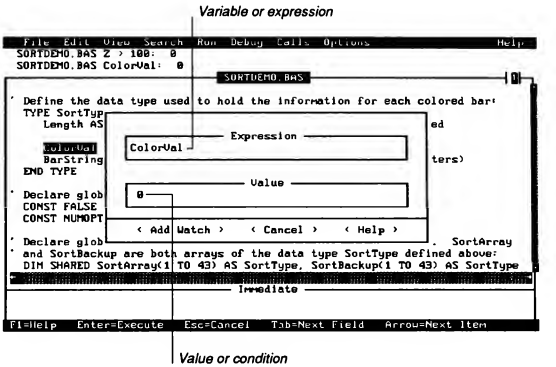


Figure 18.2 Instant Watch Dialog Box

Use the Instant Watch command for debugging assignment statements and for checking variables or expressions. If you need to watch a variable or expression constantly, use the Add Watch command (or press the Add Watch button in the Instant Watch dialog box) to add the item to the Watch window.

The Instant Watch command is convenient for displaying a value or condition in a suspended or stopped program.

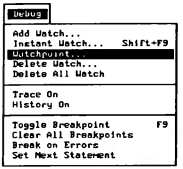
Shortcut Key

SHIFT+F9

See Also

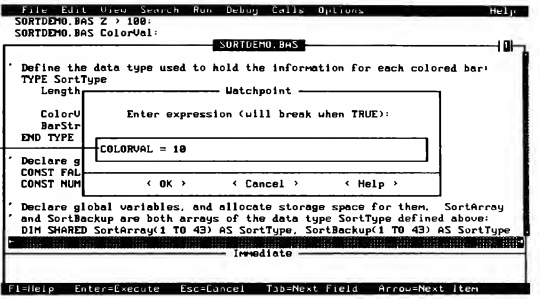
Section 17.3.3, "Watch Expressions"; Section 17.3.4, "Watch Window"

18.3 Watchpoint Command (Full Menus Only)



The Watchpoint command allows you to set a watchpoint—an expression that halts program execution when the expression becomes true (for example, when the value of a variable `X` equals 100). QuickBASIC displays the watchpoint in the Watch window.

When you choose the Watchpoint command a dialog box appears. You can enter any variable name or expression along with relational operators in the Watchpoint text box. See Figure 18.3.



Condition that will halt the program

Figure 18.3 Watchpoint Dialog Box

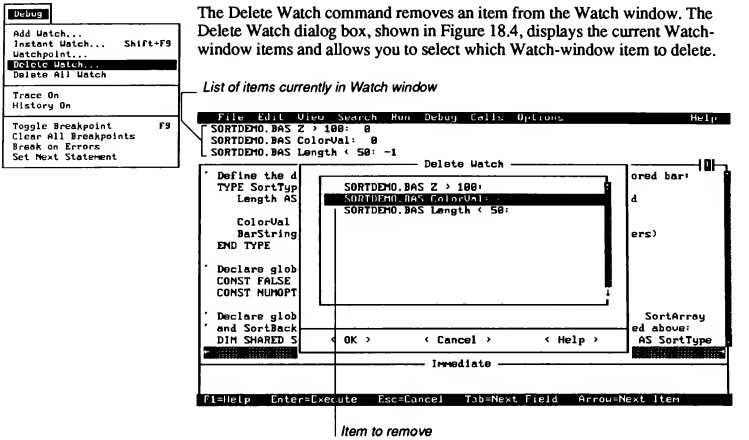
QuickBASIC's relational operators include =, <>, >, <, >=, and <=. If an expression is entered without a relational statement, QuickBASIC assumes the relation <> 0. QuickBASIC then suspends execution of the program as soon as the expression becomes a nonzero value.

If you set a watchpoint in a **SUB** or **FUNCTION** procedure, the Watch window describes the given expression as `TRUE` or `FALSE` only when the program is executing in that procedure. At all other times the message `Not watchable` will appear beside the watchpoint in the Watch window. Similarly, watchpoints at the module level display the message `Not watchable` when execution is in a procedure.

See Also

Section 17.3.2, "Breakpoints and Watchpoints"; Section 18.1, "Add Watch Command"; Section 18.4, "Delete Watch Command"

18.4 Delete Watch Command



The Delete Watch command removes an item from the Watch window. The Delete Watch dialog box, shown in Figure 18.4, displays the current Watch-window items and allows you to select which Watch-window item to delete.

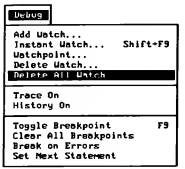
Figure 18.4 Delete Watch Dialog Box

Use the Delete All Watch command to remove all Watch-window items at once.

See Also

Section 18.5, "Delete All Watch Command"

18.5 Delete All Watch Command (Full Menus Only)



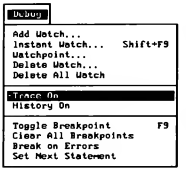
The Delete All Watch command in the Debug menu removes all of the Watch-window entries and closes the window. You typically use the Delete All Watch command when the section of program code you are working on runs properly.

If you want to selectively remove Watch-window items, use the Delete Watch command.

See Also

Section 18.4, "Delete Watch Command"

18.6 Trace On Command (Full Menus Only)



The Trace On command steps through the program in slow motion, highlighting the executing statement. When this command is active, a bullet (•) appears next to the Trace On command on the Debug menu. Choosing Trace On again toggles the feature off.

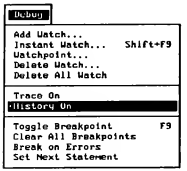
Trace On is a useful method for determining whether the general flow of your program is what you intended, especially if you use it in combination with watchpoints, breakpoints, and CTRL+BREAK (which suspends execution). Trace On also records the last 20 statements executed, so you can review them with the History Back and History Forward commands (see Sections 18.7.1 and 18.7.2).

If a program sends output to the screen, running it with Trace On active causes the screen to flash back and forth between QuickBASIC and the output screen. You can use the Output Screen command from the View menu to control which screen appears during program suspension (see Section 14.5).

See Also

Section 18.1, "Add Watch Command"; Section 18.7, "History On Command"; Section 18.8, "Toggle Breakpoint Command"

18.7 History On Command (Full Menus Only)



Choose the History On command from the Debug menu to record the last 20 lines executed by the program the next time it runs. When this command is active, a bullet (•) appears next to the History On command on the Debug menu. Execution speed is slower with this feature active.

The History On command allows you to trace backward and forward through the last 20 statements executed by the program. This is particularly useful when you get a run-time error and you want to see which 20 statements were executed immediately preceding the error. Checking the execution history also lets you review the specific branching of a program through nested sequences of conditional structures like **IF...THEN...ELSE** statements. Use History On in conjunction with breakpoints to review groups of up to 20 consecutive statements in your program.

You use History On with the History Back and History Forward commands (see Sections 18.7.1 and 18.7.2).

NOTE *The History commands do not execute statements. They just show the previous execution sequence.*

18.7.1 History Back

Activated with SHIFT+F8, History Back steps back in your program through the last 20 program statements recorded by either the History On or Trace On command.

When you wish to review the flow of a program, interrupt the program execution by pressing CTRL+BREAK; then press SHIFT+F8 to check back through the last 20 statements executed. This shows you the program flow through statements and is particularly useful for following conditional branches.

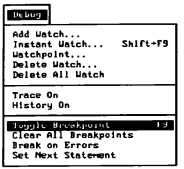
18.7.2 History Forward

Activated with SHIFT+F10, History Forward lets you step forward in your program through the last 20 program statements. You can use History Forward only after using History Back. Because History Back shows the reverse execution order, step backward first, and then watch the execution sequence using History Forward.

See Also

Section 18.6, "Trace On Command"

18.8 Toggle Breakpoint Command



The Toggle Breakpoint command turns a breakpoint (a specific location in the program where execution stops) on and off. The Toggle Breakpoint command affects the line containing the cursor. Lines on which breakpoints are set appear in reverse video.

Use a breakpoint to pause at points where you suspect problems in your program, then test the values of variables to confirm or refute your suspicion (in the Immediate window). Alternately, insert a breakpoint prior to a problem area and then use single stepping (see Section 17.3.6, "Other Debugging Features," for a description of single stepping) or use the Instant Watch command for a detailed view of program flow and variable values.

Clear a breakpoint on the current cursor line by selecting Toggle Breakpoint again.

Shortcut Key

F9

See Also

Section 17.3.2, "Breakpoints and Watchpoints"; Section 18.9, "Clear All Breakpoints Command"

18.9 Clear All Breakpoints Command

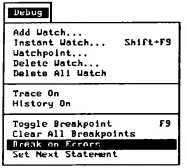
The Clear All Breakpoints command removes all breakpoints in a program. Remove individual breakpoints (while leaving others in place) with the Toggle Breakpoint command.

You typically use the Clear All Breakpoints command when you know a section of program code runs properly.

See Also

Section 17.3.2, "Breakpoints and Watchpoints"; Section 18.8, "Toggle Breakpoint Command"

18.10 Break on Errors Command (Full Menus Only)



The advanced debugging command Break on Errors allows you to trace an error-causing statement in a program that uses error handling. When toggled on (indicated by a bullet beside the command), Break on Errors sets an implicit breakpoint after the label specified in the most recently executed error-handling (**ON ERROR**) statement. If you do not use error handling, you will not need this command.

When an error occurs in a program with error handlers, QuickBASIC routes execution to the label specified in the appropriate **ON ERROR** statement. If there is no **ON ERROR** statement that explicitly addresses the type of error QuickBASIC encounters, finding the source of the error can be difficult. Break on Errors solves this error-finding problem.

The Break on Errors command enables History On and records the statement that caused the error. Choosing History Back (SHIFT+F8) shows you the error-causing statement.

In the following program fragment, an error occurs at the line `Y=5/X`. This example demonstrates how QuickBASIC uses Break on Errors to stop after the label specified in the most recent `ON ERROR` statement:

```
ON ERROR GOTO  FirstHandler  'First error handler
ON ERROR GOTO  NextHandler   'Second error handler
ON ERROR GOTO  LastHandler   'Third and last error handler
.
X = 0
Y = 5/X
.
LastHandler:  'Label in most recently executed error handler
.             'QuickBASIC stops immediately after label
.
```

With Break on Errors on, an error transfers program flow to `LastHandler` and execution stops. History Back will now show you the statement that caused the error.

See Also

Section 18.7, "History On Command"

18.11 Set Next Statement Command

When you want to skip over or rerun a section of a program, halt execution, then use the Set Next Statement command.

The Set Next Statement command changes program execution sequence so that the next statement executed is the one at the cursor. QuickBASIC ignores any intervening program code. Use the Set Next Statement command with care; skipping code can lead to undefined variables and result in errors.

The effect of the Set Next Statement command is similar to that of a GOTO statement, and the same restrictions apply. For example, you cannot branch to the middle of a procedure from the module level of a program.

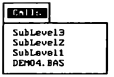
Note that the Next Statement command on the View menu moves the cursor to the next statement to be executed when the program continues to run, while the Set Next Statement command on the Debug menu allows you to establish (set) the next statement to be executed. Next Statement is used to indicate which statement will execute next if no changes are made; Set Next Statement allows you to alter this by choosing a new resumption point for program execution.

See Also

Section 14.4, "Next Statement Command"

CHAPTER 19

19 The Calls Menu (Full Menus Only)



The Calls menu is an advanced feature that shows all of the procedure calls that led to the current execution location. It lets you do the following:

- See the sequence of nested procedure calls your program executed in order to arrive at the current procedure.
- Continue your program from the point execution stopped to any listed procedure.

Use the Calls menu to see which procedures execution passed through. During debugging, you can use the Calls menu and single stepping (described in Section 17.3.6, "Other Debugging Features") together to execute up to an error, recreating the same variable values and conditions that led to a program crash.

If your procedures do not call other procedures, the Calls menu shows only the name of the main module and any currently executing procedure. In any program, the Calls menu displays only the program name prior to running.

19.1 Using the Calls Menu

The Calls menu shows calls between procedures. The following example illustrates the use of the Calls menu for both debugging and executing.

Suppose you want to write a program to draw histograms (bar charts) on the screen, and you want your user to specify the fraction of the screen the chart will fill. Since you believe strongly in the use of structured programming, you break the process into several procedures.

Your main module prompts your user for data and then calls a procedure `DrawHisto` to draw the chart. `DrawHisto` then sends the task of drawing the chart's bars to a separate procedure, `DrawBars`. Since the number of bars in the chart can vary (depending on the data your user entered), `DrawBars` farms out the job of calculating how wide each bar will be to the procedure `CalcBarWidth`. However, before `CalcBarWidth` can determine how wide each bar needs to be, it needs to know the width of the histogram. So, `CalcBarWidth` calls the procedure `CalcHistoWidth` which calculates the width of the screen to be used in the histogram.

During execution, `DrawHisto` calls `DrawBars`, `DrawBars` calls `CalcBarWidth`, and `CalcBarWidth` calls `CalcHistoWidth`. If your program stops running in `CalcHistoWidth`, the Calls menu items appear in the order shown in Figure 19.1.

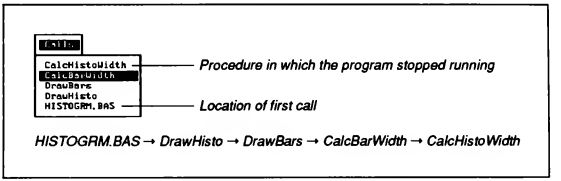


Figure 19.1 Sequence of Procedures on Calls Menu

The Calls menu provides you with a "trail" you can follow to understand how your program executed to the `CalcHistoWidth` procedure; it is particularly valuable when your program stops in a series of deeply nested procedures (as in the histogram example). The order of the items in the Calls menu tells you the order in which the procedures were called; the item at the top of the list was called by the second item, the second item was called by the third item, the third item was called by the fourth item, etc. The first menu item is the one that was most recently executed.

When a procedure crashes, the Calls menu gives you the sequence of procedures that led to the crash.

Suppose the program you write to draw histograms (described earlier in this section) crashes in the `CalcHistoWidth` procedure. The Calls menu can help you continue the program or recreate the conditions that led to the crash.

To continue the program through the rest of the `CalcHistoWidth` procedure, follow these steps:

1. Use the Immediate window to change any variables necessary to allow the program to run (that is, correct the error that caused the program to crash).
2. Choose `CalcBarWidth` from the Calls menu and press ENTER. This places the `CalcBarWidth` procedure in the View window, with the cursor on the line just after the line that called `CalcHistoWidth`.
3. Press F7. This runs the program from the current execution point (the statement where the program crashed) to the current cursor location (just after the call to `CalcHistoWidth`). The remaining code is executed in the `CalcHistoWidth` procedure and stops executing at the cursor in `CalcBarWidth`.
4. Use the Watch window, Immediate window, or Instant Watch command to verify that the variables have the values you desire, or use the Output Screen command to check any screen changes. Either way, you can verify that the program—with the changes you made in step 1—now runs properly.

To recreate the conditions that led to the crash:

1. Move the cursor to the beginning of the program and choose the Set Next Statement command on the Debug menu. This prepares the program to run.
2. Choose `CalcBarWidth` in the Calls menu and press ENTER. QuickBASIC displays the `CalcBarWidth` procedure, with the cursor on the line below the call to `CalcHistoWidth`. Move the cursor to the line above the call to `CalcHistoWidth`.
3. Press F7. This runs the program from the current execution point (the first executable statement in the program) to the current cursor location.
4. Single step (press F8) from `CalcBarWidth` into `CalcHistoWidth` until you find the error-causing statement.
5. Use the Instant Watch command or other debugging techniques to resolve the error.

In complex programs, using the Calls menu can simplify debugging.

19.2 Active Procedures

The procedure at the top of the Calls menu is the currently active procedure. The one directly below it is the procedure that called the active procedure (and the one to which control returns when QuickBASIC leaves the active procedure). You can execute your program to any procedure in the Calls menu by choosing the procedure's name and pressing F7.

Consider the procedures shown in Figure 19.1. Since `CalcBarWidth` is highlighted, pressing ENTER and then F7 executes code from the beginning of the

program to the statement in `CalcBarWidth` that follows the call to `CalcHistoWidth`. Note that QuickBASIC tracks only the most recent execution with the Calls menu, so that `DrawBars` would now be at the top of the list and `CalcBarWidth` and `CalcHistoWidth` would be removed (they were not involved in the most recent program execution).

The Calls menu is a "stack" in that it displays the most recently called procedure on the top of the list. The Calls menu displays only the eight most recent procedures; any deeper nesting is not displayed until you start executing to return through the sequence of procedure calls.

You can experiment with the characteristics of the call stack using the Fibonacci example at the end of Section 18.1, "Add Watch Command." 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 (either by using a watchpoint or pressing CTRL+BREAK), you can display the Calls menu and use the methods described above to see the different iterations of the `Fib` procedure. Since the procedure is recursive, all the procedure names are identical. Each one listed represents another instance of the call.

CHAPTER 20

The Options Menu

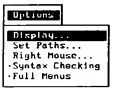
The Options menu controls some of QuickBASIC's special features, including those that customize the QuickBASIC screen, set default search paths, determine the function for the right mouse button, toggle syntax checking off and on, and switch between Full Menus and Easy Menus.

The Option menu commands are listed below:

- Display. Customizes the appearance of the QuickBASIC environment (colors, scroll bars, tab stops).
- Set Paths. Sets default search paths depending on file type.
- Right Mouse. Changes effect of clicking the right mouse button (Full Menus only).
- Syntax Checking. Toggles syntax checking on and off (Full Menus only).
- Full Menus. Toggles Full Menus on and off.

When you alter the QuickBASIC environment, either by starting QuickBASIC with a command-line option (described in Section 10.1.1, "The QB Command") or by using the Options menu, QuickBASIC saves your modifications in a file that is called QB.INI. If the file already exists when you make changes, it is updated; if it does not exist, QuickBASIC creates it. QuickBASIC uses its default settings in the absence of the QB.INI file.

20.1 Display Command



When you choose the Display command, the Display dialog box appears. It allows you to set the background color, foreground color, and the number of spaces the TAB key advances the cursor in the View window. The dialog box also lets you decide whether or not scroll bars appear in the View window. See Figure 20.1.

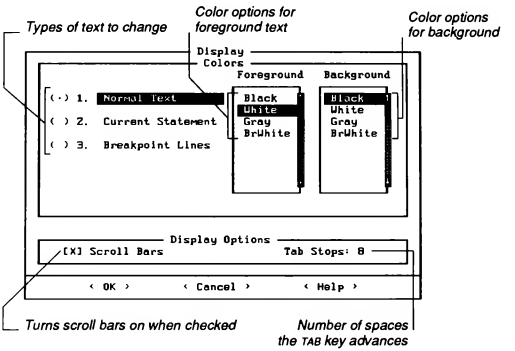


Figure 20.1 Display Dialog Box

Follow these steps to change the background and foreground colors:

1. Move the cursor to the parentheses preceding the color setting you want to change (Normal Text, Current Statement, or Breakpoint Lines).
2. Press TAB. The cursor moves to the Foreground Color list box. Use the DIRECTION keys to select the new foreground color for the item you chose in step 1. The list box item you have selected changes to reflect the colors you choose.

3. Press TAB again to move to the Background Color list box, then repeat the color selection process in step 2 for the background color.
4. Press ENTER.

QuickBASIC now reflects your changes.

The Display Options portion of the Display dialog box controls the scroll bar and TAB key settings. You can turn the View window's scroll bars on and off by pressing the SPACEBAR when your cursor is in the Scroll Bars check box (the default setting is "on"). The Tab Stops option controls the number of spaces the TAB key advances the cursor. Use the setting you find most helpful for improving the readability of your programs.

20.2 Set Paths Command

The Set Paths command changes the default search paths for specified types of files.

You may want to keep all files of a particular type within a single directory for organizational purposes. For example, you might wish to keep all of QuickBASIC's libraries and any of your own Quick libraries within a specific subdirectory or subdirectories of your QuickBASIC disk or directory. Since the environment requires access to its libraries, you would ordinarily need to provide a path name for each library used.

The Set Paths dialog box lets you type in a search path (up to 127 characters) for your files. This saves you from fully qualifying each path name every time you access a library or other files not in the current working directory.

For example, suppose you keep all of your include files and programs in a directory named `QB_BI`, which is in a directory named `QB_45` on a disk in your `A` drive. You would take the following steps to direct QuickBASIC to automatically look in that directory any time it searched for a file with the .BI or .BAS extension:

1. Choose the Set Paths command on the Options menu. A dialog box appears.
2. Press the TAB key to move the cursor to the Include files text box.

3. Type in the path name:

   ```
   A: \QB_45\QB_BI
   ```

   See Figure 20.2.

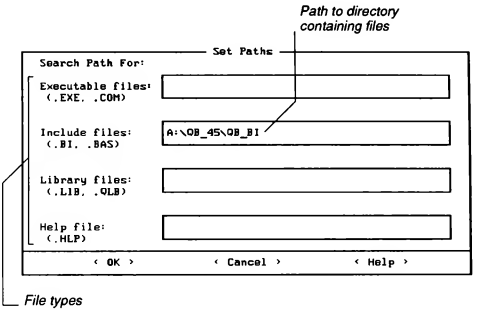


Figure 20.2 Set Paths Dialog Box

4. Press ENTER.

NOTE *The search paths you enter in the Set Paths dialog box do not change paths or environment variables previously set at the DOS level with the DOS PATH and SET commands. However, paths set with Set Paths are in effect whenever you are using QuickBASIC.*

20.3 Right Mouse Command (Full Menus Only)

The Right Mouse command lets you change the effect of clicking the right mouse button.

When you choose the Right Mouse command, the dialog box in Figure 20.3 appears.

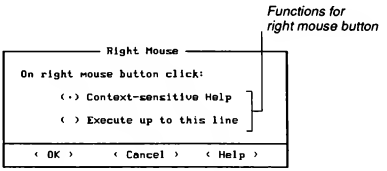


Figure 20.3 Right Mouse Dialog Box

The Right Mouse dialog box lets you select one of two functions for the right mouse button:

- Invoking on-line help (same function as the Topic command on the Help menu). This is the default setting.
- Executing the program from the beginning of the program to the current cursor location.

Both functions have advantages in different situations. You may find the Help function more useful when you are writing your program, especially if you are new to QuickBASIC. During debugging, you may find the execution feature more valuable.

If you use the Context-sensitive Help option, remember that the subject of the help is the word containing the cursor.

20.4 Syntax Checking Command(Full Menus Only)

The Syntax Checking command on the Options menu toggles automatic syntax checking on and off.

QuickBASIC defaults to the "on" setting, indicated by a bullet (•) next to the command. To turn syntax checking off, choose the Syntax Checking command; the bullet disappears. QuickBASIC records your options when you quit. If you

exit QuickBASIC with syntax-checking off, it will remain off the next time you start QuickBASIC.

When on, this feature checks the syntax of each line when it is entered.

See Also

Section 12.5.2, "Automatic Syntax Checking"

20.5 Full Menus Command

The Full Menus command toggles QuickBASIC's extended menus on and off. When Full Menus is toggled on, a bullet (•) appears next to the command.

You can use QuickBASIC's menus in either their "easy" or "full" form. In their easy form, the menus contain all of the functions necessary to create beginning to intermediate-level programs. Some advanced features do not appear under Easy Menus in order to simplify the use of the environment for programmers new to QuickBASIC. Full Menus include an additional item (the Calls menu) and more commands under each of the other menus. Once you are comfortable with QuickBASIC and need advanced features, use the Full Menus option.

When you exit, QuickBASIC saves all of the options you have selected, including Display options, Search Paths, Syntax Checking, and Full Menus. When you next return to QuickBASIC your choices will still be active.

CHAPTER 21

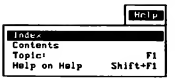
The Help Menu

Choosing commands on the Help menu is one way to access QuickBASIC's on-line help. (For information on other methods of accessing on-line help, see Section 10.8, "Using Help.")

The Help menu contains the following commands:

- Index. Displays an alphabetical list of QuickBASIC keywords and a brief description of each.
- Contents. Displays a visual outline of the on-line help contents.
- Topic. Provides syntax and usage information on specific variables and QuickBASIC keywords.
- Help On Help. Describes how to use on-line help and common keyboard shortcuts.

21.1 Index Command



The Index command displays an alphabetical list of the QuickBASIC keywords. To receive information on a particular keyword, move the cursor to the line containing that keyword and press F1. QuickBASIC displays the Help window for that keyword.

You can scroll through the keyword list or jump directly to a specific letter in the list. For example, if you type G, QuickBASIC moves the cursor to GET, the first keyword beginning with the letter "G." See Figure 21.1.

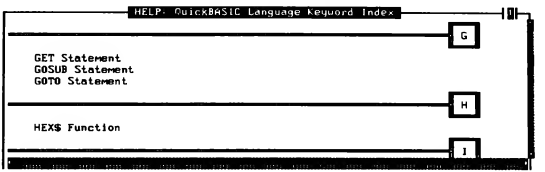
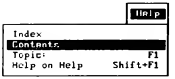


Figure 21.1 Index Entries for Help

See Also

Section 21.3, "Topic Command"

21.2 Contents Command



The Contents command displays a table of contents for on-line help. It provides hints on using the QuickBASIC environment and the BASIC language, along with useful tables. From here you can transfer to the subject that interests you. See Figure 21.2.

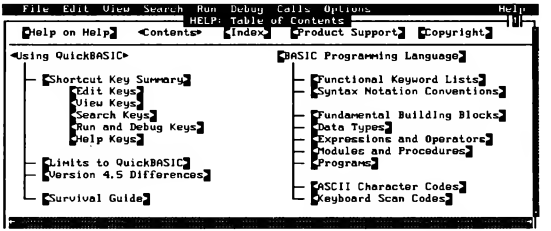
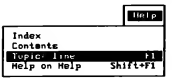


Figure 21.2 Table of Contents for Help

For detailed information on a particular topic, place your cursor on the topic and press F1. QuickBASIC displays on-line help on that topic.

For example, the Table of Contents window shows several entries under "Using QuickBASIC." If you move the cursor to "Editing keys" and press F1, a screen appears with the most frequently used editing shortcut keys.

21.3 Topic Command



The Topic command provides specific information on the syntax and usage of the QuickBASIC variable, symbol, keyword, menu, command, or dialog box at or containing the cursor. When the Topic command is active, the subject on which it will provide on-line help appears after the word `Topic`. Choosing the Topic command is the same as pressing F1.

You can get help on any keyword or variable at any time by typing the word or name and immediately choosing the Topic command.

If you choose the Topic command whenever you encounter a syntax error, you can immediately correct the usage of your statement. For example, suppose you enter the line below, which is missing the word `TO` and instead contains a comma:

```
FOR i% = 1,10
```

QuickBASIC flags the syntax error with the message `Expected: TO`. Press F1 or the Help button for help on the error itself. For information on the keyword `FOR`, move the cursor to the `FOR` statement and choose the Topic command. QuickBASIC keeps your program code visible at the bottom of the screen. See Figure 21.3.

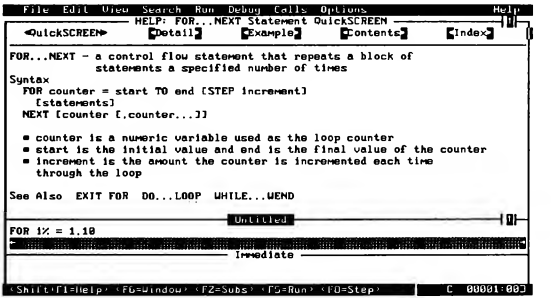


Figure 21.3 Help on the FOR...NEXT Statement

To copy sample code from the FOR help screen into your program, do the following:

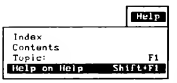
1. Choose the Example hyperlink at the top of the window.
2. Select the example code and choose Copy from the Edit menu.
3. Press ESC to close the Help window.
4. Move the cursor to the place in your program you want the text to appear, and choose Paste from the Edit menu.

You can customize the pasted code or use it as it is.

Shortcut Key

F1

21.4 Help On Help Command



The Help on Help command describes how to use the QuickBASIC on-line help system.

The Help on Help screen gives a description of the main help features. For more specific information, press ESC to exit Help on Help, and use the Topic command on the Help menu or press F1.

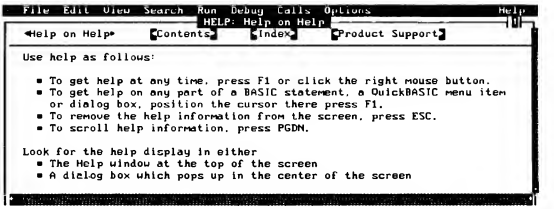


Figure 21.4 Help on Help Screen

Shortcut Key

SHIFT+F1

The purpose of this glossary is to define unique terms used in all the manuals that accompany QuickBASIC Version 4.5. Neither the individual definitions nor the list of terms is intended to be a comprehensive dictionary of computer-science terminology.

8087, 80287, or 80387 coprocessor Intel® hardware products dedicated to very fast number processing.

absolute coordinates Horizontal and vertical distances from the screen's upper-left corner, which has the coordinates (0,0). Absolute coordinates are measured in pixels. See also "physical coordinates."

address A location in memory.

algorithm A step-by-step procedure for solving a problem or computing a result.

animation 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, procedure, or program. In the following example, the string `"Hello"` is an argument to the BASIC 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. Use the COMMAND$ function to make DOS command line arguments available to the programs.

arithmetic conversion Converting numbers between integral and floating-point types.

array A sequentially numbered set of elements with the same type.

array bounds The upper and lower limits of the dimensions of an array.

ASCII (American Standard Code for Information Interchange) A set of 256 codes that many computers use to represent letters, digits, special characters, and other symbols. Only the first 128 of these codes are standardized; the remaining 128 are special characters that are defined by the computer manufacturer.

aspect ratio The ratio of a video screen's width to its height. This term is also frequently used to describe pixels. If you think of a pixel as being rectangular, its aspect ratio would be the ratio of its width to height. A typical IBM PC display has an aspect ratio of about 4:3.

assignment The process of transferring a value to a variable name. The statement `X = 3` assigns *x* a value of 3.

attributes Color, intensity, blinking, and other displayed characteristics of characters or pixels.

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.

base name The part of a file name before the extension. For example, SAMPLE is the base name of the file SAMPLE.BAS.

binary A mathematical term referring to the base-two numbering system, which uses only two digits: 0 and 1.

binary file In input/output operations, files or data that contain nonprinting characters as well as ordinary ASCII characters. Binary file access is the ability to access any byte within a file, treating it as a numeric value rather than as text.

BIOS (Basic Input/Output System) Machine-language programs built into a computer. These programs provide the lowest level of operations in a computer system. You can use the BIOS programs directly with the **INTERRUPT** and **CALL INTERRUPTX** statements.

bit The smallest unit of information used with computers; corresponds to a binary digit (either 0 or 1). Eight bits make up one byte.

block A sequence of declarations, definitions, and statements.

breakpoint A specified location where program execution will halt during debugging. See "watchpoint" for a description of variable breakpoints.

byte The unit of measure used for computer memory and data storage. One byte contains eight bits and can store one character (a letter, number, punctuation mark, or other symbol).

call by reference See "pass by reference."

call by value See "pass by value."

CGA (Color Graphics Adapter) A video adapter capable of displaying text characters or graphics pixels. Color can also be displayed with the appropriate display monitor.

character string A group of characters enclosed in quotation marks (" ").

check box A pair of brackets on a QuickBASIC dialog box where you 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 using a command button from a dialog box or an item from a menu.

click To press and release one of the mouse buttons (usually while pointing the mouse cursor at an object on the screen).

Clipboard A storage area that holds copied or cut text. Text does not accumulate in the Clipboard: new text added to the Clipboard replaces any text that was already there. You can use the Paste command to insert text from the Clipboard into a file.

communications port See "port."

compile To translate statements into a form that the computer can execute.

compiler A program that translates statements into a language understood by the computer.

constant A value that does not change during program execution.

current directory The directory in which DOS looks for files unless otherwise instructed.

current drive The drive containing the disk on which DOS looks for a directory or file unless otherwise instructed.

cursor The blinking line or box on the screen. The cursor indicates the current location where text from the keyboard will appear.

data The numbers and text processed by a computer in doing its work.

data file A file that contains the data used or generated by a program; data can be numbers, text, or a combination of the two.

debugger A program that helps to locate the source of problems discovered during run-time testing of a program.

declaration A construct that associates the name and characteristics of a variable, function, or type.

DEF FN function The object defined by a single-line **DEF FN** statement or by a **DEF FN...END DEF** block. (A **FUNCTION** procedure is generally preferred to a **DEF FN** function because a procedure is more portable and modular.)

default A setting that is assumed unless you specify otherwise.

default data type The type that BASIC uses for data unless you use a **DEF***type* statement to specify otherwise. The default type for numeric data is single precision.

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**).

device A piece of computer equipment, such as a display or a printer, that performs a specific task.

device name The name by which DOS refers to a device (for example, PRN, LPT1, LPT2, or LPT3 for a printer). Device names are treated like file names by DOS.

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.

DIRECTION keys A set of keys, found on many computer keyboards, that move the cursor left, right, up, or down on the screen. These keys usually have arrows on their key tops.

directory The index of files that DOS maintains on a disk.

document A way of classifying a file as text 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 click To press the mouse button twice quickly (usually while pointing the mouse cursor at an object on the screen).

double precision A real (floating-point) value that occupies eight bytes of memory. Double-precision values are accurate to 15 or 16 digits.

drag Pointing the mouse cursor at an object on the screen and pressing a mouse button, then moving the mouse while holding the button down. The object moves in the direction of the mouse movement. When the item is where you want it, release the button.

dynamic array An array whose memory location is allocated when the program runs. The locations are not predetermined and can change during the course of program execution.

EGA (Enhanced Graphics Adapter) A video adapter capable of displaying in all the modes of the color graphics adapter (CGA) and many additional modes. The EGA supports both text and graphics at medium resolution in up to 64 colors.

emulator A floating-point math package that uses software to perform the same operations as a hardware math coprocessor.

enabled event An event for which QuickBASIC actively checks. Events are enabled by the ON *event* statements.

event An action the computer can be programmed to detect. Events include items such as mouse clicks, menu selections, keyboard activity, and time passage.

event trapping To detect and respond to events, for example, to branch to an appropriate routine.

executable file A file with the extension .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.

exponentiation The raising of a number x to a power y; the number x is multiplied by itself y times. For example, 2^3 is `2 * 2 * 2`, or `2` raised to the power of `3`.

expression A combination of operands and operators that yields a single value.

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 (a 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.

file buffer An area in memory in which temporary data are stored. QuickBASIC keeps a copy of the file in a buffer and changes it as you edit. This buffer is copied to disk when you save the file.

filespec The complete specification of a file. The filespec can include a drive letter, path name, file name, and extension.

flags register A register containing information about the status of the central processing unit and the last operation performed.

floating point Notation, similar to scientific notation, that is a convenient method for expressing very large numbers, 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."

formal parameters Variables that receive the arguments passed to a function or subprogram. See "parameter."

FUNCTION A block of statements beginning with **FUNCTION** and ending with **END FUNCTION** that defines a procedure. **FUNCTION** procedures can be used instead of the **DEF FN** functions used in older forms of BASIC.

function declaration A declaration that establishes the name and type of a function. The function itself is defined explicitly elsewhere in the program.

function definition A statement block that specifies a function's name, its formal parameters, and the declarations and statements that define what it does (i.e., carry out the function).

global constant A constant available throughout a program. Symbolic constants defined in the module-level code are global constants.

global symbol A symbol available throughout a program. Function names are always global symbols. See also "symbol" and "local symbol."

global variable A variable available throughout a program. 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.

hexadecimal The base-16 numbering system whose digits are 0 through F (the letters A through F represent the digits 10 through 15). Hexadecimal numbers are often used in computer programming because they are easily converted to and from binary, the base-2 numbering system the computer itself uses.

highlight A reverse-video area in a text box, window, or menu marking the current command chosen or the text that has been selected for copying or deleting.

hyperlink A link between two related pieces of information. Hyperlinks appear between two triangles in help screens.

I/O redirection See "redirect."

IEEE format An international standard method of representing floating-point numbers in the computer's hardware. 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 statements and execute them immediately.

Include file A text file that is translated into a program with the $INCLUDE metacommand.

Input/output (I/O) A term that refers to the devices and processes involved in the computer's reading (input) and writing (output) data.

Input The data that a program reads.

Insert mode A mode for inputting data to the QuickBASIC environment. In this mode, you can insert new characters rather than replace characters already in the file as they are entered.

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."

Interface The boundary between two systems or entities, such as a disk drive and the computer, or the user and a program.

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. By comparison, a compiler translates all statements prior to executing any.

Keyword The name of a BASIC statement, function, or operator. Examples of keywords are **MOD**, **OPEN**, **OR**, **PRINT**, and **SIN**.

Label An identifying string that marks, precedes, or denotes a specific location in a program. Labels are used by **GOTO** statements.

Link To resolve references among the modules in a program and from the program to library routines. The link step results in a complete program ready for execution.

List box A box that lists a series of items. For example, the box that lists files in the Open Program dialog box is a list box. See also "dialog box."

Local constant A constant whose scope is limited to a particular unit of code, such as the module-level code, or a procedure within a module. All constants defined in **CONST** statements inside a procedure are local constants. See "global constant."

Local symbol A symbol that has value only within a particular function. A function argument or a variable declared as static within a function can be a local symbol. See also "global symbol."

Local variable A variable whose scope is confined to a particular unit of code, such as the module-level code, or a procedure within a module. 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 "global constant."

Logical device A symbolic name for a device that the user can cause to be mapped to any physical (actual) device.

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."

LPT1, LPT2, LPT3 Abbreviations for line printer. The names refer to the three ports to which parallel printers can be attached.

main module The module containing the first executable statement (the program's entry point). See "module."

math coprocessor An optional hardware component, such as an 8087, 80287, or 80387 chip, that improves the speed of floating-point arithmetic. QuickBASIC automatically takes advantage of a math coprocessor if it is present, or emulates it in software if it is not. See "floating point."

MCGA (Multicolor Graphics Array) A printed-circuit card that controls the display. The MCGA card, located in the system unit of a computer, shows both text and graphics at low to medium resolution in up to 256 colors.

MDA (Monochrome Display Adapter) A printed-circuit card that controls the display. The MDA shows text only at medium resolution in one color.

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 translate statements found in a specified file. See "include file."

Microsoft Binary format A method of representing floating-point numbers internally. Microsoft Binary format is used by versions of QuickBASIC prior to Version 3.0. See "floating point" and "IEEE format."

modular programming An approach to programming in which the program is divided into functional blocks, each performing a specific task.

module A section of program code that is stored in a separate file and can be separately compiled. Every program has at least one module (the main module).

module-level code Program statements within any module that are outside a **FUNCTION** or **SUB** 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 move the mouse cursor in a corresponding direction on the screen.

mouse cursor The reverse-video rectangle that moves to indicate the current position of the mouse. The mouse cursor appears only if a mouse is installed.

multidimensional array An array of with more than one subscript, for example, `x(5,35)`. A multidimensional array is sometimes referred to as an array of arrays.

near address A memory location specified by using only the offset from the start of the segment. A near address requires only two bytes of memory. See also "far address."

null character The ASCII character encoded as the value 0.

object file A file (with the extension .OBJ) containing relocatable machine code produced by compiling a program and used by the linker to generate an executable file.

offset The number of bytes from the beginning of a segment in memory to a particular byte in that segment.

operand A constant or variable value that is manipulated in an expression.

operator One or more symbols that specify how the operand or operands of an expression are manipulated.

output The result of a program's processing, usually based on some form of input data.

output screen The screen where output appears when you run a program in the QuickBASIC environment. The output screen is the same as it would be if you ran the debugged program outside of QuickBASIC.

overflow An error condition that occurs when the value assigned to a numeric variable falls outside the allowable range for that variable's type.

overlay Part of a program that is read into memory from disk only if and when it is needed.

palette The displayable colors for a given video mode. The CGA mode operates with a set of predetermined palette colors. The EGA, MCGA, and VGA color modes operate with a redefinable palette of colors.

parallel port The port to which the printer is usually attached.

parameter A variable symbol in a **DEF FN**, **FUNCTION**, or **SUB** statement. Parameters are replaced by actual variables and values when you invoke the function or subprogram.

pass by reference To transfer 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.

pass by value To transfer the value (rather than the address) of an argument to a **SUB** or **FUNCTION**.

path name The complete specification that gives the location of a file. A path name usually includes a drive letter followed by a colon, one or more directory names, and a file name.

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. Units are pixels. See also "absolute coordinates" and "view coordinates."

place marker A location in the program code to which you can return by pressing a control code.

pointer See "mouse cursor."

port An electrical connection through which the computer sends and receives data to and from devices or other computers.

precedence The relative position of an operator in the hierarchy that determines the order in which expressions are evaluated.

procedure A general term for a SUB or FUNCTION. See "FUNCTION" and "SUB."

procedure call A call to a SUB or FUNCTION.

program One or more modules linked together to form a complete set of executable instructions.

program step To highlight and execute the next instruction.

prompt A request displayed by the computer for you to provide some information or perform an action.

QB Advisor The detailed, context-sensitive on-line help on the BASIC language.

Quick library A file produced with the Make Library command or with the /Q option of the LINK command. A Quick library contains procedures for use in the QuickBASIC environment as the program is developed. Quick libraries were known in previous versions of QuickBASIC as user libraries; however, user libraries cannot be used as Quick libraries, and vice versa.

random-access file A file that can be read or written in any order. See "sequential file."

read-only file A file whose read-only attribute is set so that its contents can be displayed and read, but not changed.

record A series of fields treated as a meaningful unit in a data file or a variable declared as having a user-defined type. See "user-defined type."

recursion The practice of having a subprogram call itself; some algorithms can be coded quickly and efficiently by using recursion.

redirect To specify a device (other than the default device) from which a program will receive input or to which it will send output. Normally program input comes from the keyboard, and program output goes to the screen.

reference bar The line on the bottom of the QuickBASIC environment screen containing frequently used commands and keyboard shortcuts.

relational database A database in which any field or record can be associated with any other field or record.

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 register."

root directory The main directory that DOS creates on each disk. The root directory is the top directory in a hierarchical file system.

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 library A file containing the routines required to implement certain language functions.

run-time module A module containing most of the routines needed to implement a language. In QuickBASIC, the run-time module is an executable file named BRUN45.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, to see parts that cannot fit within a single 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.

segment An area of memory, less than or equal to 64K, containing code or data.

select To pick items, such as option buttons, in a dialog box. This verb also describes the action of highlighting areas of text or graphics to edit.

sequential file A file that can be read or written only in order from first byte to last. See "random-access file."

serial port The communications port (COM1, COM2, COM3, or COM4) to which devices, such as a modem or a serial printer, can be attached.

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.

smart editor Collective term for QuickBASIC's advanced editing features, including syntax checking, automatic keyword capitalization, and automatic procedure declaration.

source file A text file containing BASIC statements. Although all modules are source files, not all source files are modules (include files, for example).

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. Source files saved as text files (ASCII files) can be read or changed by any text editor.

stack An area of random-access memory used by BASIC to store data (e.g., information), such as intermediate results of some arithmetic computations, automatic variables in procedures, and procedure invocation information.

stand-alone file An executable file that can be run without a run-time file—a compiled program linked with BCOM45.LIB.

standard input The device from which a program reads its input unless the input is redirected. In normal DOS operation, standard input is the keyboard.

standard output The device to which a program sends its output unless redirected. In normal DOS operation, standard output is the display.

statement A combination of one or more BASIC keywords, variables, and operators that the compiler or interpreter treats as a single unit. 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 entire multiline compound statements 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 array An array whose memory location is allocated when the program is compiled. The memory location of a static array does not change while the program runs.

stop bit A signal used in serial communications that marks the end of a character.

string A 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.

SUB A unit of code delimited by the **SUB** and **END SUB** statements. A **SUB** (usually called a SUB procedure) 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. (Also called a **GOSUB** block.)

symbol A name that identifies a location in memory.

symbolic constant A constant represented by a symbol rather than the literal constant itself. Symbolic constants are defined using the **CONST** statement.

text Ordinary, printable characters, including the uppercase and lowercase letters of the alphabet, the numerals 0 through 9, and punctuation marks.

text box A box in which you type information needed to carry out a command. A text box appears within a dialog box.

text editor A program that you use to create or change text files.

text file A file containing text only.

tiling The filling of a figure with a pattern (instead of a solid color) by using the graphics statement **PAINT**.

title bar A bar, located at the top of a View window, that shows the name of the text file or 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 toggles tracing on or off.

two's complement A type of base-2 notation used to represent positive and negative in which negative values are formed by complementing all bits and adding 1 to the results.

type The numeric format of a variable. Types include integer, long integer, single precision, and double precision. Types may also be user-defined.

type checking An operation in which the compiler verifies that the operands of an operator are valid or that the actual arguments in a function call are of the same types as the corresponding formal parameters in the function definition.

type declaration A declaration that defines variable names as members of a particular data structure or type.

underflow An error condition caused when a calculation produces a result too small for the computer to handle internally.

unresolved external A procedure declared in one module, but not defined 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 procedure 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.

variable A value that can—and usually does—change during program execution.

view coordinates Pairs of numbers (for example, *x* and *y*) that uniquely define any point in two-dimensional space. The numbers *x* and *y* can be any real numbers. View coordinates can be mapped to the screen with the graphics **WINDOW** statement. View coordinates let you plot data using coordinates appropriate to the graph or picture without worrying about the actual screen coordinates. See "physical coordinates."

View window The main editing window and the window into which programs load.

VGA (Video Graphics Array) A printed-circuit card that controls the display. A VGA card shows both text and graphics at medium to high resolution in up to 256 colors.

Watch window A window that displays information about variables or expressions that you are monitoring during program execution. For each watched item, the Watch window displays the item's originating location, 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 execution location. An item is watchable only during execution of the part of the program from which you entered the item in the Watch window.

watchpoint An expression (such as `X = 3`) that stops program execution when it becomes true. Watchpoints can slow program execution. See "breakpoint."

wild card One of the DOS characters (? and *) that can be expanded into one or more characters in file-name references.

window 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. The term window also refers to the portion of a view coordinate system mapped to the physical coordinates by the **WINDOW** graphics statement. See "Immediate window," "View window,"and "Watch window."

. (dot) beside command, 14
... (ellipsis dots) with menu item, 15
80287, 279
80387, 279
8087, 279

A

Absolute coordinates, 279
Active window
  changing, 164
  changing size, 30
  described, 23
  line and column counters, 155
  switching, 164
Add Watch command, 33, 130, 248
Address
  defined, 279
  near, 285
Advisor. *See* QB Advisor, 177
Algorithms, 279
ALT key
  described, 156
  dialog boxes, use in, 16
  entering high-order ASCII characters, 202
  menus and commands, 11
Altering text, 227
AND operator, 60
Animation, 279
Apostrophe ('), xxx
Arguments, 279
Arithmetic
  conversion, 279
  floating point, 238
  overflow, 238
Array bounds
  checking, 237
  defined, 279
Array elements, specifying, 56
Array variables, 56
Arrays
  defined, 279
  dynamic, 282
  multidimensional, 285
  static, 289
Arrow keys. *See* DIRECTION keys
ASCII characters
  corresponding to control sequences, 202
  entering high-order, 202
  set, 278–279
Aspect ratio, 279
Assignment, 279
Assignment statement, 45
Attributes, 279
Automatic formatting, 202
Automatic syntax checking, 198, 273

B

/B option (QB), 150
Bar, menu, 285
Base name, 240, 280
BASIC command-line compiler. *See* BC
BASICA programs, running with QuickBASIC, 232
BC, 234
BCOM45.LIB, 234–235
BIOS, 280
Bit, 280
Black-and-white monitor. *See* Monitor, monochrome
Block, 280
Boolean expressions
  compound, 60
  precedence of evaluation, 61
  simple, 59
Branching, 61
Break on Errors command, 250
Breaking out of programs, 113
Breakpoints
  defined, 280
  deleting, 130
  described, 248
  setting, 130, 137
Breaks, checking at run time, 238
BRUN45.EXE, 237
BRUN45.LIB, 234–235
Buffer, 282
Bugs. *See* Debugging; Errors
Buttons, right mouse, 272
Byte, 280

C

/C option (QB), 151
Calculations
  floating point, 238
  Immediate window, 167
Call by reference. *See* Passing by reference
Call by value. *See* Passing by value
Calls menu, 265

Calls, procedure, 287
Canceling commands, 12
Capitalization conventions, smart editor, 89
CAPS LOCK/NUM LOCK indicators
  described, 10
  illus., 154
Case sensitivity, 88
CGA. *See* Color Graphics Adapter
CHAIN statement, 237
Change command
  described, 227
  dialog box, illus., 103, 227
Changing
  active window, 164
  default search paths, 271
  main modules, 242
  programs, 170
  screen display, 16
  text, 227
  values of variables, 33
  window size, 30, 164, 218
Character strings, 280
Check boxes
  defined, 280
  described, 17, 161
  illus., 17, 160
Checking
  syntax, 90, 198, 273
  types, 290
Choosing
  commands
    ENTER key, 156
    Easy Menus, 14
    keyboard, 156
    menus, 11, 14, 155
    mouse, 172
  defined, 280
CHR$ function, 203
Clear All Breakpoints command, 130
Clear command, 210
Clearing
  screens, 155
  View window, 28
Clicking, 280
Clipboard, 95, 207, 280
Closing Help, 174
Closing menus, ESC key, 156
/CMD option (QB), 151
Code, module level, 80, 285
Color Graphics Adapter (CGA), 280
Colors. *See* Screen colors
Column counter, illus., 154
Command buttons
  described, 16, 160–161
  Example, 107
  Help, 107
  illus., 17
  Make EXE, 236
  Make EXE and Exit, 236
  Make Library and Exit, 241
  Program, 107
  Remarks, 107
COMMAND$ function, 233
Commands
  Add Watch, 33, 130, 248
  availability, 97
  Break on Errors, 250
  canceling, 12
  Change, 103, 227
  choosing
    ENTER key, 156
    Easy Menus, 14
    keyboard, 156
    menus, 11, 14, 155
    mouse, 172
  Clear, 210
  Clear All Breakpoints, 130
  Contents, 13, 276
  Continue, 130, 135
  Copy, 97, 209
  Create File, 186
  Cut, 97, 132, 208
  debugging
    Debug menu, 130
    described, 251
    F7, 131, 250
    F8, 131, 136
    F9, 137
    F10, 131
    function keys, 129
    (table), 250, 253
  Delete Watch, 130, 140
  Display, 270
  DOS Shell, 191
  editing (table), 203
  Execute to cursor, 250
  Exit, 192
  Find, 102, 224
  Full Menus, 274
  Help, 277
  Help on Help, 13, 278
  History On, 247
  Include File, 219
  Include Lines, 221

Commands (*continued*)
Index, 13, 106, 275
Instant Watch, 110, 130
Label, 229
Load File, 188
Make EXE File
dialog box, illus., 142
introduction, 235
Make Library, 240
Merge, 183
New FUNCTION, 213
New Program, 28, 180
New SUB, 210
Next Statement, 219
Next SUB, 218
Open Program, 181
Output Screen, 29, 219
Paste, 97
Print
defined, 191
dialog box, illus., 112
Help, 111
procedure stepping, 251
QB, 150
Repeat Last Find, 226
Restart, 130
Right Mouse, 272
Save, 184
Save All, 185
Save As, 37, 185
Selected Text, 226
Set Main Module, 242
Set Next Statement, 250
Set Paths, 271
shortcut keys, 157
single stepping, 251
Split, 163, 218
Start, 29, 130
starting QuickBASIC, 150
SUBs
defined, 162, 216
moving procedures, 82, 133, 210
Syntax Checking, 198, 273
Toggle Breakpoint, 130
Topic, 13, 277
Trace On, 247
Undo, 208
Unload File, 189
Comments, appearing in QCARDS, 78
COMMON statement, 237
Communications port, 280
Compile, 280
Compiler, 281
Compiler/environment differences, 238
Constants
defined, 281
local, 284
symbolic, 289
Contents command, 13, 276
Context
debugging, 248
execution, 248
Context-sensitive help, 34
Continue command, 130, 135
Continuing suspended programs, 135
Control-sequence characters, 203
Conventions, document, xxix
Conversion, arithmetic, 279
Coordinates
absolute, 279
physical, 286
Coprocessor, 238, 285
Copy command, 97, 209
Copying
examples from help, 118
text, 205, 209
Create File command, 186
Creating
executable files, 142, 234–236
FUNCTION procedures, 213–214
libraries, 240
main modules, 241
new subprograms, 211
procedures, 131
CTRL key, entering special characters, 202
CTRL+BREAK, 238
Current directory, 281
Current drives, 281
Current Statement option (Display command), 18
Cursor
defined, 281
illus., 155
moving, 27, 204
Customizing screen display, 16, 270
Cut command, 97, 132, 208

D

Data, 281
Data types, default
changing, 212
defined, 281
for procedures, 211
Database, 77

Debug menu
commands (list), 253
illus., 253
(table), 130
Debugger
defined, 281
symbolic, 245
Debugging
breakpoints, 248
commands
debug menu, 130
F7, 131
F8, 131, 136
F9, 137
F10, 131
function keys, 129
Run menu, 130
context, illus., 170
Continue command, 233
features, 247
Immediate window, 246
introduction, 245
Next Statement command, 219
procedures, 134, 245, 247, 249, 251
watch variables, 248
Watch window, 170
while writing code, 246
Debugging features (table), 250
Decision-making, 57
Declarations
defined, 281
procedures, automatic, 141
DECLARE statement, procedure declarations, 213
DEF FN function
defined, 281
Watch window, 249
Default, 281
Default data types
changing in procedures, 212
for procedures, 211
Default search paths, 271
DEF*type*
defined, 281
introduction, 211
Delete Watch command
closing Watch window, 140
deleting variables, 130
Deleting
breakpoints, 130
modules, 189
text, command summary, 204
variables from Watch Window, 130
Details command button, 107
Devices
defined, 281
logical, 284
names, 281
Dialog boxes
clearing, 155
defined, 15, 281
illus., 159–160
Merge command, 183
Print command, 191
selecting items, 161
setting options, 16–17
text box, 160
DIM statement, 237
Dimensions, 281
DIRECTION keys
defined, 281
use with menus, 11, 176
Directories
changing from within QuickBASIC, 183
changing search paths, 271
current, 281
defined, 281
listing, 182
root, 287
Display command, 270
Division, 47
DO statement, Help, illus., 116
DO...LOOP
defined, 65
terminating condition, 65
UNTIL version, 67
WHILE version, 66
Document, 282
Document conventions, xxix
DOS, returning to, 191
DOS Shell command, 191
Dots, marking command, 14
Double precision, 282
Double-click, 282
Drag, 282
Drives, 281
Duplicating text. *See* Copying
Dynamic arrays, 282

E

Easy Menus
commands, choosing, 14
described, 9

Easy menus (*continued*)
vs. full, 274
windows available with, 23
Edit menu
Clear command, 210
commands (list), 207
Copy command, 209
Cut command, 208
New FUNCTION command, 213
New SUB command, 210
Syntax Checking command, 198
Undo command, 208
Editing
commands (table), 203
include files, 219
material from Help, 122
Editor
automatic formatting, 88, 202
capitalization conventions, 89
entering text, 195
selecting text, 196
syntax checking, 198
text, 289
EGA. *See* Enhanced Graphics Adapter
Ellipsis dots (...) with menu item, 15
Emptying View window, 28
Emulator, 282
Enabled events, 282
Enhanced Graphics Adapter (EGA), 282
Entering
special characters, 202
text, 195
Environment/compiler differences, 238
Error checking, executable files, 237
ERROR function, 169
Error messages
clearing, 155
help for, 94
Errors
correcting, 105
run-time
checking, 237
line numbers, 237
reporting locations. 235
simulating, 169
running programs, 92
syntax, checking, 199
typing, 199
Event trapping
defined, 282
Quick libraries, 241
Events, 282
Example code
copying from Help, 118
Help hyperlinks, 107
Example command button, 107
EXE requiring BRUN45.EXE option, 235
Executable files
creating, 234–236
defined, 282
Quick libraries, 236
stand-alone, 142, 290
Execute option (QB), 150
Execute to cursor command, 250
Executing
all statements of procedures, 131
commands. *See* Choosing commands
programs
right mouse button, 273
suspending, 108
to cursor, 111, 131
Exit command, 192
Exiting QuickBASIC, 37, 192
Exponentiation, 282
Expressions
defined, 51, 282
simple, 51
use, 51
watch, 248

F

F1 key, 34
Falsity, representation, 59
Fibonacci sequence, 255
File menu
commands (list), 179
Create File command, 186
DOS Shell command, 191
Exit command, 192
illus., 155
Load File command, 188
Merge command, 183
Open Program command, 180–181
Print command, 191
Save All command, 185
Save As command, 185
Save command, 184
Unload File command, 189
Files
changing default search paths, 271
data, 281

Files (*continued*)
executable
creating, 234–236
defined, 282
error checking in, 237
Quick libraries, 236
stand-alone, creating, 142
include
defined, 187, 284
finding, 220
nesting, 220
viewing and editing, 219
viewing only, 221
inserting in active window, 183
loading, 188
.MAK, 243
merging, 183
object, 286
printing, 191
QB.INI, 20, 269
read only, 287
saving, 184–185
sequential, 288
source, 288
stand-alone, 289
text, 187, 289
filespec, 283
Find command
described, 224
dialog box, illus., 224
options (table), 225
restrictions (table), 225
Finding
See also Searching
include files, 220
text, 224
Fixed disk based Help, 177
Flags, 283
Floating point
arithmetic, executable files, 238
defined, 283
variables, 44
Floppy disk-based help, 177
Floppy disk-based QB Advisor, 177
Flow, programs, tracing, 259
FOR...NEXT loop, 63
Formal parameters, 283
Formats
IEEE, 283
Microsoft binary, 285
Formatting, automatic, 88, 202
Full Menus, 274
Function declarations, 283
Function definitions, 283
Function keys, debugging, 129
FUNCTION procedures
creating, 213
default data type, 211
naming, 213
viewing, 216
Functions, conversion option (QB), 151

G

/G option (QB), 151
Global symbols, 283
Global variables, 283
GOSUB statement, 238

H

/H option (QB), 151
Hard disk-based help, 177
Hardware needed for running QuickBASIC, xxvi
Heap, 283
Help
command buttons, 107
constructing statements with, 105
context sensitive, 34
copying examples from, 118
copying syntax blocks from, 115
described, xxviii, 12
editing material from, 122
error messages, 94
examples, 107
files
floppy-disk system, 177
hard-disk system, 177
help with, 278
hyperlinks, 35, 174
invoking, 152
keys (table), 176
keywords
command button, 107
described, 105
illus., 107
listing with Index, 275
language, 174
missing Help files, 176
outline of system, 276
place markers in, 174
printing text from, 111
QB Advisor, 177
right mouse button, 273

Help (*continued*)
sample programs, 276
searching for Help files, 176
Set Paths command for files, 177
syntax, 34
trouble evoking, 177
using, xxv, 12, 174
Help menu
commands (list), 275
Contents command, 276
Help on Help command, 278
Index command, 275
introduction, 12
Topic command, 277
Help on Help command, 13, 278
Help window
described, 25, 34, 162
illus., 109
moving in , 176
Hexadecimal, 283
Hierarchical filing system (HFS), 283
High-order ASCII characters, 202
High-resolution-display option (QB), 151
Highlight, 283
History Back, 260, 262
History Forward, 260
History On command, 247
Hyperlinks, 35, 174–175

I

I/O. *See* Input/output
IEEE format, 283
IF...THEN...ELSE statement
defined, 61
help, illus., 106
syntax, 61
Immediate window
calculations, 167
calling procedures from, 168
capacity, 166
changing, active window, 164
context, 170
debugging in, 246, 249
defined, 284
described, 24, 162, 165
illus., 10, 154–155
limits, 33
run-time error messages, 247
run-time errors, simulating, 169
running statements in, 31
Immediate window (*continued*)
size, changing, 164
statements not supported, 167
testing programs, 167
variable values, changing, 168
Implicit links, 36
$INCLUDE metacommand, 220, 243
Include files
defined, 187, 284
finding, 220
nesting, 220
viewing and editing, 219
viewing only, 221
Included File command, 219
Included Lines command, 221
Indentation controls, 196
Indenting
program blocks, 118
text, 100, 196
Index command
described, 13, 275
screen, illus., 276
using, 106
Input
defined, 284
standard, 289
INPUT statement, 54, 238
Input/output (I/O), 284
Insert mode, 195, 284
Inserting
commands, summary, 204
text, 195
Installing QuickBASIC, 5–6
Instant Watch command
dialog box, illus., 256
symbol help, 110
Integer variables, 44
Integers, 284
Interface, 284
Interpreter, 284

K

Key combinations
See also Shortcut keys
cursor movement, 19
described, 14
scrolling, 28
Keyboard
choosing commands from, 156
moving in Help windows, 176

Keys
  control (CTRL), 281
  CTRL+BREAK, 113
  DIRECTION, 281
  F7, 111, 131
  F8, 131, 136
  F9, 137
  F10, 131
  Help (table), 176
  shortcut
    editing commands, 19
    (table), 157
Keywords
  capitalization of, 43, 202
  defined, 43, 284
  help, 105, 275, 277

L

/L option (QB), 151
Label command, 229
Labels, 284
Language help, 174
Leaving QuickBASIC. *See* Exiting QuickBASIC
Libraries
  creating, 240
  Quick
    defined, 287
    described, 236
    non-library modules, used with, 241
  run-time, 288
  stand-alone, executable files, 236
Line counter, illus., 154
Line-number checking, 237
Lines, setting attributes, 18
Linker, BRUN45.EXE, advantage of, 236
Linking, 284
Links, implicit, 36
List boxes
  defined, 284
  described, 16, 161
  illus., 17
  scroll bar, illus., 160
Listing directories, 182
Load File command, 188
Loading
  files, 188
  modules, 186, 188, 239
Local
  constants, 284
  symbols, 284
  variables, 284
Logical devices, 284
Logical operators, 60
Logical relations, 57
Long integers, 284
Loop counters, 63
Loop variables, 63
Loops
  DO, 65
  FOR...NEXT, 63
LPT1, 285

M

Main modules, 82, 241–242, 285
.MAK file, 243
Make EXE and Exit command button, 236
Make EXE command button, 236
Make EXE File command, dialog box, illus., 142, 235
Make Library and Exit command button, 241
Make Library command, 240
Math coprocessor, 285
Math functions, 49
/MBF option (QB), 151
MCGA. *See* Multicolor Graphics Array
MDA. *See* Monochrome Display Adapter
Menu bar, 10, 155
Menu-item description, illus., 10
Menus
  browsing, 11
  Calls, 265
  clearing, 155
  closing, 156
  commands, choosing, 155
  Debug, 130, 247, 253
  Easy
    described, 9
    vs. full, 274
  File
    commands (list), 179
    illus., 155
  Help, 12, 275
  opening, 11
  Options, 269
  Run, 235
  selecting, 155
  setup, 6
Merge command, 183
Metacommands
  $INCLUDE, 220, 243
  defined, 285
Microsoft Binary format, 285
Microsoft Overlay Linker (LINK), 234

Modes
insert, 195, 284
overtype, 195
Modular programming, 285
Module-level code
defined, 285
described, 80
Modules
defined, 187, 285
deleting, 189
existing, 188
loading and unloading, 239
main
changing, 242
defined, 285
new, 186
printing, 191
run-time, 288
Monitors, monochrome
starting QuickBASIC, 10
use with Color Graphics Adapter, 150
Monochrome Display Adapter (MDA), 285
Mouse
compatibility with QuickBASIC, 8
cursor, 285
defined, 285
hyperlinks, using with, 175
pointer, illus., 155
QuickBASIC, using with, 172
scrolling with, 173
setting right button, 272
Moving
procedures
Help windows, 176
SUBs command, 133
View window, 82
text, 97, 208
Multicolor Graphics Array (MCGA), 285
Multidimensional arrays, 285
Multiple-module programs, 265

N

Names
base, 280
device, 281
path, 286
Naming procedures, 213
Near address, 285
Nested procedures, 265
Nesting Include files, 220
New FUNCTION command, 213
New Program command, 28, 180
New SUB command, 210
Next Statement command, 219
Next SUB command, 218
NOT operator, 60
Not watchable message, 249
Null characters, 285
NUM LOCK indicator
described, 10
illus., 154
Numeric coprocessor. *See* Coprocessor

O

Object files
compiling with BC, 234
defined, 286
Offset, 286
On-line Help. *See* Help
Open Program command, 181
Opening
menus, 11
programs, 25
screen, illus., 155
Operand, 286
Operators
arithmetic, 47
defined, 286
logical, 60
Option buttons
described, 16, 161
illus., 17
Option boxes, illus., 160
Options, QB command, 150
Options menu
commands (list), 269
Display command, 270
Full Menus command, 274
Right Mouse command, 272
Set Paths command, 176
Syntax Checking command, 273
OR operator, 60
Organizing QuickBASIC files, 271
Output
See also PRINT statement
defined, 286
program, how displayed, 29
standard, 289
Output screen
defined, 286
viewing, 219

Output Screen command, 29, 219
Outputting
  high-order ASCII characters, 202
  low-order ASCII control sequences, 203
Overflow, 286
Overlay, 286
Overtyping text, 95, 195

P

PACKING.LST, xxix
Palette, 286
Parallel ports, 286
Parameters
  defined, 286
  formal, 283
Parentheses, used to establish precedence, 49
Passing by reference, 286
Passing by value, 286
Paste command, 97, 209
Pasting text, 99, 126, 209
Path names, 286
Path-specification line, 160
Paths, changing default search, 271
PGDOWN key, 176
PGUP key, 176
Physical coordinates, 286
Placemarkers
  defined, 286
  using, 197
Pointers, 286
Ports
  parallel, 286
  serial, 288
Precedence
  defined, 287
  introduction, 48
Preventing bugs, 245–246
Print command
  described, 191
  dialog box, illus., 112, 191
  Help, 111
PRINT statement
  combining, 54
  numeric expressions, 29
  tabbing, 53
  use, 40, 52
Problems, software, reporting, xxxi
Procedure declarations, automatic, 95, 141
Procedure stepping, 251
Procedures
  calling from Immediate window, 168
  changing default data type, 212
  creating, 131
  creating FUNCTION, 213
  creating SUB, 210
  debugging, 134
  declaring, 213
  default data types, 211
  defined, 287
  described, 80
  executing all statements of, 131
  naming, 213
  nested, 265
  sequence of, 265
  SUB, 289
  View window, illus., 132
  viewing, 216, 218
Produce Debug Code check box, 235, 238, 241
Program command button, 107
Program execution, right mouse button, 273
Program statements, 40
Program symbols. *See* Symbols
Programming environment, described, xxv
Programming, modular, 285
Programs
  BASICA, running with QuickBASIC, 232
  blocks, indenting, 118
  continuing, 263
  continuing if suspended, 135
  defined, 40, 287
  executing to cursor, 111, 131
  execution
    continuing, 130
    suspending, 113
  existing, starting, 181
  exiting, 192
  flow, tracing, 259
  instructions, 40
  multiple module, 265
  new, starting, 180
  output, 29
  resetting variables, 232
  restarting, 130, 263
  restarting at first executable statement, 131
  run-time module required with, 236
  running, 29, 130, 168
  sample, on disk, xxix
  selecting, 27
  setup, 5

Programs (*continued*)
stand-alone, 237
starting, 263
steps, 287
stopping execution, 261
structured, 79–80
suspending execution, 108
testing, 167
Prompts, 287

Q

QB Advisor, 174, 177. *See also* Help
QB command, 150–151
QB Express, xxix, 7
QB.INI file, 20, 269
QCARDS program, 77–78
Quick libraries
See also Libraries
defined, 287
file naming, 240
updating previous, 239
Quick-library loading option (QB), 151
QuickBASIC
debugging features, 247
installing, 5–6
keywords, 275
manuals, how to use, 7
mouse use, 8
options, 269
starting, 9
QuickBASIC environment, returning to, 113
QuickBASIC, Fast Load and Save option, 185
Quitting QuickBASIC, 37, 192

R

RAM space, saving with executable files, 237
Read-only files, 287
README.DOC, xxix
Records, 287
Recursion, 287
Redirection, 287
Reference bar
defined, 287
described, 11, 155
illus., 10, 154
Reference, passing by, 286
Registers, 287
Relational operators, 57
Relational statements, 57
Remote-reception buffer-size option (QB), 151
Removing
See also Deleting
modules, 239
text, 208, 210
Renaming programs, 85
Repeat Last Find command, 226
Repeat previous search. *See* Repeat Last Find command
Repeating program statements, 62
Replacing text, 102–103. *See also* Pasting text
Reserved words. *See* Keywords
Resetting variables, 232
Restart command, 130
RETURN statement, 238
RETURN without GOSUB error, cause, 238
Right Mouse command, 272
Row/column counters, 11
Run menu
debugging commands, 130
Make EXE File command, 235
Make Library command, 240
Set Main Module command, 242
/RUN option (QB), 150
Run time
defined, 288
errors
checking, 237
hint regarding, 247
reporting locations, 235
simulating, 169
Run-time libraries
defined, 288
executable files, 234
Run-time module
defined, 288
programs using, 236
Running programs
debugging commands, 130
errors in, 92
new, 29

S

Save All command, 185
Save As command
described, 185
dialog box, illus., 37, 185
Fast Load and Save option, 185
QuickBASIC option, 185
Text option, 185

Save command, 184
Saving
  files, 184–185
  .MAK files, 243
  procedures, 213
Saving programs, 85
Scope, 288
Screen
  clearing, 155
  colors, setting, 270
  display, customizing, 16
  displaying data on, 40
  illus., 10
  QuickBASIC, 152
Screen-update-speed option (QB), 151
Scroll arrow, illus., 153
Scroll bars
  defined, 288
  illus., 10, 154–155
  option (Display command), 17
  setting, 17, 270
Scroll box, 173
Scrolling
  defined, 288
  keyboard commands, 164
  keys, summary of , 205
  mouse, 173
  text in a window, 28
Search menu
  Change command, 227
  commands (list), 223
  Find command, 224
  Label command, 229
  Repeat Last Find command, 226
  Selected Text command, 226
Search Paths command, 271
Searching
  labels, 229
  procedures, 216
  text, 102, 223, 226
Segments, 288
SELECT CASE statement
  described, 122
  help screen, illus., 119
Selected Text command, 226
Selecting
  menus, 155
  programs, 27
Selecting (*continued*)
  text
    cautions, 96
    described, 96
    editing functions, 196
    removing selection, 96
    summary of, 205
Sequence of procedures, 265
Sequential files, 288
Serial ports, 288
Set Main Module command, 242
Set Next Statement command, 250
Set Paths command, 176–177, 271
Setting
  breakpoints, 130
  screen attributes, currently executing lines, 18
  screen colors, 270
  scroll bars, 270
  tab stops, 270
Setting up QuickBASIC, 5–6
Setup menu, 6
Setup program, 5
SHARED, 249
Shortcut keys
  commands, 157
  described, 155
  introduction, 19
  (table), 157
Simple variables, 288
Single precision, 288
Single stepping, 251
Smart editor
  See also Editor
  automatic procedure declarations, 95
  defined, 197, 288
  described, 87
  syntax checking, 90
Source files, 288
SPACEBAR, using in dialog boxes, 17
Special characters, methods for entering, 202
Split command, 163, 218
Splitting windows, 218
Stack
  Calls menu, 268
  defined, 288
Stand-Alone EXE File option, 235
Stand-alone files, 289. *See also* Executable files
Stand-alone libraries. *See* Libraries

Stand-alone programs, advantages, 237
Standard input and output, 289
Start command, 29, 130
Starting
  See also Loading
  programs
    at specific statement, 263
    existing, 25, 181
    new, 180
  QuickBASIC, 9, 150
Statement checking. *See* Syntax checking
Statements
  beginning execution from, 263
  CHAIN, 237
  COMMON, 237
  DECLARE, 213
  defined, 289
  DIM, 237
  DO, help, illus., 116
  ERROR, 169
  GOSUB, 238
  IF...THEN...ELSE, help, illus., 106
  INPUT, 54, 238
  next to execute, 219
  PRINT, 29
  RETURN, 238
  SELECT CASE
    described, 122
    help screen, illus., 119
  STEP, 64
  using help to construct, 105
Static arrays, 289
STEP statement, 64
Stop bits, 289
Stopping programs, 232
String variables, 45
Strings
  character, 280
  combining, 46
  defined, 41, 289
  passing to QuickBASIC, 233
Structured programming, 79–80, 196
SUB procedures
  creating, 210
  default data type, 211
  defined, 289
  naming, 213
  viewing, 216
Subprograms
  creating, 210–211
  defined, 289
Subroutines, 289
SUBs command
  described, 216
  dialog box, illus., 216
  illus., 82
  moving procedures, 82, 133, 210
Suspending program execution, 108, 113
Switching active window, 164
Symbol help, 138
Symbolic constants, 289
Symbolic debuggers, 245
Symbols
  defined, 289
  global, 283
  help, 108
  local, 284
  using, 110
Syntax
  checking, 90, 198, 273
  errors, checking, 198–199
  help, 115, 277
Syntax Checking command
  described, 198
  introduction, 273
  turning off, 199

T

Tab, 197
TAB key, dialog boxes, using in, 17–18
Tab stops, setting, 18, 196, 270
Testing, programs, 167
Text
  changing, 227
  copying, 209
  defined, 289
  deleting, 97, 210
  editors, 289
  entering, 195
  finding, 224
  indenting, 100, 196
  inserting, 195
  moving, 97, 208
  option, Save As command, 185
  overtyping, 95, 195
  pasting, 99, 126, 209
  removing, 208
  replacing, 102–103, 208
  searching for, 102, 223, 226
  selecting, 96, 196

Text boxes
  defined, 289
  described, 16, 160
  errors, correcting, 160
  illus., 17, 160
Text files, 187, 289
Tiling, 289
Title bar
  defined, 289
  described, 155
  illus., 10
Toggle Breakpoint command, 130
Toggles, 14, 290
Topic command, 13, 277
Trace On command, 247
Tracing
  backward, 260
  forward, 260
  introduction, 247
  procedure steps, 259
  program flow, 259
Tracing through Help screens, 174
Trapping events, 282
Truth, representation, 59
Tutorial, how to use, 7
Two's complement, 290
Types, 290
Typing mistakes, 199
Typographic conventions. *See* Capitalization conventions, smart editor; Document conventions

U

Underflow, 290
Undo command, 208
Unload File command, 189
Unloading modules, 239
Unresolved externals, 290
UNTIL keyword, 65
User-defined types, 290
Using help, 174

V

Value, passing by, 286
Values, changing in running programs, 168
Variables
  automatic, 280
  default type, 45
  defined, 290
  deleting from Watch window, 130
Variables (*continued*)
  displaying, 52
  floating-point, 44
  global, 283
  help, 277
  illegal assignments, 47
  initialization, 43
  integer, 44
  local, 284
  naming conventions, 42–43
  resetting, 232
  simple, 288
  string, 45
  types, 44
  values
    changing, 33, 46
    current, 130
    determining, 110
  watch, 248
Video Graphics Array (VGA), 290
View menu
  commands (list), 215
  Include File command, 219
  Included Lines command, 221
  Next Statement command, 219
  Next SUB command, 218
  Output Screen command, 219
  Split command, 163, 218
  SUBs command, 162, 216
View window
  clearing, 28
  defined, 290
  described, 24, 26, 162
  illus., 10, 155
  loading a program in, 25
  moving procedures, 82
  procedures in, 132
  programming in, 28
  splitting, 163
Viewing procedures, 216

W

Watch expressions, 248
Watch variables, debugging, 248
Watch window
  closing, 140
  commands, 247
  debugging programs, 33
  defined, 290
  deleting variables from, 130

Watch window (*continued*)
described, 25, 162, 248–249
illus., 170
monitoring variable values, 34
placing variable into, 130
watchpoints, 248
Watchable, 290
Watching expressions, in Watch window, 247
Watchpoints, 247–248, 290
WHILE keyword, 65
Windows
activating, 30
active
changing, 164
described, 23
available with Easy Menus, 23
defined, 291
Help
described, 25, 34, 162
illus., 109
moving in, 176
Immediate
calculations, 167
calling procedures from, 168
capacity, 166
debugging in, 249
defined, 284
described, 24, 162, 165
illus., 154–155
run-time errors, simulating, 169
Windows (*continued*)
Immediate (*continued*)
running statements in, 31
statements not supported, 167
testing programs, 167
variable values, changing, 168
window size, changing, 164
moving between, 30
size, changing, 30, 164, 218
splitting, 218
View
defined, 290
described, 24, 26, 162
illus., 155
loading a program, 25
procedures in, 132
programming in, 28
splitting, 163
Watch
closing, 140
commands, 247
debugging programs, 33
defined, 290
deleting variables from, 130
described, 25, 162, 170, 249
monitoring variable values, 34
placing variable into, 130
watchpoints, 248
WordStar-style command indicator, illus., 154

Common Questions

This section answers the most commonly asked questions about QuickBASIC. If you have a question about QuickBASIC, you may want to read this section before calling Microsoft Product Support Services (PSS). You'll find instructions on contacting PSS at the end of this section.

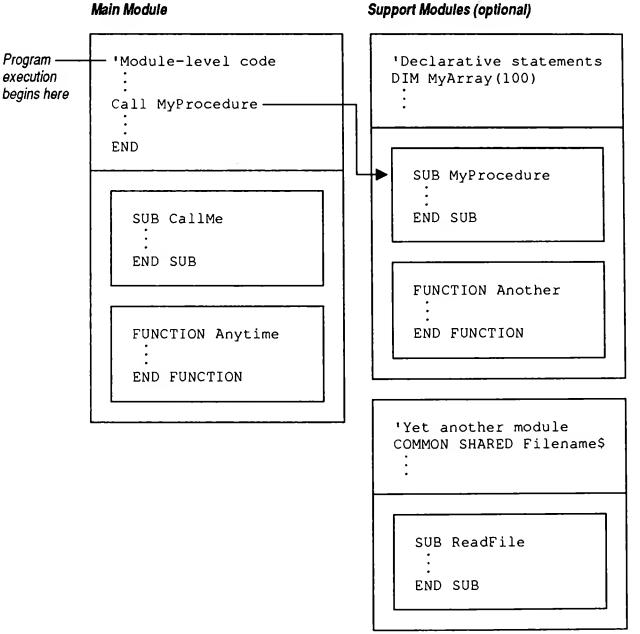
The topics covered in this section include:

- Modular programming
- String space and dynamic arrays
- Performance and optimization
- Floating-point math
- Key trapping
- Quick libraries
- Mixed-language compatibility
- Communications
- Additional readings

What is modular programming?

In a modular program, you isolate functional blocks of BASIC source code in separate **SUB** or **FUNCTION** procedures. You can also place procedures in separate .BAS files, called *modules*.

The following figure illustrates a program with multiple modules.



Every BASIC program has a main module containing the first executable statements of the program. Program execution starts at the beginning of the main module. The module-level code then calls **SUB** or **FUNCTION** procedures contained in the main module or other modules, called *support modules*. You can also call **SUB** or **FUNCTION** procedures from within any other procedure.

How can I benefit from programming with procedures and modules?

Modular programming offers several advantages:

- The logic of a modular program is clearer and easier to understand, because program code is contained in functional blocks.
- Your program code is easier to edit, since each procedure has its own editing window. You can quickly locate specific sections of your program using QuickBASIC's View Subs command.
- Debugging is greatly simplified. You can test procedures separately, and isolate potential problems before putting together your whole program.
- Your code is easy to reuse. Separate **SUB** and **FUNCTION** procedures or entire modules can be reused in other programs that perform some of the same actions. For example, you could place procedures for working with files in their own module and use them in any program that requires file operations.
- You can compile larger programs. Since QuickBASIC allows only one 64K code segment for each compiled module, you must split programs larger than this limit into separate modules before compiling.

How do I use procedures and modules?

To use procedures, follow these steps:

1. Identify each major action that your program must perform.
2. Place the code that performs each action in a separate **SUB** or **FUNCTION** procedure.
3. Write module-level code. Your program's flow of control begins with this code.
4. As you write your program, identify the tasks that belong in new **SUB** or **FUNCTION** procedures. If there are tasks that will be performed repeatedly by your program, place this code in separate procedures, and call them whenever needed.
5. If you are developing a large program, place related procedures in support modules. You can call procedures contained in any module from anywhere in your program. The module-level code of a support module, however, is not in the program's direct flow of control. It should contain only declarative statements (such as **COMMON** or **DIM**) and error- and event-handling routines.

The logic of a modular program can be clearly outlined using procedures. For example, if your program displays a menu for the user to choose from, you could write a separate procedure to handle each menu choice. The following module-level code calls the appropriate procedure based on the action the user chooses from a menu:

```
Choice% = DoMenu%          ' Call a FUNCTION procedure that draws
                           ' a menu and returns the item chosen.
SELECT CASE Choice%
   CASE 1                  ' Call the SUB procedure that performs
      CALL ReadData        ' the requested action.
   CASE 2
      CALL DisplayForm
   CASE 3
      CALL PrintReport
      .
      .
      .
   CASE 8
      END                  ' User chose Quit.
END SELECT
```

Write your procedures so that they are general enough to be used for various operations. For example, a procedure that displays a message on the screen might allow you to specify the message text and the screen location:

```
SUB CenterMessage (Text$, Row%)
   LOCATE 40 - (LEN(Text$) \ 2), Row%  ' Center the message on
   PRINT Text$;                        ' the specified row.
END SUB
```

Each time you need to display a message, you would call this procedure as follows:

```
CenterMessage "Value must be between 0 and 100.", 12
```

Remember that the variables in a procedure are normally available only within that procedure. When a procedure ends, the variables it uses are erased from memory. You can preserve the value of a variable between calls to a procedure by declaring the variable with the **STATIC** statement. To share variables between procedures, use the **SHARED** statement, or pass the variables as arguments when calling the procedures. To share variables with procedures contained in separate modules, use the **COMMON** statement.

How do I use procedures and modules in the QuickBASIC environment?

QuickBASIC provides ways for you to create and organize procedures.

To create a new procedure:

➤ In the View window, type a **SUB** or **FUNCTION** statement followed by a procedure name. When you press Enter at the end of the statement, QuickBASIC opens a new editing window for the procedure and inserts an **END SUB** or **END FUNCTION** statement. You can then write the code for the procedure.

To edit another procedure in your program or switch back to the module-level code:

1. Press F2. QuickBASIC displays the SUBs dialog box, which lists the modules and procedures of your program.
2. From the SUBs dialog box, select a procedure to edit, move a procedure to another module, or delete a procedure from your program.

To create a new module:

➤ From the File menu, choose Create File and specify a filename. You can then write new procedures to include in the module, or move procedures from other modules into the new module.

To load an existing source-code module:

➤ From the File menu, choose Load File.

When you exit QuickBASIC with more than one module loaded, QuickBASIC creates a .MAK file that contains the filename of each module. When you open your program later, QuickBASIC uses the .MAK file to load all your program's modules for you.

When you choose the Make EXE File command, QuickBASIC automatically compiles and links all the currently loaded modules. If you compile your programs from the command line using BC, you must compile each .BAS file separately, then combine the resulting object-module files into an executable file using LINK. For more information about using multiple modules, see Chapter 7, "Programming with Modules," in *Programming in BASIC*.

For more information about using procedures in QuickBASIC, see Chapter 2, "SUB and FUNCTION Procedures," in *Programming in BASIC*.

How can I get more string space?

The QuickBASIC compiler places ordinary variables, variable-length strings, and static arrays in one 64K near-data segment called DGROUP. If you run out of space for string variables in DGROUP, QuickBASIC displays the error message `Out of string space`. You can determine if you are close to filling up this segment by using the **FRE** function. For example, the following statement obtains the number of bytes available in DGROUP:

```
FreeMem& = FRE("")
```

Large arrays can easily use up the memory in DGROUP. By making your arrays dynamic, you can cause BASIC to place them in far memory rather than in DGROUP. You can declare a dynamic array in the following ways:

- Use the **$DYNAMIC** metacommand before declaring the array with the **DIM** statement.
- Declare the array with the **REDIM** statement.
- Declare the array with a variable or expression as a dimension.
- Declare the array in a **SUB** or **FUNCTION** procedure.

QuickBASIC stores all variable-length string arrays in DGROUP. To store strings in far memory, use fixed-length string arrays wherever possible.

The following code fragment demonstrates how to declare static and dynamic arrays:

```
'$DYNAMIC
DIM ID(100) AS INTEGER          ' Dynamic array, stored in far memory.

'$STATIC
DIM X(50) AS LONG               ' Static array, stored in DGROUP.

REDIM A$(10)                    ' Dynamic array of variable-length
                                ' strings, still stored in DGROUP.
N% = 100
DIM Name(N%) AS STRING * 25     ' Dynamic array of fixed-length
                                ' strings, stored in far memory.
```

How do I use huge dynamic arrays?

Arrays larger than 64K are referred to as *huge arrays*. To use numeric arrays or fixed-length string arrays larger than 64K, you must start QuickBASIC or BC with the /AH option. Also, the size of the elements in a huge array should be a power of 2 (2, 4, 8, 16, etc.); otherwise, the array will be limited to 128K.

The following code fragment demonstrates how to pad a user-defined data type to a power of 2, so that an array of that data type can be larger than 128K:

```
TYPE Padded
   Num   AS DOUBLE          ' 8 bytes.
   Fixed AS STRING * 215
   Pad   AS STRING * 33      ' 8 + 215 + 33 = 256 bytes, a power of 2.
END TYPE

'$DYNAMIC
DIM Huge(600) AS Padded      ' Array is larger than 128K.
```

What can I do to increase the speed of my programs?

Since integer math is much faster than floating-point math, the best way to increase the speed of your programs is to use integer variables wherever possible.

By default, BASIC variables are single-precision. To make integer the default data type for all variables, insert the following statement at the beginning of each module of your program:

```
DEFINT A-Z
```

To use variables of other data types, use the appropriate type-declaration character (!, #, &, or $) following the variable name.

NOTE *If your program has existing procedures when you add the* ***DEFINT*** *statement to the module, the default data type for variables in these procedures is not changed to integer. Since the default data type in BASIC is single-precision, QuickBASIC will add an implicit* ***DEFSNG*** *statement to previously existing procedures that did not already have a* ***DEF****type statement. For this reason, it is a good idea to start programs with a* ***DEFINT*** *statement before you create procedures.*

To further increase the speed of your programs, you should use integer division, which is much faster than floating-point division. Instead of the floating-point division operator (/), use the integer division operator (\) wherever possible.

How can I work around the limitations of floating-point math?

QuickBASIC stores numbers in binary form (base 2). While integers can easily be represented in binary, most fractional numbers cannot be exactly represented in a finite number of bits. For this reason, it is possible to get unexpected results from floating-point calculations.

Since all computer programs store numbers in binary form, this limitation exists in every programming language. Floating-point math is designed for scientific calculation, where you want as many digits of precision as possible. However, this type of precision is not appropriate for some applications.

The following example demonstrates how the small difference between the binary and decimal representation of fractions can compound in floating-point calculations. Because .0001 cannot be exactly represented in binary form, the example prints 1.000054 instead of 1. The small error in representing the fraction causes the calculation to be inaccurate:

```
Sum! = 0
FOR I% = 1 TO 10000
   Sum! = Sum! + .0001
NEXT I%
PRINT Sum!      ' Theoretically, Sum! is 1, but output is 1.000054.
```

When comparing floating-point variables, you should be aware of the possible difference in binary representation of fractions. For example, the following example will not print "Equal" as expected:

```
X# = 69.2 + .62
IF X# = 69.82 THEN PRINT "Equal"
```

The binary representation of the two numbers is not precisely equal. You can anticipate this discrepancy with the following statement:

```
IF ABS(X# - 69.82) < .01 THEN PRINT "Equal"
```

The limitations of floating-point math can be particularly troublesome in monetary calculations. In this case, you want exactly two digits after the decimal point, and single-precision variables have more digits of precision than you require. To work around this difficulty, you can use the long integer data type to represent monetary values. Since the long integer data type does not store fractional values, you must use the last two digits in the number to represent cents.

How can I use key trapping?

You can cause BASIC to pass control to an event-handling routine when a key is pressed. The following statements set up event trapping for the F10 key. When F10 is pressed, the program jumps to the event-handling routine:

```
ON KEY(10) GOSUB FTenHit    ' If F10 is pressed, go to label.
KEY(10) ON                  ' Turn on trapping for the F10 key.
   .
   .
   .

FTenHit:                    ' Event-handling routine, executed when
   PRINT "You hit F10!"     ' F10 is pressed.
   RETURN
```

To trap keys pressed in combination with the Ctrl, Alt, or Num Lock keys, you must first define these keys using the **KEY** statement. The key numbers 15–25 are reserved for user-defined key trapping. To trap a combination of keys, you must add the keyboard scan codes together. The following statement defines Ctrl+Break as key number 15, which could be trapped to prevent the user from breaking out of a program:

```
KEY 15, CHR$(4 + 128) + CHR$(70)' Scan codes: Ctrl = 4, Extended
                                ' Kbd = 128, Scroll Lock = 70.
```

For more information on event trapping, see Chapter 6, "Error and Event Trapping," in *Programming in BASIC*. The keyboard scan codes are found in Appendix D, "Keyboard Scan Codes and ASCII Character Codes," in *Programming in BASIC*.

How do I add routines to a Quick library?

Quick libraries contain compiled procedures that are loaded into memory when you start QuickBASIC. To add routines to a Quick library, you must rebuild it with all the desired routines. You should also maintain a .LIB file for each library, which is used in creating an executable (.EXE) file. You can build a Quick library from within QuickBASIC or at the command line.

To rebuild a Quick library from within QuickBASIC:

1. Invoke QuickBASIC with the /L option followed by the filename of the library you want to update. The Quick library and associated .LIB file must be available to QuickBASIC (in the current directory or a directory included in the LIB environment variable).
2. Load the .BAS files that contain the procedures you want to add to the Quick library.
3. From the Run menu, choose Make Library. Type a filename for the Quick library and choose Make Library.

To rebuild a Quick library from the command line:

1. Compile the routines you want to add to the library.
2. Using LIB, create a library that contains all the desired routines. You can include compiled or assembled routines written in other languages, provided the language and compiler or assembler are compatible with QuickBASIC.
3. Create the Quick library using LINK with the /Q option.

The following commands demonstrate how to build a new Quick library and associated .LIB file. The new libraries will contain the routines in NEW.BAS as well as those contained in OLD.LIB:

```
BC NEW.BAS ;
LIB NEW.LIB + OLD.LIB + NEW.OBJ ;
LINK /Q NEW.LIB, NEW.QLB, , BQLB45.LIB ;
```

For more information about Quick libraries, see Appendix H, "Creating and Using Quick Libraries," in *Programming in BASIC*.

What versions of other programming languages can I use with QuickBASIC?

You can use routines written in other programming languages with your BASIC programs by compiling the routines and linking the resulting object-module files with your BASIC programs. To use the routines from within QuickBASIC, you must also create a Quick library that contains the routines. The following versions of other programming languages are compatible with QuickBASIC:

Microsoft C version 5.10
Microsoft Quick C versions 2.00 or 2.01
Microsoft FORTRAN versions 4.10 and 5.00
Microsoft Macro Assembler (MASM), any version (5.10 recommended)

When compiling programs in other languages for use with BASIC, use the medium, large, or huge memory model.

Is there a general statement I can use to open a communications port?

If you cannot open a communications port, the problem may be related to the hardware configuration expected by QuickBASIC. The following **OPEN** statement will work on a variety of hardware configurations:

```
OPEN "COM1:300,N,8,1,BIN,CD0,CS0,DS0,OP0,RS,TB2048,RB2048" AS #1
```

The parameters in this **OPEN** statement set the following options:

- 300 baud
- No parity, 8 data bits, 1 stop bit
- Binary mode
- All hardware handshaking off
- 2048-byte transmit and receive buffers (512-byte buffers are the default)

To receive input from the communications port, use the **INPUT$** function. The **LOC** function returns the number of characters waiting to be read. The following statement reads all available data from a communications port opened with file number 1:

```
ComInput$ = INPUT$(LOC(1), #1)
```

For an example of programming for the communication port, see the example program TERMINAL.BAS described in Chapter 3, "File and Device I/O," in *Programming in BASIC*.

Where can I find more information about programming with QuickBASIC?

Several books about BASIC and DOS are listed at the end of Chapter 4, "Interlude: BASIC for Beginners," in *Learning to Use Microsoft QuickBASIC*. In addition to those books, the following publications contain information about BASIC programming, using the QuickBASIC environment, and programming for DOS.

- *Learn BASIC Now* by Michael Halvorson and David Rygmyr (Microsoft Press, 1989).
- *Microsoft QuickBASIC: Programmer's Quick Reference* by Kris Jamsa (Microsoft Press, 1989).
- *Microsoft QuickBASIC Programmer's Reference* by Douglas Hergert (Howard W. Sams & Company, 1990).
- *The QuickBASIC Journal*. Northeast Publishing,Warwick, RI. Phone (401) 274-5492.
- *Advanced MS-DOS Programming*, 2nd ed., by Ray Duncan (Microsoft Press, 1988).
- *Microsoft Mouse Programmer's Reference* (Microsoft Press, 1989).
- *MS-DOS Extensions: Programmer's Quick Reference* by Ray Duncan (Microsoft Press, 1989).
- *The New Peter Norton Programmer's Guide to the IBM PC and PS/2: the Ultimate Reference Guide to the Entire Family of IBM Personal Computers* by Peter Norton and Richard Wilton (Microsoft Press, 1988).
- *Programmer's Guide to PC & PS/2 Video Systems* by Richard Wilton (Microsoft Press, 1987).

Product Assistance Request

When you need assistance with this Microsoft product, please gather all information that applies to your problem. To help us assist you as quickly as possible, note any messages that appear on-screen when the problem occurs. Have your manual and product disks close at hand and have all the information requested on this form available when you call. You will also need to have your product serial number available; this number can be found in the README.DOC file located on the disk labelled "Setup."

Support Within the United States

Entry-level assistance for QuickBASIC version 4.5 for DOS and version 1.0 for the MacIntosh is available by calling (206) 646-5101. Support on this line is limited to questions about setup and installation, product inquiries, and bug or documentation error reports. Service hours are 8:00 AM to 6:00 PM PST, Monday through Friday.

Advanced support on "how-to" programming issues and code debugging is available through Microsoft OnCall for BASIC at (900) 896-9999. This service operates from 6:00 AM to 6:00 PM PST, Monday through Friday (excluding holidays). When you call this number you will be charged $2.00 per minute on your phone bill.

Although Microsoft cannot provide a personal response to technical questions sent by letter, please feel free to report problems and submit suggestions through the mail.

The Microsoft KnowlegeBase database containing answers to common questions and descriptions of known problems is available on the CompuServe electronic information service. For information on CompuServe call (800) 848-8990.

Microsoft Product Support Services also provides Microsoft OnLine and OnLine Plus, which are comprehensive electronic support options oriented specifically for the software developer and corporate customer. For more information on these services, please contact Microsoft OnLine Telemarketing at (800) 443-4672.

Support Outside the United States

Contact your nearest Microsoft subsidiary for information on technical support.

Diagnosing the Problem

Many common problems are related to how the system's environment is set up. To insure that your system is using the utilities and programs that came with your Microsoft product, check your system's PATH and directory structure. Pay particular attention to device drivers loaded in your CONFIG.SYS file and any programs that you load from your AUTOEXEC.BAT file, such as terminate-and-stay-resident (TSR) utilities. You can rename these system configuration files and restart your system to determine if the contents of these files are related to your problem.

Please be prepared to provide the following information regarding your problem, your software, and your hardware.

General

- What product and version number are you using?
- What is your product serial number?
- Can you reproduce the problem?
- Does the problem occur with another copy of the original disk of your Microsoft software?
- Does the problem occur on another computer, if available?
- If you are running any other software at the same time, such as a windowing environment or memory-resident utility, does the problem still occur when you don't use the other software?

Software

- Operating system name and version number. (You can determine the version by using the VER command at the MS DOS or OS/2 prompt.)
- Network software name. Does the problem still occur without the network loaded?
- Microsoft Windows or other windowing environment and version number.
- Other software loaded (keyboard enhancers, print spoolers, etc.).

Hardware

- Computer manufacturer and model.
- Memory configuration.
- Video graphics adapter and display manufacturer and model.
- Other boards and peripherals (mouse, printer, etc.).

Microsoft® QuickBASIC

Programming in BASIC

Table of Contents

Introduction

The QuickBASIC Language xxi
The QuickBASIC Environment xxii
Using This Manual xxii
Selected Programming Topics xxii
The Heart of BASIC . . . xxiii
Appendixes . xxiv
Document Conventions xxiv
Programming Style in this Manual . xxvi

PART 1 Selected Programming Topics

Chapter 1 Control-Flow Structures 5

1.1 Changing Statement Execution Order 5
1.2 Boolean Expressions 6
1.3 Decision Structures 9
1.3.1 Block IF...THEN...ELSE 11
1.3.2 SELECT CASE 13
1.3.2.1 Using the SELECT CASE Statement 14
1.3.2.2 SELECT CASE Versus ON...GOSUB 17
1.4 Looping Structures 19
1.4.1 FOR...NEXT Loops 19
1.4.1.1 Exiting a FOR...NEXT Loop with EXIT FOR 23
1.4.1.2 Pausing Program Execution with FOR...NEXT 24
1.4.2 WHILE...WEND Loops 25
1.4.3 DO...LOOP Loops 26

1.4.3.1 Loop Tests: One Way to Exit DO...LOOP 30
1.4.3.2 EXIT DO: An Alternative Way to Exit DO...LOOP 31
1.5 Sample Applications 32
1.5.1 Checkbook-Balancing Program (CHECK.BAS) 32
1.5.2 Carriage-Return and Line-Feed Filter (CRLF.BAS) 34

Chapter 2 SUB and FUNCTION Procedures 39

2.1 Procedures: Building Blocks for Programming 39
2.2 Comparing Procedures with Subroutines and Functions 40
2.2.1 Comparing SUB with GOSUB 40
2.2.1.1 Local and Global Variables 41
2.2.1.2 Use in Multiple-Module Programs 41
2.2.1.3 Operating on Different Sets of Variables 41
2.2.2 Comparing FUNCTION with DEF FN 42
2.2.2.1 Local and Global Variables 42
2.2.2.2 Changing Variables Passed to the Procedure 42
2.2.2.3 Calling the Procedure within Its Definition 43
2.2.2.4 Use in Multiple-Module Programs 43
2.3 Defining Procedures 44
2.4 Calling Procedures 45
2.4.1 Calling a FUNCTION Procedure 45
2.4.2 Calling a SUB Procedure 46
2.5 Passing Arguments to Procedures 47
2.5.1 Parameters and Arguments 48
2.5.2 Passing Constants and Expressions 49
2.5.3 Passing Variables 51
2.5.3.1 Passing Simple Variables 51
2.5.3.2 Passing an Entire Array 52
2.5.3.3 Passing Individual Array Elements 52
2.5.3.4 Using Array-Bound Functions 53

2.5.3.5 Passing an Entire Record . . . 53
2.5.3.6 Passing Individual Elements of a Record . . . 54
2.5.4 Checking Arguments with the DECLARE Statement . . . 54
2.5.4.1 When QuickBASIC Does Not Generate a DECLARE Statement . . . 55
2.5.4.2 Developing Programs outside the QuickBASIC Environment . . . 56
2.5.4.3 Using Include Files for Declarations . . . 57
2.5.4.4 Declaring Procedures in Quick Libraries . . . 60
2.5.5 Passing Arguments by Reference . . . 60
2.5.6 Passing Arguments by Value . . . 61
2.6 Sharing Variables with SHARED . . . 62
2.6.1 Sharing Variables with Specific Procedures in a Module . . . 62
2.6.2 Sharing Variables with All Procedures in a Module . . . 64
2.6.3 Sharing Variables with Other Modules . . . 66
2.6.4 The Problem of Variable Aliasing . . . 68
2.7 Automatic and STATIC Variables . . . 69
2.8 Preserving Values of Local Variables with the STATIC Statement . . . 69
2.9 Recursive Procedures . . . 71
2.9.1 The Factorial Function . . . 71
2.9.2 Adjusting the Size of the Stack . . . 72
2.10 Transferring Control to Another Program with CHAIN . . . 73
2.11 Sample Application: Recursive Directory Search (WHEREIS.BAS) 75

Chapter 3 File and Device I/O . . . 81
3.1 Printing Text on the Screen . . . 82
3.1.1 Screen Rows and Columns . . . 82
3.1.2 Displaying Text and Numbers with PRINT . . . 83
3.1.3 Displaying Formatted Output with PRINT USING . . . 85
3.1.4 Skipping Spaces and Advancing to a Specific Column . . . 85
3.1.5 Changing the Number of Columns or Rows . . . 86

3.1.6 Creating a Text Viewport 86
3.2 Getting Input from the Keyboard 88
3.2.1 The INPUT Statement 88
3.2.2 The LINE INPUT Statement 90
3.2.3 The INPUT$ Function 91
3.2.4 The INKEY$ Function 91
3.3 Controlling the Text Cursor 92
3.3.1 Positioning the Cursor 92
3.3.2 Changing the Cursor's Shape 93
3.3.3 Getting Information about the Cursor's Location 94
3.4 Working with Data Files 95
3.4.1 How Data Files Are Organized 95
3.4.2 Sequential and Random-Access Files 96
3.4.3 Opening a Data File 96
3.4.3.1 File Numbers in BASIC 97
3.4.3.2 File Names in BASIC 98
3.4.4 Closing a Data File 99
3.4.5 Using Sequential Files 100
3.4.5.1 Records in Sequential Files 100
3.4.5.2 Putting Data in a New Sequential File 101
3.4.5.3 Reading Data from a Sequential File 102
3.4.5.4 Adding Data to a Sequential File 103
3.4.5.5 Other Ways to Write Data to a Sequential File 103
3.4.5.6 Other Ways to Read Data from a Sequential File 105
3.4.6 Using Random-Access Files 108
3.4.6.1 Records in Random-Access Files 108
3.4.6.2 Adding Data to a Random-Access File 108
3.4.6.3 Reading Data Sequentially 113
3.4.6.4 Using Record Numbers to Retrieve Records 114

3.4.7 Binary File I/O 115
3.4.7.1 Comparing Binary Access and Random Access 115
3.4.7.2 Positioning the File Pointer with SEEK 116
3.5 Working with Devices 118
3.5.1 Differences between Device I/O and File I/O 119
3.5.2 Communications through the Serial Port 120
3.6 Sample Applications 121
3.6.1 Perpetual Calendar (CAL.BAS) 121
3.6.2 Indexing a Random-Access File (INDEX.BAS) . 125
3.6.3 Terminal Emulator (TERMINAL.BAS) 132

Chapter 4 String Processing . 135

4.1 Strings Defined 135
4.2 Variable- and Fixed-Length Strings 137
4.2.1 Variable-Length Strings . 137
4.2.2 Fixed-Length Strings 137
4.3 Combining Strings 138
4.4 Comparing Strings 140
4.5 Searching for Strings 141
4.6 Retrieving Parts of Strings 142
4.6.1 Retrieving Characters from the Left Side of a String 142
4.6.2 Retrieving Characters from the Right Side of a String 143
4.6.3 Retrieving Characters from Anywhere in a String 144
4.7 Generating Strings 145
4.8 Changing the Case of Letters 145
4.9 Strings and Numbers 146
4.10 Changing Strings 147
4.11 Sample Application: Converting a String to a Number (STRTONUM.BAS) . 147

Chapter 5 Graphics 149

5.1 What You Need for Graphics Programs 150

5.2 Pixels and Screen Coordinates 151

5.3 Drawing Basic Shapes: Points, Lines, Boxes, and Circles 152

5.3.1 Plotting Points with PSET and PRESET 152

5.3.2 Drawing Lines and Boxes with LINE 153

5.3.2.1 Using the STEP Option 154

5.3.2.2 Drawing Boxes 156

5.3.2.3 Drawing Dotted Lines 157

5.4 Drawing Circles and Ellipses with CIRCLE 158

5.4.1 Drawing Circles 158

5.4.2 Drawing Ellipses 159

5.4.3 Drawing Arcs 160

5.4.4 Drawing Pie Shapes 162

5.4.5 Drawing Shapes to Proportion with the Aspect Ratio 163

5.5 Defining a Graphics Viewport 165

5.6 Redefining Viewport Coordinates with WINDOW 168

5.6.1 The Order of Coordinate Pairs 171

5.6.2 Keeping Track of View and Physical Coordinates 171

5.7 Using Colors 172

5.7.1 Selecting a Color for Graphics Output 173

5.7.2 Changing the Foreground or Background Color 174

5.7.3 Changing Colors with PALETTE and PALETTE USING 176

5.8 Painting Shapes 178

5.8.1 Painting with Colors 179

5.8.2 Painting with Patterns: Tiling 181

5.8.2.1 Pattern-Tile Size in Different Screen Modes 181

5.8.2.2 Creating a Single-Color Pattern in Screen Mode 2 182

5.8.2.3 Creating a Multicolor Pattern in Screen Mode 1 . . . 185
5.8.2.4 Creating a Multicolor Pattern in Screen Mode 8 . . . 188
5.9 DRAW: a Graphics Macro Language . . . 191
5.10 Basic Animation Techniques . . . 193
5.10.1 Saving Images with GET . . . 194
5.10.2 Moving Images with PUT . . . 196
5.10.3 Animation with GET and PUT . . . 200
5.10.4 Animating with Screen Pages . . . 205
5.11 Sample Applications . . . 207
5.11.1 Bar-Graph Generator (BAR.BAS) . . . 207
5.11.2 Color in a Figure Generated Mathematically (MANDEL.BAS) . . . 212
5.11.3 Pattern Editor (EDPAT.BAS) . . . 218

Chapter 6 Error and Event Trapping . . . 225

6.1 Error Trapping . . . 225
6.1.1 Activating Error Trapping . . . 226
6.1.2 Writing an Error-Handling Routine . . . 226
6.1.2.1 Using ERR to Identify Errors . . . 227
6.1.2.2 Returning from an Error-Handling Routine . . . 229
6.2 Event Trapping . . . 231
6.2.1 Detecting Events by Polling . . . 232
6.2.2 Detecting Events by Trapping . . . 232
6.2.3 Specifying the Event to Trap and Activating Event Trapping . . . 233
6.2.4 Events That BASIC Can Trap . . . 233
6.2.5 Suspending or Disabling Event Trapping . . . 234
6.2.6 Trapping Keystrokes . . . 235
6.2.6.1 Trapping User-Defined Keys . . . 236
6.2.6.2 Trapping User-Defined Shifted Keys . . . 237
6.2.7 Trapping Music Events . . . 239
6.3 Error and Event Trapping in SUB or FUNCTION Procedures . . . 241

6.4 Trapping across Multiple Modules 242
6.4.1 Event Trapping across Modules 242
6.4.2 Error Trapping across Modules 243
6.5 Trapping Errors and Events in Programs Compiled with BC 246
6.6 Sample Application: Trapping File-Access Errors (FILERR.BAS) 248

Chapter 7 Programming with Modules 251
7.1 Why Use Modules? 251
7.2 Main Modules 252
7.3 Modules Containing Only Procedures 252
7.4 Creating a Procedures-Only Module 254
7.5 Loading Modules 254
7.6 Using the DECLARE Statement with Multiple Modules 255
7.7 Accessing Variables from Two or More Modules 256
7.8 Using Modules During Program Development 256
7.9 Compiling and Linking Modules 257
7.10 Quick Libraries 257
7.10.1 Creating Quick Libraries 258
7.11 Tips for Good Programming with Modules 259

PART 2 Heart of BASIC

Chapter 8 Statement and Function Summary 265

Chapter 9 Quick-Reference Tables 297
9.1 Summary of Control-Flow Statements 298
9.2 Summary of Statements Used in BASIC Procedures 299
9.3 Summary of Standard I/O Statements 301
9.4 Summary of File I/O Statements 302
9.5 Summary of String-Processing Statements and Functions 304

9.6 Summary of Graphics Statements and Functions 306
9.7 Summary of Trapping Statements and Functions 308

Appendix A Converting BASICA Programs to QuickBASIC 311
A.1 Source-File Format 311
A.2 Statements and Functions Prohibited in QuickBASIC 311
A.3 Statements Requiring Modification 312
A.4 Editor Differences in Handling Tabs 313

Appendix B Differences from Previous Versions of QuickBASIC. . . 315
B.1 QuickBASIC Features 315
B.1.1 Features New to QuickBASIC 4.5 317
B.1.2 Features Introduced in QuickBASIC 4.0 317
B.1.2.1 User-Defined Types 317
B.1.2.2 IEEE Format and Math-Coprocessor Support 317
B.1.2.3 Ranges of IEEE-Format Numbers 318
B.1.2.4 PRINT USING and IEEE-Format Numbers . . 318
B.1.3 Recompiling Old Programs with /MBF 319
B.1.4 Converting Files and Programs . . 319
B.1.5 Other QuickBASIC Features 321
B.1.5.1 Long (32-Bit) Integers 321
B.1.5.2 Fixed-Length Strings 322
B.1.5.3 Syntax Checking on Entry 322
B.1.5.4 Binary File I/O 322
B.1.5.5 FUNCTION Procedures 322
B.1.5.6 Support for the CodeView® Debugger 322
B.1.5.7 Other-Language Compatibility 323
B.1.5.8 Multiple Modules in Memory 323
B.1.5.9 ProKey™, SideKick®, and SuperKey® Compatibility 323

B.1.5.10 Insert and Overtype Modes 323
B.1.5.11 WordStar®-Style Keyboard Commands 323
B.1.5.12 Recursion 324
B.1.5.13 Error Listings during Separate Compilation 324
B.1.5.14 Assembly-Language Listings during Separate Compilation 324
B.2 Differences in the Environment 324
B.2.1 Choosing Commands and Options 325
B.2.2 Windows 325
B.2.3 New Menu 325
B.2.4 Menu Commands 325
B.2.5 Editing-Key Changes 327
B.3 Differences in Compiling and Debugging 327
B.3.1 Command-Line Differences 327
B.3.2 Separate Compilation Differences 329
B.3.3 User Libraries and BUILDLIB 329
B.3.4 Restrictions on Include Files 329
B.3.5 Debugging 330
B.4 Changes to the BASIC Language 330
B.5 File Compatibility 336

Appendix C Limits in QuickBASIC 337

Appendix D Keyboard Scan Codes and ASCII Character Codes . . 339
D.1 Keyboard Scan Codes 339
D.2 ASCII Character Codes 342

Appendix E BASIC Reserved Words 345

Appendix F Metacommands 347
F.1 Metacommand Syntax 347

F.2 Processing Additional Source Files: $INCLUDE . . . 347
F.3 Dimensioned Array Allocation: $STATIC and $DYNAMIC . . . 348

Appendix G Compiling and Linking from DOS . . . 349
G.1 BC, LINK, and LIB . . . 349
G.2 The Compiling and Linking Process . . . 350
G.3 Compiling with the BC Command . . . 351
G.3.1 Specifying File Names . . . 352
G.3.1.1 Uppercase and Lowercase Letters . . . 352
G.3.1.2 File-Name Extensions . . . 352
G.3.1.3 Path Names . . . 353
G.3.2 Using BC Command Options . . . 353
G.4 Linking . . . 355
G.4.1 Defaults for LINK . . . 357
G.4.2 Specifying Files to LINK . . . 358
G.4.3 Specifying Libraries to LINK . . . 359
G.4.4 Memory Requirements for LINK . . . 359
G.4.5 Linking with Mixed-Language Programs . . . 360
G.4.5.1 Pascal and FORTRAN Modules in QuickBASIC Programs . . . 361
G.4.5.2 STATIC Array Allocation in Assembly-Language Routines . . . 361
G.4.5.3 References to DGROUP in Extended Run-Time Modules . . . 361
G.4.6 Using LINK Options . . . 361
G.4.6.1 Viewing the Options List (/HE) . . . 363
G.4.6.2 Pausing during Linking (/PAU) . . . 363
G.4.6.3 Displaying Linker Process Information (/I) . . . 363
G.4.6.4 Preventing Linker Prompting (/B) . . . 363
G.4.6.5 Creating Quick Libraries (/Q) . . . 364
G.4.6.6 Packing Executable Files (/E) . . . 364
G.4.6.7 Disabling Segment Packing (/NOP) . . . 364
G.4.6.8 Ignoring the Usual BASIC Libraries (/NOD) . . . 364

G.4.6.9 Ignoring Dictionaries (/NOE) . . . 365
G.4.6.10 Setting Maximum Number of Segments (/SE) 365
G.4.6.11 Creating a Map File (/M) . . . 365
G.4.6.12 Including Line Numbers in a Map File (/LI) 367
G.4.6.13 Packing Contiguous Segments (/PAC) 367
G.4.6.14 Using the CodeView Debugger (/CO) 367
G.4.6.15 Distinguishing Case (/NOI) 368
G.4.7 Other LINK Command-Line Options . . . 368
G.5 Managing Stand-Alone Libraries: LIB . . . 369
G.5.1 Running LIB . . . 370
G.5.2 Usual Responses for LIB 372
G.5.3 Cross-Reference-Listing Files . . . 372
G.5.4 Command Symbols 372
G.5.5 LIB Options . . . 375
G.5.5.1 Ignoring Case for Symbols . . . 375
G.5.5.2 Ignoring Extended Dictionaries 375
G.5.5.3 Distinguishing Case for Symbols . . . 375
G.5.5.4 Setting Page Size . . . 375

Appendix H Creating and Using Quick Libraries . . . 377
H.1 Types of Libraries . . . 377
H.2 Advantages of Quick Libraries . . . 378
H.3 Creating a Quick Library . . . 378
H.3.1 Files Needed to Create a Quick Library . . . 379
H.3.2 Making a Quick Library 380

H.3.3 Making a Quick Library from within the Environment . . . 380
H.3.3.1 Unloading Unwanted Files . . . 380
H.3.3.2 Loading Desired Files . . . 381
H.3.3.3 Creating a Quick Library . . . 381
H.4 Using Quick Libraries . . . 382
H.4.1 Loading a Quick Library . . . 382
H.4.2 Floating-Point Arithmetic in Quick Libraries . . . 383
H.4.3 Viewing the Contents of a Quick Library . . . 384
H.5 The Supplied Library (QB.QLB) . . . 384
H.6 The .QLB File-Name Extension . . . 384
H.7 Making a Library from the Command Line . . . 385
H.8 Using Routines from Other Languages in a Quick Library . . . 385
H.8.1 Building a Quick Library . . . 386
H.8.2 Quick Libraries with Leading Zeros in the First Code Segment . . . 386
H.8.3 The B_OnExit Routine . . . 387
H.9 Memory Considerations with Quick Libraries . . . 388
H.10 Making Compact Executable Files . . . 389

Appendix I Error Messages . . . 391
I.1 Error-Message Display . . . 391
I.2 Invocation, Compile-Time, and Run-Time Error Messages . . . 393
I.3 LINK Error Messages . . . 420
I.4 LIB Error Messages . . . 431

Index . . . 437

Figure 1.1	Logic of FOR...NEXT Loop with Positive STEP	20
Figure 1.2	Logic of FOR...NEXT Loop with Negative STEP	21
Figure 1.3	Logic of WHILE...WEND Loop	26
Figure 1.4	Logic of DO WHILE...LOOP	27
Figure 1.5	Logic of DO UNTIL...LOOP	28
Figure 1.6	Logic of DO...LOOP WHILE	29
Figure 1.7	Logic of DO...LOOP UNTIL	30
Figure 2.1	Parameters and Arguments	48
Figure 3.1	Text Output on Screen	83
Figure 3.2	Records in Sequential Files	100
Figure 3.3	Records in a Random-Access File	108
Figure 5.1	Coordinates of Selected Pixels in Screen Mode 2	152
Figure 5.2	How Angles Are Measured for CIRCLE	161
Figure 5.3	The Aspect Ratio in Screen Mode 1	164
Figure 5.4	WINDOW Contrasted with WINDOW SCREEN	169
Figure 5.5	Patterned Circle	184
Figure 6.1	Program Flow of Control with RESUME	230
Figure 6.2	Program Flow of Control with RESUME NEXT	231
Figure 7.1	Main Module Showing Module-Level Code and Procedures	253
Figure H.1	Make Library Dialog Box	381

Tables

Table 1.1	Relational Operators Used in BASIC	6
Table 1.2	Block IF...THEN...ELSE Syntax and Example	11
Table 1.3	SELECT CASE Syntax and Example	14
Table 1.4	FOR...NEXT Syntax and Example	19
Table 1.5	WHILE...WEND Syntax and Example	25
Table 1.6	DO...LOOP Syntax and Example: Test at the Beginning	27
Table 1.7	DO...LOOP Syntax and Example: Test at the End	28
Table 3.1	Devices Supported by BASIC for I/O	118
Table 5.1	Color Palettes in Screen Mode 1	174
Table 5.2	Background Colors in Screen Mode 1	175
Table 5.3	Binary to Hexadecimal Conversion	183
Table 5.4	The Effect of Different Action Options in Screen Mode 2	197
Table 5.5	The Effect of Different Action Options on Color in Screen Mode 1 (Palette 1)	200
Table 6.1	BC Command-Line Options for Error and Event Trapping	255
Table 9.1	Statements Used in Looping and Decision-Making	298
Table 9.2	Statements Used in Procedures	299
Table 9.3	Statements and Functions Used for Standard I/O	301
Table 9.4	Statements and Functions Used for Data-File I/O	302
Table 9.5	Statements and Functions Used for Processing Strings	304
Table 9.6	Statements and Functions Used for Graphics Output	306
Table 9.7	Statements and Functions Used in Error and Event Trapping	308
Table A.1	Statements Requiring Modification	312
Table B.1	Features of Microsoft QuickBASIC Version 4.5	316
Table B.2	Menus with New Commands in QuickBASIC Version 4.5	326
Table B.3	Editing-Key Changes	327
Table B.4	QB and BC Options Not Used in QuickBASIC Versions 4.0 or 4.5	328

Table B.5 Options Introduced in Version 4.0 for the QB and BC Commands 328
Table B.6 Debugging-Key Changes 330
Table B.7 Changes to the BASIC Language 331
Table C.1 QuickBASIC Limits 337
Table G.1 Input to the BC Command 352
Table G.2 Input to the LINK Command 356
Table G.3 Input to the LIB Command 371
Table I-1 Run-Time Error Codes 392

Introduction

Microsoft® QuickBASIC 4.5 is a major advance in making BASIC both more powerful and easier to use. It provides the most advanced BASIC yet offered for microcomputers, supported by an operating environment that allows you to focus on program creation—not the mechanics of writing or debugging.

The QuickBASIC Language

If you already know how to program in BASICA (or a similar interpreted BASIC), you'll appreciate the enhanced language features QuickBASIC provides to make it easier to write and maintain your software. For example:

- The **SELECT CASE** statement cleanly transfers control to any block of code without the use of nested **IF...THEN...ELSE** statements. **SELECT CASE** permits an exceptionally wide range of test expressions, so you can create exactly the comparison you need.
- QuickBASIC's **SUB** and **FUNCTION** procedures allow you to place groups of program statements into subprograms that your main program can call repeatedly. QuickBASIC's modularity makes it easy to save these procedures and reuse them in other programs.
- QuickBASIC procedures are fully recursive—a procedure can call itself repeatedly. This simplifies the programming of many numerical and sorting algorithms that are best expressed recursively.
- You can define your own data types, made up of any combination of integer, real, and string variables. Related variables can be conveniently grouped under a single name, which simplifies passing them to a procedure or writing them to a file.
- QuickBASIC supports binary file access. Your programs can read and manipulate files in any format because binary I/O can directly access any byte in the file.

QuickBASIC is a powerful development tool for professional use. Yet it is also the ideal language for beginning and intermediate programmers—people who aren't professional programmers but need a language that helps them reach their programming goals efficiently.

The QuickBASIC Environment

QuickBASIC isn't just an outstanding language. It is also an integrated programming environment that significantly simplifies writing and debugging software:

- As you type in your program, a smart editor checks each line for syntax errors. When you are ready to run, press a single key to execute the program instantly. If something is wrong, use the full-screen editor to correct the problem, then run the program again.
- You can debug your programs without exiting QuickBASIC. The integrated debugger lets you examine and alter variables, execute any part of the program, or halt execution when a particular condition is met. All these things happen within the QuickBASIC environment; you don't have to alter the program or add **PRINT** statements.
- QuickBASIC 4.5 has two new commands to make the debugger even more powerful: the Instant Watch command and the Break on Errors command.
- Also new to QuickBASIC 4.5 is the Microsoft QB Advisor, our on-line help. The QB Advisor is always at hand, whether you are writing, running, or debugging. Just place the cursor on the keyword or user-defined name you want to know more about, then press F1. The QB Advisor describes the syntax of BASIC statements and functions, explains how to use them, and even provides usable programming examples.

Using This Manual

This manual is in three parts. Part 1, "Selected Programming Topics," provides information on specific programming techniques and strategies. Part 2, "Heart of BASIC," and the appendixes contain important reference material.

Selected Programming Topics

Each chapter in this first section focuses on a single programming area. Studying this material will help you to quickly master

- Control-flow structures
- **SUB** and **FUNCTION** procedures
- File and device input and output
- String processing
- Graphics

- Error and event trapping
- Programming with modules

The presentation of each topic is straightforward and easy to understand, with many short programming examples that demonstrate how each part of BASIC works. The progression is from simple to more complex topics, so you can work through this material at your own pace without worrying about the order in which to study it. The focus throughout is on utility, not theory—how you can solve common programming problems with QuickBASIC.

In addition to the short examples, the chapters contain complete working programs that demonstrate the programming principles presented in that chapter. For your convenience, these programs are also included on your QuickBASIC release disks.

If you're an experienced BASIC programmer, you'll probably want to browse through the table of contents for a topic that catches your interest. If you're a novice programmer, though, you should probably work through each chapter from beginning to end. If you have never programmed in any language, you should start with Chapter 4 of *Learning to Use Microsoft QuickBASIC*, "Interlude: BASIC for Beginners."

Regardless of your interests or background, these seven chapters will help you learn almost everything you need to know to write sophisticated BASIC applications.

The Heart of BASIC

The second part of this manual, the "Heart of BASIC," is a handy, two-part quick reference to BASIC statements and functions.

Chapter 8, "Statement and Function Summary," is an alphabetically arranged summary of every BASIC keyword, describing its action or use and displaying its syntax. If your memory needs jogging on statement or function use, turn to this section.

In Chapter 9, "Quick-Reference Tables," the most commonly used BASIC statements and functions are displayed in six sections in table form, with each statement given a brief description. The contents of these sections match the material presented in Chapters 1 through 6 of "Selected Programming Topics." If you are trying to figure out how to accomplish a particular programming task, turn to this section.

Appendixes

The third section of this manual is a group of appendixes containing reference information on

- Converting BASICA programs
- Differences from previous versions
- Limits in QuickBASIC
- Keyboard scan codes and ASCII codes
- BASIC reserved words
- Metacommands
- Compiling and linking from DOS
- Creating and using Quick libraries
- Error messages

Document Conventions

This manual uses the following typographic conventions:

Example of Convention	Description
QB.LIB, ADD.EXE, COPY, LINK, /X	Uppercase (capital) letters indicate file names and DOS-level commands. Uppercase is also used for command-line options (unless the application accepts only lowercase).
SUB, IF, LOOP, PRINT, WHILE, TIME$	Bold capital letters indicate language-specific keywords with special meaning to Microsoft BASIC. Keywords are a required part of 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 can use uppercase letters or lowercase letters.
`CALL NewProc(arg1!, var2%)`	This kind of type is used for program examples, program output, and error messages within the text.

```
'$INCLUDE:'BC.BI'
  .
  .
  .
CHAIN "PROG1"
END
```

A column of three dots indicates that part of the example program has been intentionally omitted.

```
' Make one pass
```

The apostrophe (single right quotation mark) marks the beginning of a comment in sample programs.

filespec

Italic letters indicate placeholders for information you must supply, such as a file name. Italics are also occasionally used in the text for emphasis.

[[*optional-item*]]

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.

ALT+F1

Small capital letters are used for the names of keys and key sequences, such as ENTER and CTRL+R.

A plus (+) indicates a combination of keys. For example, CTRL+E means to hold down the CTRL key while pressing the E key.

The carriage-return key, sometimes marked with a bent arrow, is referred to as ENTER.

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

	The key names used in this manual correspond to the names on the IBM® Personal Computer keys. Other machines may use different names.
"defined term"	Quotation marks usually indicate a new term defined in the text.
Video Graphics Array (VGA)	Acronyms are usually spelled out the first time they are used.

The syntax below (for the "**LOCK...UNLOCK**" statements) illustrates many of the typographic conventions in this manual:

LOCK[[#]] *filenumber*[[,{*record* | [[*start*]] **TO** *end* }]]
.
.
.
UNLOCK[[#]] *filenumber*[[,{*record* | [[*start*]] **TO** *end* }]]

NOTE *Throughout this manual, the term "DOS" refers to both the MS-DOS® and IBM Personal Computer DOS operating systems. The name of a specific operating system is used when it is necessary to note features that are unique to that system. The term "BASICA" refers to interpreted versions of BASIC in general.*

Programming Style in this Manual

The following guidelines were used in writing programs in this manual and on the distribution disks. These guidelines are only recommendations for program readability; you are not obliged to follow them when writing your own programs.

- Keywords and symbolic constants appear in uppercase letters:

```
' PRINT, DO, LOOP, UNTIL are keywords:
PRINT "Title Page"
DO LOOP UNTIL Response$ = "N"

' FALSE and TRUE are symbolic constants
' equal to 0 and -1, respectively:
CONST FALSE = 0, TRUE = NOT FALSE
```

- Variable names are in lowercase with an initial capital letter; variable names with more than one syllable may contain other capital letters to clarify the division:

```
NumRecords% = 45
DateOfBirth$ = "11/24/54"
```

- Line labels are used instead of line numbers. The use of line labels is restricted to event-trapping and error-handling routines, as well as **DATA** statements when referenced with **RESTORE**:

```
' TimerHandler and ScreenTwoData are line labels:
ON TIMER GOSUB TimerHandler
RESTORE ScreenTwoData
```

- As noted in the preceding section, a single apostrophe (') introduces comments:

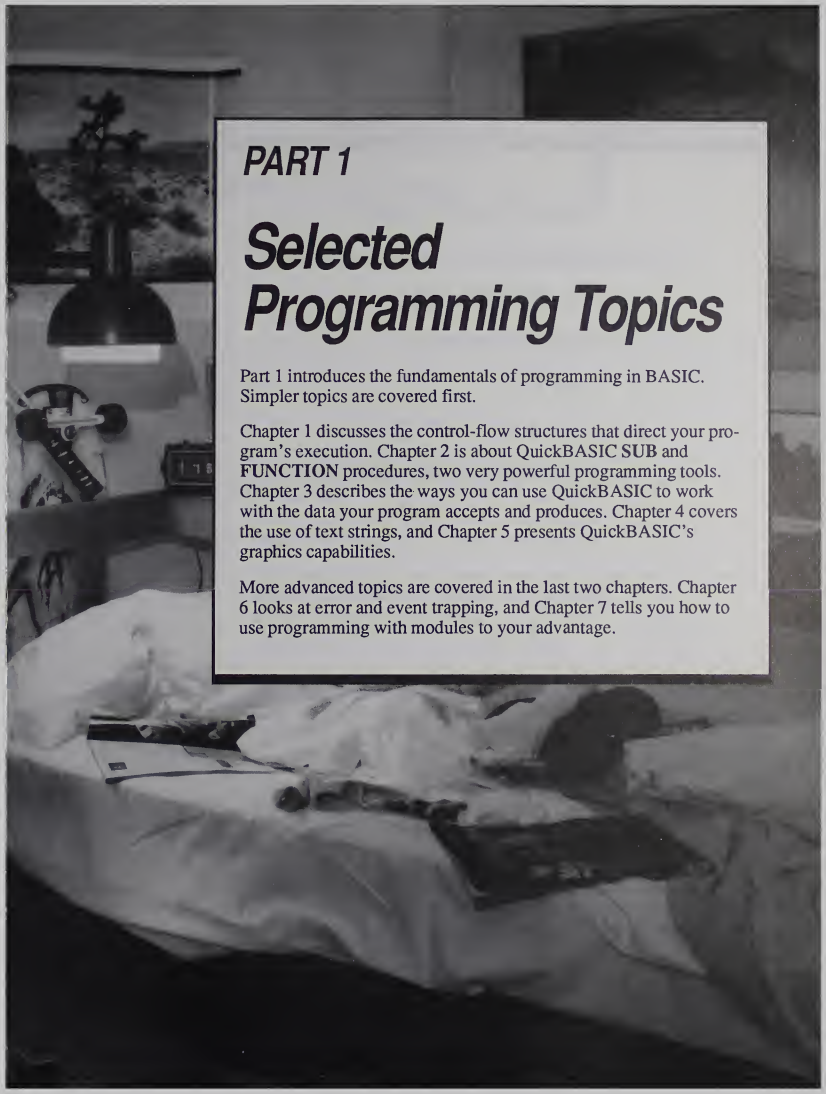
```
' This is a comment; these two lines
' are ignored when the program is running.
```

- Control-flow blocks and statements in procedures or subroutines are indented from the enclosing code:

```
SUB GetInput STATIC
   FOR I% = 1 TO 10
      INPUT X
      IF X > 0 THEN
         .
         .
         .
      ELSE
         .
         .
         .
      END IF
   NEXT I%
END SUB
```

PART 1

Selected Programming Topics



PART 1

Selected Programming Topics

Part 1 introduces the fundamentals of programming in BASIC. Simpler topics are covered first.

Chapter 1 discusses the control-flow structures that direct your program's execution. Chapter 2 is about QuickBASIC **SUB** and **FUNCTION** procedures, two very powerful programming tools. Chapter 3 describes the ways you can use QuickBASIC to work with the data your program accepts and produces. Chapter 4 covers the use of text strings, and Chapter 5 presents QuickBASIC's graphics capabilities.

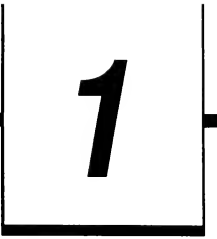
More advanced topics are covered in the last two chapters. Chapter 6 looks at error and event trapping, and Chapter 7 tells you how to use programming with modules to your advantage.

CHAPTERS

1 Control-Flow Structures 5
2 SUB and FUNCTION Procedures 39
3 File and Device I/O 81
4 String Processing 135
5 Graphics 149
6 Error and Event Trapping 225
7 Programming with Modules 251

CHAPTER 1

Control-Flow Structures



This chapter shows you how to use control-flow structures—specifically, loops and decision statements—to control the flow of your program's execution. Loops make a program execute a sequence of statements however many times you want. Decision statements let the program decide which of several alternative paths to take.

When you are finished with this chapter, you will know how to do the following tasks related to using loops and decision statements in your BASIC programs:

- Compare expressions using relational operators
- Combine string or numeric expressions with logical operators and determine if the resulting expression is true or false
- Create branches in the flow of the program with the statements **IF...THEN...ELSE** and **SELECT CASE**
- Write loops that repeat statements a specific number of times
- Write loops that repeat statements while or until a certain condition is true

1.1 Changing Statement Execution Order

Left unchecked by control-flow statements, a program's logic flows through statements from left to right, top to bottom. While some very simple programs can be written with only this unidirectional flow, most of the power and utility of any programming language comes from its ability to change statement execution order with decision structures and loops.

With a decision structure, a program can evaluate an expression, then branch to one of several different groups of related statements (statement blocks) depending on the result of the evaluation. With a loop, a program can repeatedly execute statement blocks.

If you are coming to this version of BASIC after programming in BASICA, you will appreciate the added versatility of these additional control-flow structures:

- The block **IF...THEN...ELSE** statement
- The **SELECT CASE** statement
- The **DO...LOOP** and **EXIT DO** statements
- The **EXIT FOR** statement, which provides an alternative way to exit **FOR...NEXT** loops

1.2 Boolean Expressions

A Boolean expression is any expression that returns the value "true" or "false." BASIC uses Boolean expressions in certain kinds of decision structures and loops. The following **IF...THEN...ELSE** statement contains a Boolean expression, `X < Y`:

```
IF X < Y THEN CALL Procedure1 ELSE CALL Procedure2
```

In the previous example, if the Boolean expression is true (if the value of the variable `X` is in fact less than the value of the variable `Y`), then `Procedure1` is executed; otherwise (if `X` is greater than or equal to `Y`), `Procedure2` is executed.

The preceding example also demonstrates a common use of Boolean expressions: comparing two other expressions (in this case, `X` and `Y`) to determine th relationship between them. These comparisons are made with the relational ope ators shown in Table 1.1.

Table 1.1 Relational Operators Used in BASIC

Operator	Meaning
=	Equal
< >	Not equal
<	Less than
<=	Less than or equal to
>	Greater than
>=	Greater than or equal to

You can use these relational operators to compare string expressions. In this case "greater than," "less than," and so on refer to alphabetical order. For example, the following expression is true, since the word "deduce" comes alphabetically before the word "deduct":

```
"deduce" < "deduct"
```

Boolean expressions also frequently use the "logical operators" **AND, OR, NOT, XOR, IMP,** and **EQV**. These operators allow you to construct compound tests from one or more Boolean expressions. For example,

expression1 **AND** *expression2*

is true only if *expression1* and *expression2* are both true. Thus, in the following example, the message `All sorted` is printed only if both the Boolean expressions `X <= Y` and `Y <= Z` are true:

```
IF (X <= Y) AND (Y <= Z) THEN PRINT "All sorted"
```

The parentheses around the Boolean expressions in the last example are not really necessary, since relational operators such as **<=** are evaluated before logical operators such as **AND**. However, parentheses make a complex Boolean expression more readable and ensure that its components are evaluated in the order that you intend.

BASIC uses the numeric values –1 and 0 to represent true and false, respectively. You can see this by asking BASIC to print a true expression and a false expression, as in the next example:

```
x = 5
y = 10
PRINT x < y  ' Evaluate, print a "true"  Boolean expression.
PRINT x > y  ' Evaluate, print a "false" Boolean expression.
```

Output

```
-1
 0
```

The value –1 for true makes more sense when you consider how BASIC's **NOT** operator works: **NOT** inverts each bit in the binary representation of its operand, changing 1 bits to 0 bits, and 0 bits to 1 bits. Therefore, since the integer value 0 (false) is stored internally as a sequence of sixteen 0 bits, **NOT** 0 (true) is stored internally as sixteen 1 bits, as shown below:

```
0000000000000000

TRUE = NOT FALSE = 1111111111111111
```

In the two's-complement method that BASIC uses to store integers, sixteen 1 bits represent the value –1.

Note that BASIC outputs –1 when it evaluates a Boolean expression as true; however, BASIC considers any nonzero value to be true, as shown by the output from the following example:

```
INPUT "Enter a value: ", x

IF x THEN PRINT x "is true."
```

Output

```
Enter a value: 2
 2 is true.
```

The **NOT** operator in BASIC is a "bitwise" operator. Some programming languages, such as C and Pascal, have both a bitwise **NOT** operator and a "logical" **NOT** operator. The distinction is as follows:

- A bitwise **NOT** returns false (0) only for the value –1.
- A logical **NOT** returns false (0) for any true (nonzero) value.

In BASIC, for any true *expression* not equal to –1, **NOT** *expression* returns another true value, as shown by the following list:

Value of expression	Value of NOT expression
1	–2
2	–3
–2	1
–1	0

So beware: **NOT** *expression* is false only if *expression* evaluates to a value of –1. If you define Boolean constants or variables for use in your programs, use –1 for true.

You can use the values 0 and –1 to define helpful mnemonic Boolean constants for use in loops or decisions. This technique is used in many of the examples in this manual, as shown in the following program fragment, which sorts the elements of an array named `Amount` in ascending order:

```
' Define symbolic constants to use in program:
CONST FALSE = 0, TRUE = NOT FALSE
.
.
.
```

```
DO
   Swaps = FALSE
   FOR I = 1 TO TransacNum - 1
      IF Amount(I) < Amount(I+1) THEN
         SWAP Amount(I), Amount(I+1)
         Swaps = TRUE
      END IF
   NEXT I
LOOP WHILE Swaps        ' Keep looping while Swaps is TRUE.
.
.
.
```

1.3 Decision Structures

Based on the value of an expression, decision structures cause a program to take one of the following two actions:

1. Execute one of several alternative statements within the decision structure itself
2. Branch to another part of the program outside the decision structure

In BASICA, decision-making is handled solely by the single-line **IF...THEN [[...ELSE]]** statement. In its simplest form (**IF...THEN**) the expression following the **IF** keyword is evaluated. If the expression is true, the program executes the statements following the **THEN** keyword; if the expression is false, the program continues with the next line after the **IF...THEN** statement. Lines 50 and 70 from the following BASICA program fragment show examples of **IF...THEN**:

```
30  INPUT A
40  ' If A is greater than 100, print a message and branch
45  ' back to line 30; otherwise, go on to line 60:
50  IF A > 100 THEN PRINT "Too big": GOTO 30
60  ' If A is equal to 100, branch to line 300;
65  ' otherwise, go on to line 80:
70  IF A = 100 THEN GOTO 300
80  PRINT A/100: GOTO 30
.
.
.
```

By adding the **ELSE** clause to an **IF...THEN** statement, you can have your program take one set of actions (those following the **THEN** keyword) if an expression is true, and another set of actions (those following the **ELSE** keyword) if it is false. The next program fragment shows how **ELSE** works in an **IF...THEN...ELSE** statement:

```
10 INPUT "What is your password"; Pass$
15 ' If user enters "sword", branch to line 50;
20 ' otherwise, print a message and branch back to line 10:
30 IF Pass$="sword" THEN 50 ELSE PRINT "Try again": GOTO 10
.
.
.
```

While BASICA's single-line **IF...THEN...ELSE** is adequate for simple decisions, it can lead to virtually unreadable code in cases of more complicated ones. This is especially true if you write your programs so all alternative actions take place within the **IF...THEN...ELSE** statement itself or if you nest **IF...THEN...ELSE** statements (that is, if you put one **IF...THEN...ELSE** inside another, a perfectly legal construction). As an example of how difficult it is to follow even a simple test, consider the next fragment from a BASICA program:

```
10 ' The following nested IF...THEN...ELSE statements print
15 ' different output for each of the following four cases:
20 '    1) A <= 50, B <= 50      3)  A > 50, B <= 50
25 '    2) A <= 50, B > 50       4)  A > 50, B > 50
30
35 INPUT A, B
40
45 ' Note: even though line 70 extends over several physical
50 ' lines on the screen, it is just one "logical line"
55 ' (everything typed before the <ENTER> key was pressed).
60 ' BASICA wraps long lines on the screen.
65
70  IF A <= 50 THEN IF B <= 50 THEN PRINT "A <= 50, B <= 50"
ELSE PRINT "A <= 50, B > 50" ELSE IF B <= 50 THEN PRINT "A >
50, B <= 50" ELSE PRINT "A > 50, B > 50"
```

To avoid the kind of complicated statement shown above, BASIC now includes the block form of the **IF...THEN...ELSE** statement, so that a decision is no longer restricted to one logical line. The following shows the same BASICA program rewritten to use block **IF...THEN...ELSE**:

```
INPUT A, B
IF A <= 50 THEN
   IF B <= 50 THEN
      PRINT "A <= 50, B <= 50"
   ELSE
      PRINT "A <= 50, B > 50"
   END IF
ELSE
   IF B <= 50 THEN
      PRINT "A > 50, B <= 50"
   ELSE
      PRINT "A > 50, B > 50"
   END IF
END IF
```

QuickBASIC also provides the **SELECT CASE...END SELECT** (referred to as **SELECT CASE**) statement for structured decisions.

Both the block **IF...THEN...ELSE** statement and the **SELECT CASE** statement allow the appearance of your code to be based on program logic, rather than requiring many statements to be crowded onto one line. This gives you increased flexibility while you are programming, as well as improved program readability and ease of maintenance when you are done.

1.3.1 Block IF...THEN...ELSE

Table 1.2 shows the syntax of the block **IF...THEN...ELSE** statement and gives an example of its use.

Table 1.2 Block IF...THEN...ELSE Syntax and Example

Syntax	Example

IF *condition1* **THEN**
  [*statementblock-1*]
[[**ELSEIF** *condition2* **THEN**
  [*statementblock-2*]]
.
.
.
[[**ELSE**
  [*statementblock-n*]]
END IF
	`IF X > 0 THEN`
	`  PRINT "X is positive"`
	`  PosNum = PosNum + 1`
	`ELSEIF X < 0 THEN`
	`  PRINT "X is negative"`
	`  NegNum = NegNum + 1`
	`ELSE`
	`  PRINT "X is zero"`
	`END IF`

The arguments *condition1*, *condition2*, and so on are expressions. They can be any numeric expression—in which case true becomes any nonzero value, and false is zero—or they can be Boolean expressions, in which case true is –1 and false is zero. As explained in Section 1.2, Boolean expressions typically compare two numeric or string expressions using one of the relational operators such as < or >=.

Each **IF**, **ELSEIF**, and **ELSE** clause is followed by a block of statements. None of the statements in the block can be on the same line as the **IF**, **ELSEIF**, or **ELSE** clause; otherwise, BASIC considers it a single-line **IF...THEN** statement.

BASIC evaluates each of the expressions in the **IF** and **ELSEIF** clauses from top to bottom, skipping over statement blocks until it finds the first true expression. When it finds a true expression, it executes the statements corresponding to the expression, then branches out of the block to the statement following the **END IF** clause.

If none of the expressions in the **IF** or **ELSEIF** clauses is true, BASIC skips to the **ELSE** clause, if there is one, and executes its statements. Otherwise, if there

is no **ELSE** clause, the program continues with the next statement after the **END IF** clause.

The **ELSE** and **ELSEIF** clauses are both optional, as shown in the following example:

```
' If the value of X is less than 100, do the two statements
' before END IF; otherwise, go to the INPUT statement
' following END IF:

IF X < 100 THEN
   PRINT X
   Number = Number + 1
END IF
INPUT "New value"; Response$
.
.
.
```

A single block **IF...THEN...ELSE** can contain multiple **ELSEIF** statements, as shown below:

```
IF C$ >= "A" AND C$ <= "Z" THEN
   PRINT "Capital letter"
ELSEIF C$ >= "a" AND C$ <= "z" THEN
   PRINT "Lowercase letter"
ELSEIF C$ >= "0" AND C$ <= "9" THEN
   PRINT "Number"
ELSE
   PRINT "Not alphanumeric"
END IF
```

At most, only one block of statements is executed, even if more than one condition is true. For example, if you enter the word `ace` as input to the next example, it prints the message `Input too short` but does not print the message `Can't start with an a`:

```
INPUT Check$
IF LEN(Check$) > 6 THEN
   PRINT "Input too long"
ELSEIF LEN(Check$) < 6 THEN
   PRINT "Input too short"
ELSEIF LEFT$(Check$, 1) = "a" THEN
   PRINT "Can't start with an a"
END IF
```

IF...THEN...ELSE statements can be nested; in other words, you can put an **IF...THEN...ELSE** statement inside another **IF...THEN...ELSE** statement, as shown here:

```
IF X > 0 THEN
   IF Y > 0 THEN
      IF Z > 0 THEN
         PRINT "All are greater than zero."
      ELSE
         PRINT "Only X and Y greater than zero."
      END IF
   END IF
ELSEIF X = 0 THEN
   IF Y = 0 THEN
      IF Z = 0 THEN
         PRINT "All equal zero."
      ELSE
         PRINT "Only X and Y equal zero."
      END IF
   END IF
ELSE
   PRINT "X is less than zero."
END IF
```

1.3.2 SELECT CASE

The **SELECT CASE** statement is a multiple-choice decision structure similar to the block **IF...THEN...ELSE** statement. Block **IF...THEN...ELSE** can be used anywhere **SELECT CASE** can be used.

The major difference between the two is that **SELECT CASE** evaluates a single expression, then executes different statements or branches to different parts of the program based on the result. In contrast, a block **IF...THEN...ELSE** can evaluate completely different expressions.

Examples

The following examples illustrate the similarities and differences between the **SELECT CASE** and **IF...THEN...ELSE** statements. Here is an example of using block **IF...THEN...ELSE** for a multiple-choice decision:

```
INPUT X
IF X = 1 THEN
   PRINT "One"
ELSEIF X = 2 THEN
   PRINT "Two"
ELSEIF X = 3 THEN
   PRINT "Three"
ELSE
   PRINT "Must be integer from 1-3."
END IF
```

The above decision is rewritten using **SELECT CASE** below:

```
INPUT X
SELECT CASE X
   CASE 1
      PRINT "One"
   CASE 2
      PRINT "Two"
   CASE 3
      PRINT "Three"
   CASE ELSE
      PRINT "Must be integer from 1-3."
END SELECT
```

The following decision can be made either with the **SELECT CASE** or the block **IF...THEN...ELSE** statement. The comparison is more efficient with the **IF...THEN...ELSE** statement because different expressions are being evaluated in the **IF** and **ELSEIF** clauses.

```
INPUT X, Y
IF X = 0 AND Y = 0 THEN
   PRINT "Both are zero."
ELSEIF X = 0 THEN
   PRINT "Only X is zero."
ELSEIF Y = 0 THEN
   PRINT "Only Y is zero."
ELSE
   PRINT "Neither is zero."
END IF
```

1.3.2.1 Using the SELECT CASE Statement

Table 1.3 shows the syntax of a **SELECT CASE** statement and an example.

Table 1.3 SELECT CASE Syntax and Example

Syntax	Example

SELECT CASE *expression*
  CASE *expressionlist1*
  [*statementblock-1*]
  [**CASE** *expressionlist2*
  [*statementblock-2*]]]
  .
  .
  .
  [**CASE ELSE**
  [*statementblock-n*]]
END SELECT
	INPUT TestValue
	SELECT CASE TestValue
	  CASE 1, 3, 5, 7, 9
	  PRINT "Odd"
	  CASE 2, 4, 6, 8
	  PRINT "Even"
	  CASE IS < 1
	  PRINT "Too low"
	  CASE IS > 9
	  PRINT "Too high"
	  CASE ELSE
	  PRINT "Not an integer"
	END SELECT

The *expressionlist* arguments following a **CASE** clause can be one or more of the following, separated by commas:

- A numeric expression or a range of numeric expressions
- A string expression or a range of string expressions

To specify a range of expressions, use the following syntax for the **CASE** statement:

CASE *expression* **TO** *expression*
CASE IS *relational-operator expression*

The *relational-operator* is any of the operators shown in Table 1.1. For example, if you use `CASE 1 TO 4`, the statements associated with this case are executed when the *expression* in the **SELECT CASE** statement is greater than or equal to 1 and less than or equal to 4. If you use `CASE IS < 5`, the associated statements are executed only if *expression* is less than 5.

If you are expressing a range with the **TO** keyword, be sure to put the lesser value first. For example, if you want to test for a value between –5 and –1, write the **CASE** statement as follows:

```
CASE -5 TO -1
```

However, the following statement is not a valid way to specify the range from –5 to –1, so the statements associated with this case are never executed:

```
CASE -1 TO -5
```

Similarly, a range of string constants should list the string that comes first alphabetically:

```
CASE "aardvark" TO "bear"
```

Multiple expressions or ranges of expressions can be listed for each **CASE** clause, as in the following lines:

```
CASE 1 TO 4, 7 TO 9, WildCard1%, WildCard2%
CASE IS = Test$, IS = "end of data"
CASE IS < LowerBound, 5, 6, 12, IS > UpperBound
CASE IS < "HAN", "MAO" TO "TAO"
```

If the value of the **SELECT CASE** expression appears in the list following a **CASE** clause, only the statements associated with that **CASE** clause are executed. Control then jumps to the first executable statement following **END SELECT**, not the next block of statements inside the **SELECT CASE** structure, as shown by the output from the next example (where the user enters `1`):

```
INPUT x
SELECT CASE x
   CASE 1
      PRINT "One, ";
   CASE 2
      PRINT "Two, ";
   CASE 3
      PRINT "Three, ";
END SELECT
PRINT "that's all"
```

Output

```
? 1
One, that's all
```

If the same value or range of values appears in more than one **CASE** clause, only the statements associated with the first occurrence are executed, as shown by the next example's output (where the user has entered `ABORIGINE`):

```
INPUT Test$
SELECT CASE Test$
   CASE "A" TO "AZZZZZZZZZZZZZZZZZ"
      PRINT "An uppercase word beginning with capital A"
   CASE IS < "A"
      PRINT "Some sequence of nonalphabetic characters"
   CASE "ABORIGINE"
      ' This case is never executed since ABORIGINE
      ' falls within the range in the first CASE clause:
      PRINT "A special case"
END SELECT
```

Output

```
? ABORIGINE
An uppercase word beginning with capital A
```

If you use a **CASE ELSE** clause, it must be the last **CASE** clause listed in the **SELECT CASE** statement. The statements between a **CASE ELSE** clause and an **END SELECT** clause are executed only if the *expression* does not match any of the other **CASE** selections in the **SELECT CASE** statement. In fact, it is a good idea to have a **CASE ELSE** statement in your **SELECT CASE** block to handle unforeseen values for *expression*. However, if there is no **CASE ELSE** statement and no match is found in any **CASE** statement for *expression*, the program continues execution.

Example

The following program fragment demonstrates a common use of the **SELECT CASE** statement: it prints a menu on the screen, then branches to different subprograms based on the number typed by the user.

```
DO                          ' Start menu loop.

   CLS                      ' Clear screen.

   ' Print five menu choices on the screen:
   PRINT "MAIN MENU" : PRINT
   PRINT "1)  Add New Names"
   PRINT "2)  Delete Names"
   PRINT "3)  Change Information"
   PRINT "4)  List Names"
   PRINT
   PRINT "5)  EXIT"

   ' Print input prompt:
   PRINT : PRINT "Type your selection (1 to 5):"

   ' Wait for the user to press a key. INPUT$(1)
   ' reads one character input from the keyboard:
   Ch$ = INPUT$(1)

   ' Use SELECT CASE to process response:
   SELECT CASE Ch$

      CASE "1"
         CALL AddData
      CASE "2"
         CALL DeleteData
      CASE "3"
         CALL ChangeData
      CASE "4"
         CALL ListData
      CASE "5"
         EXIT DO            ' The only way to exit the menu loop
      CASE ELSE
         BEEP
   END SELECT

LOOP                        ' End menu loop.

END

' Subprograms AddData, DeleteData, ChangeData, and ListData:
.
.
.
```

1.3.2.2 SELECT CASE Versus ON...GOSUB

You can use the more versatile **SELECT CASE** statement as a replacement for the old **ON** *expression* {**GOSUB** | **GOTO**} statement. The **SELECT CASE**

statement has many advantages over the **ON...GOSUB** statement, as summarized below:

- The *expression* in **SELECT CASE** *expression* can evaluate to either a string or numeric value. The *expression* given in the statement **ON** *expression* {**GOSUB** | **GOTO**} must evaluate to a number within the range 0 to 255.
- The **SELECT CASE** statement branches to a statement block immediately following the matching **CASE** clause. In contrast, **ON** *expression* **GOSUB** branches to a subroutine in another part of the program.
- **CASE** clauses can be used to test an *expression* against a range of values. When the range is quite large, this is not easily done with **ON** *expression* {**GOSUB** | **GOTO**}, especially as shown in the following fragments.

In the fragment below, the **ON...GOSUB** statement branches to one of the subroutines 50, 100, or 150, depending on the value input by the user:

```
X% = -1
WHILE X%
   INPUT "Enter choice (0 to quit): ", X%
   IF X% = 0 THEN END
   WHILE X% < 1 OR X% > 12
      PRINT "Must be value from 1 to 12"
      INPUT "Enter choice (0 to quit): ", X%
   WEND
   ON x% GOSUB 50,50,50,50,50,50,50,50,100,100,100,150
WEND
.
.
.
```

Contrast the preceding example with the next one, which uses a **SELECT CASE** statement with ranges of values in each **CASE** clause:

```
DO
   INPUT "Enter choice (0 to quit): ", X%
   SELECT CASE X%
      CASE 0
         END
      CASE 1 TO 8          ' Replaces "subroutine 50"
                           ' in preceding example
      CASE 9 TO 11         ' Replaces "subroutine 100"
                           ' in preceding example
      CASE 12              ' Replaces "subroutine 150"
                           ' in preceding example
      CASE ELSE            ' Input was out of range.
         PRINT "Must be value from 1 to 12"
      END SELECT
LOOP
```

1.4 Looping Structures

Looping structures repeat a block of statements (the loop), either for a specified number of times or until a certain expression (the loop condition) is true or false.

Users of BASICA are familiar with two looping structures, **FOR...NEXT** and **WHILE...WEND**, which are discussed in Sections 1.4.1 and 1.4.2, respectively. QuickBASIC has extended the available loop structures with the **DO...LOOP** statement, discussed in Section 1.4.3.

1.4.1 FOR...NEXT Loops

A **FOR...NEXT** loop repeats the statements enclosed in the loop a definite number of times, counting from a starting value to an ending value by increasing or decreasing a loop counter. As long as the loop counter has not reached the ending value, the loop continues to execute. Table 1.4 shows the syntax of the **FOR...NEXT** statement and gives an example of its use.

Table 1.4 FOR...NEXT Syntax and Example

Syntax	Example

FOR *counter* = *start* **TO** *end* [**STEP** *stepsize*]
  [*statementblock-1*]
  [**EXIT FOR**
  [*statementblock-2*]]
NEXT [*counter*]
	`FOR I% = 1 to 10`
	`  Array%(I%) = I%`
	`NEXT`

In a **FOR...NEXT** loop, the *counter* variable initially has the value of the expression *start*. After each repetition of the loop, the value of *counter* is adjusted. If you leave off the optional **STEP** keyword, the default value for this adjustment is one; that is, one is added to *counter* each time the loop executes. If you use **STEP**, then *counter* is adjusted by the amount *stepsize*. The *stepsize* argument can be any numeric value; if it is negative, the loop counts down from *start* to *end*. After the *counter* variable is increased or decreased, its value is compared with *end*. At this point, if either of the following is true, the loop is completed:

- The loop is counting up (*stepsize* is positive) and *counter* is greater than *end*.
- The loop is counting down (*stepsize* is negative) and *counter* is less than *end*.

Figure 1.1 shows the logic of a **FOR...NEXT** loop when the value of *stepsize* is positive.

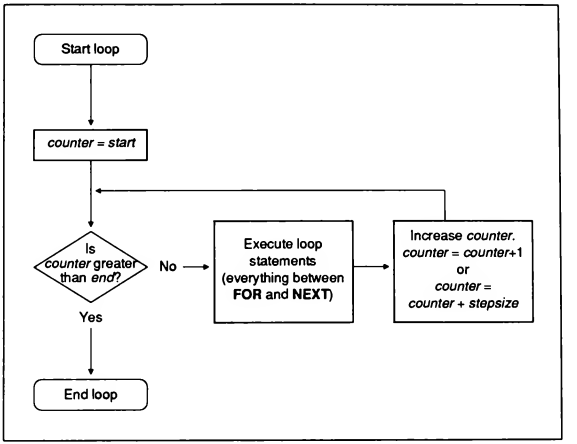


Figure 1.1 Logic of FOR...NEXT Loop with Positive STEP

Figure 1.2 shows the logic of a **FOR...NEXT** loop when the value of *stepsize* is negative.

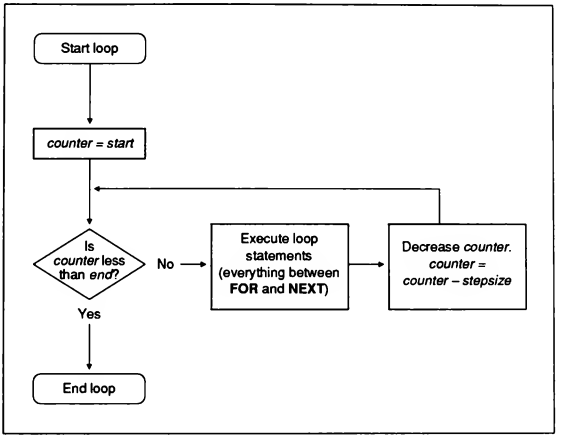


Figure 1.2 Logic of FOR...NEXT Loop with Negative STEP

A **FOR...NEXT** statement always "tests at the top," so if one of the following conditions is true, the loop is never executed:

- Step size is positive, and the initial value of *start* is greater than the value of *end*:

```
' Loop never executes, because I% starts out greater
' than 9:
FOR I% = 10 TO 9
  .
  .
  .
NEXT I%
```

- Step size is negative, and the initial value of *start* is less than the value of *end*:

```
' Loop never executes, because I% starts out less than 9:
FOR I% =  -10 TO -9 STEP -1
   .
   .
   .
NEXT I%
```

You don't have to use the *counter* argument in the **NEXT** clause; however, if you have several nested **FOR...NEXT** loops (one loop inside another), listing the *counter* arguments can be a helpful way to keep track of what loop you are in.

Here are some general guidelines for nesting **FOR...NEXT** loops:

- If you use a loop counter variable in a **NEXT** clause, the counter for a nested loop must appear before the counter for any enclosing loop. In other words, the following is a legal nesting:

```
FOR I = 1 TO 10
   FOR J = -5 TO 0
   .
   .
   .
   NEXT J
NEXT I
```

  However, the following is not a legal nesting:

```
FOR I = 1 TO 10
   FOR J = -5 TO 0
   .
   .
   .
   NEXT I
NEXT J
```

- If you use a separate **NEXT** clause to end each loop, then the number of **NEXT** clauses must always be the same as the number of **FOR** clauses.
- If you use a single **NEXT** clause to terminate several levels of **FOR...NEXT** loops, then the loop-counter variables must appear after the **NEXT** clause in "inside-out" order:

  NEXT *innermost-loopcounter, ... , outermost-loopcounter*

  In this case, the number of loop-counter variables in the **NEXT** clause must be the same as the number of **FOR** clauses.

Examples

The three program fragments below illustrate different ways of nesting **FOR... NEXT** loops to produce the identical output that follows. The first example shows nested **FOR...NEXT** loops with loop counters and separate **NEXT** clauses for each loop:

```
FOR I = 1 TO 2
   FOR J = 4 TO 5
      FOR K = 7 TO 8
         PRINT I, J, K
      NEXT K
   NEXT J
NEXT I
```

The following example also uses loop counters but only one **NEXT** clause for all three loops:

```
FOR I = 1 TO 2
   FOR J = 4 TO 5
      FOR K = 7 TO 8
         PRINT I, J, K
NEXT K, J, I
```

The final example shows nested **FOR...NEXT** loops without loop counters:

```
FOR I = 1 TO 2
   FOR J = 4 TO 5
      FOR K = 7 TO 8
         PRINT I, J, K
      NEXT
   NEXT
NEXT
```

Output

```
 1             4             7
 1             4             8
 1             5             7
 1             5             8
 2             4             7
 2             4             8
 2             5             7
 2             5             8
```

1.4.1.1 Exiting a FOR...NEXT Loop with EXIT FOR

Sometimes you may want to exit a **FOR...NEXT** loop before the counter variable reaches the ending value of the loop. You can do this with the **EXIT FOR** statement. A single **FOR...NEXT** loop can have any number of **EXIT FOR**

statements, and the **EXIT FOR** statements can appear anywhere within the loop. The following fragment shows one use for an **EXIT FOR** statement:

```
' Print the square roots of the numbers from 1 to 30,000.
' If the user presses any key while this loop is executing,
' control exits from the loop:
FOR I% = 1 TO 30000
   PRINT SQR(I%)
   IF INKEY$ <> "" THEN EXIT FOR
NEXT
.
.
.
```

EXIT FOR exits only the smallest enclosing **FOR...NEXT** loop in which it appears. For example, if the user pressed a key while the following nested loops were executing, the program would simply exit the innermost loop. If the outermost loop were still active (that is if the value of `I%` <= 100), control would pass right back to the innermost loop:

```
FOR I% = 1 TO 100
   FOR J% = 1 TO 100
      PRINT I% / J%
      IF INKEY$ <> "" THEN EXIT FOR
   NEXT J%
NEXT I%
```

1.4.1.2 Pausing Program Execution with FOR...NEXT

Many BASICA programs use an empty **FOR...NEXT** loop such as the following to insert a pause in a program:

```
.
.
.
' There are no statements in the body of this loop;
' all it does is count from 1 to 10,000
' using integers (whole numbers).
FOR I% = 1 TO 10000: NEXT
.
.
.
```

For very short pauses or pauses that do not have to be some exact interval, using **FOR...NEXT** is fine. The problem with using an empty **FOR...NEXT** loop in this way is that different computers, different versions of BASIC, or different compile-time options can all affect how quickly the arithmetic in a **FOR... NEXT** loop is performed. So the length of a pause can vary widely. QuickBASIC's **SLEEP** statement now provides a better alternative. (See Chapter 8, "Statement and Function Summary," for the syntax of **SLEEP**.)

1.4.2 WHILE...WEND Loops

The **FOR...NEXT** statement is useful when you know ahead of time exactly how many times a loop should be executed. When you cannot predict the precise number of times a loop should be executed, but do know the condition that will end the loop, the **WHILE...WEND** statement is useful. Instead of counting to determine if it should keep executing a loop, **WHILE...WEND** repeats the loop as long as the loop condition is true.

Table 1.5 shows the syntax of the **WHILE...WEND** statement and an example.

Table 1.5 WHILE...WEND Syntax and Example

Syntax	Example

WHILE *condition*
  [*statementblock*]
WEND
	INPUT R$
	WHILE R$ <> "Y" AND R$ <> "N"
	  PRINT "Answer must be Y or N."
	  INPUT R$
	WEND

Example

The following example assigns an initial value of ten to the variable X, then successively halves that value and keeps halving it until the value of X is less than .00001:

```
X = 10

WHILE X > .00001
   PRINT X
   X = X/2
WEND
```

Figure 1.3 illustrates the logic of a **WHILE...WEND** loop.

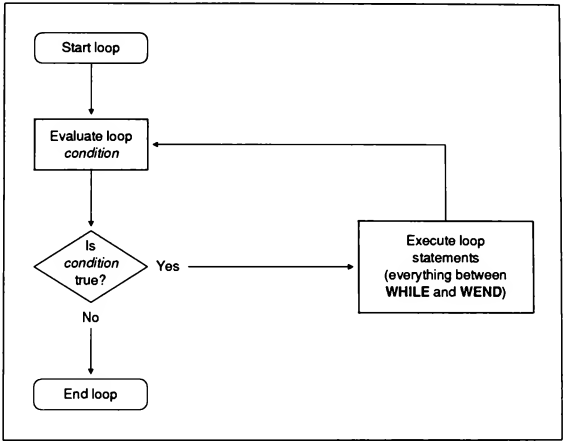


Figure 1.3 Logic of WHILE...WEND Loop

1.4.3 DO...LOOP Loops

Like the **WHILE...WEND** statement, the **DO...LOOP** statement executes a block of statements an indeterminate number of times; that is, exiting from the loop depends on the truth or falsehood of the loop condition. Unlike **WHILE...WEND**, **DO...LOOP** allows you to test for either a true or false condition. You can also put the test at either the beginning or the end of the loop.

Table 1.6 shows the syntax of a loop that tests at the loop's beginning.

Table 1.6 DO...LOOP Syntax and Example: Test at the Beginning

Syntax	Example

DO [{WHILE | UNTIL} *condition*]
[*statementblock-1*]
[EXIT DO
[*statementblock-2*]]
LOOP
	`DO UNTIL INKEY$ <> ""`
	`  ' An empty loop that`
	`  ' pauses until a key`
	`  ' is pressed`
	`LOOP`

Figures 1.4 and 1.5 illustrate the two kinds of **DO...LOOP** statements that test at the beginning of the loop.

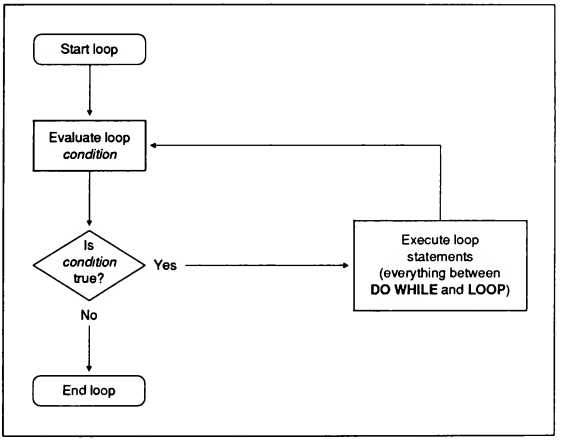


Figure 1.4 Logic of DO WHILE...LOOP

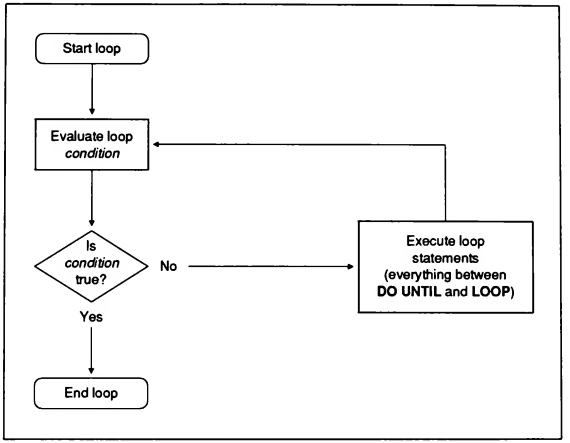


Figure 1.5 Logic of DO UNTIL...LOOP

Table 1.7 shows the syntax of a loop that tests for true or false at the end of the loop.

Table 1.7 DO...LOOP Syntax and Example: Test at the End

Syntax	Example

DO
  [*statementblock-1***]**
  [EXIT DO
  [*statementblock-2***]]**
LOOP [{WHILE | UNTIL} *condition***]**
	`DO`
	`  INPUT "Change: ", Ch`
	`  Total = Total + Ch`
	`LOOP WHILE Ch <> 0`

Figures 1.6 and 1.7 illustrate the two kinds of **DO...LOOP** statements that test at the end of the loop.

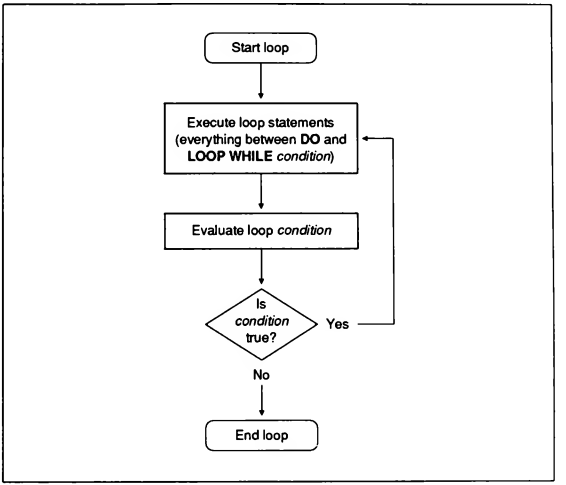


Figure 1.6 Logic of DO...LOOP WHILE

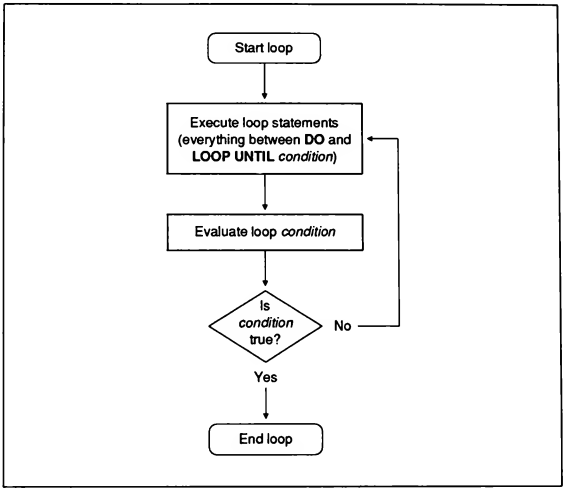


Figure 1.7 Logic of DO...LOOP UNTIL

1.4.3.1 Loop Tests: One Way to Exit DO...LOOP

You can use a loop test at the end of a **DO...LOOP** statement to create a loop in which the statements always execute at least once. With the **WHILE...WEND** statement, you sometimes have to resort to the trick of presetting the loop variable to some value in order to force the first pass through the loop. With **DO...LOOP**, such tricks are no longer necessary. The examples below illustrate both approaches:

```
' WHILE...WEND loop tests at the top, so assigning "Y"
' to Response$ is necessary to force execution of the
' loop at least once:
Response$ = "Y"
WHILE UCASE$(Response$) = "Y"
    .
    .
    .
```

```
   INPUT "Do again"; Response$
WEND

' The same loop using DO...LOOP to test after the
' body of the loop:
DO
   .
   .
   .
   INPUT "Do again"; Response$
LOOP WHILE UCASE$(Response$) = "Y"
```

You can also rewrite a condition expressed with **WHILE** using **UNTIL** instead, as in the following:

```
' ========================================================
'                  Using DO WHILE NOT...LOOP
' ========================================================

' While the end of file 1 has not been reached, read
' a line from the file and print it on the screen:
DO WHILE NOT EOF(1)
   LINE INPUT #1, LineBuffer$
   PRINT LineBuffer$
LOOP

' ========================================================
'                    Using DO UNTIL...LOOP
' ========================================================

' Until the end of file 1 has been reached, read
' a line from the file and print it on the screen:
DO UNTIL EOF(1)
   LINE INPUT #1, LineBuffer$
   PRINT LineBuffer$
LOOP
```

1.4.3.2 EXIT DO: An Alternative Way to Exit DO...LOOP

Inside a **DO...LOOP** statement, other statements are executed that eventually change the loop-test condition from true to false or false to true, ending the loop. In the **DO...LOOP** examples presented so far, the test has occurred either at the beginning or the end of the loop. However, by using the **EXIT DO** statement to exit from the loop, you can move the test elsewhere inside the loop. A single **DO...LOOP** can contain any number of **EXIT DO** statements, and the **EXIT DO** statements can appear anywhere within the loop.

Example

The following example opens a file and reads it, one line at a time, until the end of the file is reached or until it finds the pattern input by the user. If it finds the

pattern before getting to the end of the file, an **EXIT DO** statement exits the loop.

```
INPUT "File to search: ", File$
IF File$ = "" THEN END

INPUT "Pattern to search for: ", Pattern$
OPEN File$ FOR INPUT AS #1

DO UNTIL EOF(1)     ' EOF(1) returns a true value if the
                    ' end of the file has been reached.
   LINE INPUT #1, TempLine$
   IF INSTR(TempLine$, Pattern$) > 0 THEN

      ' Print the first line containing the pattern and
      ' exit the loop:
      PRINT TempLine$
      EXIT DO
   END IF
LOOP
```

1.5 Sample Applications

The sample applications for this chapter are a checkbook-balancing program and a program that ensures that every line in a text file ends with a carriage-return and line-feed sequence.

1.5.1 Checkbook-Balancing Program (CHECK.BAS)

This program prompts the user for the starting checking-account balance and all transactions—withdrawals or deposits—that have occurred. It then prints a sorted list of the transactions and the final balance in the account.

Statements and Functions Used

The program demonstrates the following statements discussed in this chapter:

- **DO...LOOP WHILE**
- **FOR...NEXT**
- **EXIT FOR**
- Block **IF...THEN...ELSE**

Program Listing

```
DIM Amount(1 TO 100)
CONST FALSE = 0, TRUE = NOT FALSE

CLS
' Get account's starting balance:
INPUT "Type starting balance, then press <ENTER>: ", Balance

' Get transactions. Continue accepting input
' until the input is zero for a transaction,
' or until 100 transactions have been entered:
FOR TransacNum% = 1 TO 100
   PRINT TransacNum%;
   PRINT ") Enter transaction amount (0 to end): ";
   INPUT "", Amount(TransacNum%)
   IF Amount(TransacNum%) = 0 THEN
      TransacNum% = TransacNum% - 1
      EXIT FOR
   END IF
NEXT

' Sort transactions in ascending order,
' using a "bubble sort":
Limit% = TransacNum%
DO
   Swaps% = FALSE
   FOR I% = 1 TO (Limit% - 1)

      ' If two adjacent elements are out of order,
      ' switch those elements:
      IF Amount(I%) < Amount(I% + 1) THEN
         SWAP Amount(I%), Amount(I% + 1)
         Swaps% = I%
      END IF
   NEXT I%

   ' Sort on next pass only to where last switch was made:
   Limit% = Swaps%

' Sort until no elements are exchanged:
LOOP WHILE Swaps%

' Print the sorted transaction array. If a transaction
' is greater than zero, print it as a "CREDIT"; if a
' transaction is less than zero, print it as a "DEBIT":
FOR I% = 1 TO TransacNum%
   IF Amount(I%) > 0 THEN
      PRINT USING "CREDIT: $$#####.##"; Amount(I%)
   ELSEIF Amount(I%) < 0 THEN
      PRINT USING "DEBIT: $$#####.##"; Amount(I%)
   END IF
```

```
    ' Update balance:
    Balance = Balance + Amount(I%)
NEXT I%

' Print the final balance:
PRINT
PRINT "--------------------------"
PRINT USING "Final Balance: $$######.##"; Balance
END
```

1.5.2 Carriage-Return and Line-Feed Filter (CRLF.BAS)

Some text files are saved in a format that uses only a carriage return (return to the beginning of the line) or a line feed (advance to the next line) to signify the end of a line. Many text editors expand this single carriage return (CR) or line feed (LF) to a carriage-return and line-feed (CR-LF) sequence whenever you load the file for editing. However, if you use a text editor that does not expand a single CR or LF to CR-LF, you may have to modify the file so it has the correct sequence at the end of each line.

The following program is a filter that opens a file, expands a single CR or LF to a CR-LF combination, then writes the adjusted lines to a new file. The original contents of the file are saved in a file with a .BAK extension.

Statements and Functions Used

This program demonstrates the following statements discussed in this chapter:

- **DO...LOOP WHILE**
- **DO UNTIL...LOOP**
- Block **IF...THEN...ELSE**
- **SELECT CASE...END SELECT**

To make this program more useful, it contains the following constructions not discussed in this chapter:

- A **FUNCTION** procedure named `Backup$` that creates the file with the .BAK extension.

  See Chapter 2, "SUB and FUNCTION Procedures," for more information on defining and using procedures.

- An error-handling routine named `ErrorHandler` to deal with errors that could occur when the user enters a file name. For instance, if the user enters the name of a nonexistent file, this routine prompts for a new name. Without this routine, such an error would end the program.

  See Chapter 6, "Error and Event Trapping," for more information on trapping errors.

Program Listing

```
DEFINT A-Z              ' Default variable type is integer.

' The Backup$ FUNCTION makes a backup file with
' the same base as FileName$ plus a .BAK extension:
DECLARE FUNCTION Backup$ (FileName$)

' Initialize symbolic constants and variables:
CONST FALSE = 0, TRUE = NOT FALSE

CarReturn$ = CHR$(13)
LineFeed$ = CHR$(10)

DO
   CLS

   ' Input the name of the file to change:
   INPUT "Which file do you want to convert"; OutFile$

   InFile$ = Backup$(OutFile$)   ' Get backup file's name.

   ON ERROR GOTO ErrorHandler    ' Turn on error trapping.

   NAME OutFile$ AS InFile$      ' Rename input file as
                                 ' backup file.

   ON ERROR GOTO 0               ' Turn off error trapping.

   ' Open backup file for input and old file for output:
   OPEN InFile$ FOR INPUT AS #1
   OPEN OutFile$ FOR OUTPUT AS #2

   ' The PrevCarReturn variable is a flag set to TRUE
   ' whenever the program reads a carriage-return character:
   PrevCarReturn = FALSE
```

```
' Read from input file until reaching end of file:
   DO UNTIL EOF(1)

      ' This is not end of file, so read a character:
      FileChar$ = INPUT$(1, #1)

      SELECT CASE FileChar$

         CASE CarReturn$         ' The character is a CR.

            ' If the previous character was also a
            ' CR, put a LF before the character:
            IF PrevCarReturn THEN
                FileChar$ = LineFeed$ + FileChar$
            END IF

            ' In any case, set the PrevCarReturn
            ' variable to TRUE:
            PrevCarReturn = TRUE

         CASE LineFeed$          ' The character is a LF.

            ' If the previous character was not a
            ' CR, put a CR before the character:
            IF NOT PrevCarReturn THEN
                FileChar$ = CarReturn$ + FileChar$
            END IF

            ' Set the PrevCarReturn variable to FALSE:
            PrevCarReturn = FALSE

         CASE ELSE               ' Neither a CR nor a LF.

            ' If the previous character was a CR,
            ' set the PrevCarReturn variable to FALSE
            ' and put a LF before the current character:
            IF PrevCarReturn THEN
               PrevCarReturn = FALSE
               FileChar$ = LineFeed$ + FileChar$
            END IF

      END SELECT

      ' Write the character(s) to the new file:
      PRINT #2, FileChar$;
   LOOP

   ' Write a LF if the last character in the file was a CR:
   IF PrevCarReturn THEN PRINT #2, LineFeed$;
```

```
CLOSE                          ' Close both files.
   PRINT "Another file (Y/N)?"  ' Prompt to continue.

   ' Change the input to uppercase (capital letter):
   More$ = UCASE$(INPUT$(1))

' Continue the program if the user entered a "y" or a "Y":
LOOP WHILE More$ = "Y"
END

ErrorHandler:           ' Error-handling routine
   CONST NOFILE = 53, FILEEXISTS = 58

   ' The ERR function returns the error code for last error:
   SELECT CASE ERR
      CASE NOFILE       ' Program couldn't find file
                        ' with input name.

         PRINT "No such file in current directory."
         INPUT "Enter new name: ", OutFile$
         InFile$ = Backup$(OutFile$)
         RESUME
      CASE FILEEXISTS   ' There is already a file named
                        ' <filename>.BAK in this directory:
                        ' remove it, then continue.
         KILL InFile$
         RESUME
      CASE ELSE         ' An unanticipated error occurred:
                        ' stop the program.
         ON ERROR GOTO 0
   END SELECT

' ======================== BACKUP$ ========================
'   This procedure returns a file name that consists of the
'   base name of the input file (everything before the ".")
'   plus the extension ".BAK"
' =========================================================

FUNCTION Backup$ (FileName$) STATIC

   ' Look for a period:
   Extension = INSTR(FileName$, ".")

   ' If there is a period, add .BAK to the base:
   IF Extension > 0 THEN
      Backup$ = LEFT$(FileName$, Extension - 1) + ".BAK"

   ' Otherwise, add .BAK to the whole name:
   ELSE
      Backup$ = FileName$ + ".BAK"
   END IF
END FUNCTION
```

CHAPTER 2

2 SUB and FUNCTION Procedures

This chapter explains how to simplify your programming by breaking programs into smaller logical components. These components—known as "procedures"—can then become building blocks that let you enhance and extend the BASIC language itself.

When you are finished with this chapter, you will know how to perform the following tasks with procedures:

- Define and call BASIC procedures
- Use local and global variables in procedures
- Use procedures instead of **GOSUB** subroutines and **DEF FN** functions
- Pass arguments to procedures and return values from procedures
- Write recursive procedures (procedures that can call themselves)

Although you can create a BASIC program with any text editor, the QuickBASIC editor makes it especially easy to write programs that contain procedures. Also, in most cases QuickBASIC automatically generates a **DECLARE** statement when you save your program. (Using a **DECLARE** statement ensures that only the correct number and type of arguments are passed to a procedure and allows your program to call procedures defined in separate modules.)

2.1 Procedures: Building Blocks for Programming

As used in this chapter, the term "procedure" covers both **SUB...END SUB** and **FUNCTION...END FUNCTION** constructions. Procedures are useful for

condensing repeated tasks. For example, suppose you are writing a program that you eventually intend to compile as a stand-alone application and you want the user of this application to be able to pass several arguments to the application from the command line. It then makes sense to turn this task—breaking the string returned by the **COMMAND$** function into two or more arguments—into a separate procedure. Once you have this procedure up and running, you can use it in other programs. In essence, you are extending BASIC to fit your individual needs when you use procedures.

These are the two major benefits of programming with procedures:

1. Procedures allow you to break your programs into discrete logical units, each of which can be more easily debugged than can an entire program without procedures.
2. Procedures used in one program can be used as building blocks in other programs, usually with little or no modification.

You can also put procedures in your own Quick library, which is a special file that you can load into memory when you start QuickBASIC. Once the contents of a Quick library are in memory with QuickBASIC, any program that you write has access to the procedures in the library. This makes it easier for all of your programs both to share and save code. (See Appendix H, "Creating and Using Quick Libraries," for more information on how to build Quick libraries.)

2.2 Comparing Procedures with Subroutines and Functions

If you are familiar with earlier versions of BASIC, you might think of a **SUB...END SUB** procedure as being roughly similar to a **GOSUB...RETURN** subroutine. You will also notice some similarities between a **FUNCTION... END FUNCTION** procedure and a **DEF FN...END DEF** function. However, procedures have many advantages over these older constructions, as shown in Sections 2.2.1 and 2.2.2 below.

NOTE *To avoid confusion between a **SUB** procedure and the target of a **GOSUB** statement, **SUB** procedures are referred to in this manual as "subprograms," while statement blocks accessed by **GOSUB...RETURN** statements are referred to as "subroutines."*

2.2.1 Comparing SUB with GOSUB

Although use of the **GOSUB** subroutine does help break programs into manageable units, **SUB** procedures have a number of advantages over subroutines, as discussed below.

2.2.1.1 Local and Global Variables

In **SUB** procedures, all variables are local by default; that is, they exist only within the scope of the **SUB** procedure's definition. To illustrate, the variable named `I` in the `Test` subprogram below is local to `Test`, and has no connection with the variable named `I` in the module-level code:

```
I = 1
CALL Test
PRINT I  ' I still equals 1.
END

SUB Test STATIC
      I = 50
END SUB
```

A **GOSUB** has a major drawback as a building block in programs: it contains only "global variables." With global variables, if you have a variable named `I` inside your subroutine, and another variable named `I` outside the subroutine but in the same module, they are one and the same. Any changes to the value of `I` in the subroutine affect `I` everywhere it appears in the module. As a result, if you try to patch a subroutine from one module into another module, you may have to rename subroutine variables to avoid conflict with variable names in the new module.

2.2.1.2 Use in Multiple-Module Programs

A **SUB** can be defined in one module and called from another. This significantly reduces the amount of code required for a program and increases the ease with which code can be shared among a number of programs.

A **GOSUB** subroutine, however, must be defined and used in the same module.

2.2.1.3 Operating on Different Sets of Variables

A **SUB** procedure can be called any number of times within a program, with a different set of variables being passed to it each time. This is done by calling the **SUB** with an argument list. (See Section 2.5, "Passing Arguments to Procedures," for more information on how to do this.) In the next example, the subprogram `Compare` is called twice, with different pairs of variables passed to it each time:

```
X = 4: Y = 5

CALL Compare (X, Y)

Z = 7: W = 2
CALL Compare (Z, W)
END
```

```
SUB Compare (A, B)
   IF A < B THEN SWAP A, B
END SUB
```

Calling a **GOSUB** subroutine more than once in the same program and having it operate on a different set of variables each time is difficult. The process involves copying values to and from global variables, as shown in the next example:

```
X = 4: Y = 5
A = X: B = Y
GOSUB Compare
X = A: Y = B

Z = 7 : W = 2
A = Z : B = W
GOSUB Compare
Z = A : W = B
END

Compare:
   IF A < B THEN SWAP A, B
RETURN
```

2.2.2 Comparing FUNCTION with DEF FN

While the multiline **DEF FN** function definition answers the need for functions more complex than can be squeezed on a single line, **FUNCTION** procedures give you this capability plus the additional advantages discussed below.

2.2.2.1 Local and Global Variables

By default, all variables within a **FUNCTION** procedure are local to it, although you do have the option of using global variables. (See Section 2.6, "Sharing Variables with SHARED," for more information on procedures and global variables.)

In a **DEF FN** function, variables used within the function's body are global to the current module by default (this is also true for **GOSUB** subroutines). However, you can make a variable in a **DEF FN** function local by putting it in a **STATIC** statement.

2.2.2.2 Changing Variables Passed to the Procedure

Variables are passed to **FUNCTION** procedures by reference or by value. When you pass a variable by reference, you can change the variable by changing its corresponding parameter in the **FUNCTION**. For example, after the call to `GetRemainder` in the next program, the value of `X` is 2, since the value of `M` is 2 at the end of the **FUNCTION**:

```
X = 89
Y = 40
PRINT GetRemainder(X, Y)
PRINT X, Y                    ' X is now 2.
END

FUNCTION GetRemainder (M, N)
   GetRemainder = M MOD N
   M = M \ N
END FUNCTION
```

Variables are passed to a **DEF FN** function only by value, so in the next example, `FNRemainder` changes `M` without affecting `X`:

```
DEF FNRemainder (M, N)
   FNRemainder = M MOD N
   M = M \ N
END DEF

X = 89
Y = 40
PRINT FNRemainder(X, Y)

PRINT X,Y    ' X is still
             ' 89.
```

See Sections 2.5.5 and 2.5.6 for more information on the distinction between passing by reference and by value.

2.2.2.3 Calling the Procedure within Its Definition

A **FUNCTION** procedure can be "recursive"; in other words, it can call itself within its own definition. (See Section 2.9 for more information on how procedures can be recursive.) A **DEF FN** function cannot be recursive.

2.2.2.4 Use in Multiple-Module Programs

You can define a **FUNCTION** procedure in one module and use it in another module. You need to put a **DECLARE** statement in the module in which the **FUNCTION** is used; otherwise, your program thinks the **FUNCTION** name refers to a variable. (See Section 2.5.4, "Checking Arguments with the DECLARE Statement," for more information on using **DECLARE** this way.)

A **DEF FN** function can only be used in the module in which it is defined. Unlike **SUB** or **FUNCTION** procedures, which can be called before they appear in the program, a **DEF FN** function must always be defined before it is used in a module.

NOTE *The name of a **FUNCTION** procedure can be any valid BASIC variable name, except one beginning with the letters **FN**. The name of a **DEF FN** function must always be preceded by **FN**.*

2.3 Defining Procedures

BASIC procedure definitions have the following general syntax:

```
{SUB | FUNCTION} procedurename [[( parameterlist)]] [[STATIC]]
    [[statementblock-1]]
    [[EXIT {SUB | FUNCTION}
        [[statementblock-2]]]
END {SUB | FUNCTION}
```

The following list describes the parts of a procedure definition:

Part	Description
{SUB | FUNCTION}	Marks the beginning of a **SUB** or **FUNCTION** procedure, respectively.
procedurename	Any valid variable name up to 40 characters long. The same name cannot be used for both a **SUB** and a **FUNCTION**.
parameterlist	A list of variables, separated by commas, that shows the number and type of arguments to be passed to the procedure. (Section 2.5.1 explains the difference between parameters and arguments.)
STATIC	If you use the **STATIC** attribute, local variables are **STATIC**; that is, they retain their values between calls to the procedure. If you omit the **STATIC** attribute, local variables are "automatic" by default; that is, they are initialized to zeros or null strings at the start of each procedure call. See Section 2.7, "Automatic and STATIC Variables," for more information.
END {SUB | FUNCTION}	Ends a **SUB** or **FUNCTION** definition. To run correctly, every procedure must have exactly one **END {SUB | FUNCTION}** statement. When your program encounters an **END SUB** or **END FUNCTION**, it exits the procedure and returns to the statement immediately following the one that called the procedure. You can also use one or more optional **EXIT {SUB | FUNCTION}** statements within the body of a procedure definition to exit from the procedure.

All valid BASIC expressions and statements are allowed within a procedure definition except the following:

- **DEF FN...END DEF, FUNCTION...END FUNCTION, or SUB...END SUB**

  It is not possible to nest procedure definitions or to define a **DEF FN** function inside a procedure. However, a procedure can call another procedure or a **DEF FN** function.
- **COMMON**
- **DECLARE**
- **DIM SHARED**
- **OPTION BASE**
- **TYPE...END TYPE**

Example

The following example is a **FUNCTION** procedure named `IntegerPower`:

```
FUNCTION IntegerPower& (X&, Y&) STATIC
   PowerVal& = 1
   FOR I& = 1 TO Y&
      PowerVal& = PowerVal& * X&
   NEXT I&
   IntegerPower& = PowerVal&
END FUNCTION
```

2.4 Calling Procedures

Calling a **FUNCTION** procedure is different from calling a **SUB** procedure, as shown in the next two sections.

2.4.1 Calling a FUNCTION Procedure

You call a **FUNCTION** procedure the same way you use an intrinsic BASIC function such as **ABS**, that is, by using its name in an expression, as shown here:

```
' Any of the following statements
' would call a FUNCTION named "ToDec" :
PRINT 10 * ToDec
X = ToDec
IF ToDec = 10 THEN PRINT "Out of range."
```

A **FUNCTION** can return values by changing variables passed to it as arguments. (See Section 2.5.5, "Passing Arguments by Reference," for an explanation of how this is done.) Additionally, a **FUNCTION** returns a single value in its name, so the name of a **FUNCTION** must agree with the type it returns. For example, if a **FUNCTION** returns a string value, either its name must have the dollar sign ($) type-declaration character appended to it or it must be declared as having the string type in a preceding **DEFSTR** statement.

Example

The following program shows a **FUNCTION** that returns a string value. Note that the type-declaration suffix for strings ($) is part of the procedure name.

```
Banner$ = GetInput$       ' Call the FUNCTION and assign the
                          ' return value to a string variable.
PRINT Banner$             ' Print the string.
END

' ======================= GETINPUT$ =======================
'    The $ type-declaration character at the end of this
'    FUNCTION name means that it returns a string value.
' =========================================================

FUNCTION GetInput$ STATIC

   ' Return a string of 10 characters read from the
   ' keyboard, echoing each character as it is typed:
   FOR I% = 1 TO 10
      Char$ = INPUT$(1)       ' Get the character.
      PRINT Char$;            ' Echo the character on the
                              ' screen.
      Temp$ = Temp$ + Char$   ' Add the character to the
                              ' string.
   NEXT
   PRINT
   GetInput$ = Temp$          ' Assign the string to the FUNCTION.
END FUNCTION
```

NOTE *The program example above is written for use in the QB environment only. It cannot be compiled using the BC command from DOS.*

2.4.2 Calling a SUB Procedure

A **SUB** procedure differs from a **FUNCTION** procedure in that a **SUB** cannot be called by using its name within an expression. A call to a **SUB** is a stand-alone statement, like BASIC's **CIRCLE** statement. Also, a **SUB** does not return a value in its name as does a **FUNCTION**. However, like a **FUNCTION**, a **SUB** can modify the values of any variables passed to it. (Section 2.5.5, "Passing Arguments by Reference," explains how this is done.)

There are two ways to call a **SUB** procedure:

1. Put its name in a **CALL** statement:

```
CALL PrintMessage
```

2. Use its name as a statement by itself:

```
PrintMessage
```

If you omit the **CALL** keyword, don't put parentheses around arguments passed to the **SUB**:

```
' Call the ProcessInput subprogram with CALL and pass the
' three arguments First$, Second$, and NumArgs% to it:
CALL ProcessInput  (First$, Second$, NumArgs%)

' Call the ProcessInput subprogram without CALL and pass
' it the same arguments (note no parentheses around the
' argument list):
ProcessInput  First$, Second$, NumArgs%
```

See Section 2.5 for more information on passing arguments to procedures.

If your program calls **SUB** procedures without using **CALL**, and if you are not using QuickBASIC to write the program, you must put the name of the **SUB** in a **DECLARE** statement before it is called:

```
DECLARE SUB CheckForKey
.
.
.
CheckForKey
```

You need to be concerned about this only if you are developing programs outside QuickBASIC, since QuickBASIC automatically inserts **DECLARE** statements wherever they are needed when it saves a program.

2.5 Passing Arguments to Procedures

Sections 2.5.1–2.5.4 explain how to tell the difference between parameters and arguments, how to pass arguments to procedures, and how to check arguments to make sure they are of the correct type and quantity.

2.5.1 Parameters and Arguments

The first step in learning about passing arguments to procedures is understanding the difference between the terms "parameter" and "argument":

Parameter	Argument
A variable name that appears in a **SUB**, **FUNCTION**, or **DECLARE** statement	A constant, variable, or expression passed to a **SUB** or **FUNCTION** when the **SUB** or **FUNCTION** is called

In a procedure definition, parameters are placeholders for arguments. As shown in Figure 2.1, when a procedure is called, arguments are plugged into the variables in the parameter list, with the first parameter receiving the first argument, the second parameter receiving the second argument, and so on.

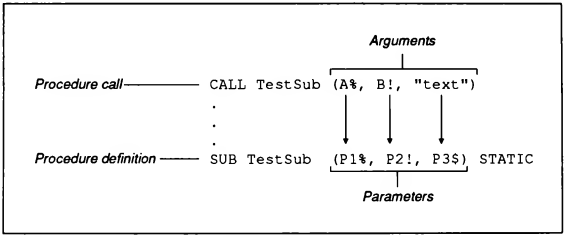


Figure 2.1 Parameters and Arguments

Figure 2.1 also demonstrates another important rule: although the names of variables in an argument list and a parameter list do not have to be the same, the number of parameters and the number of arguments do. Furthermore, the type (string, integer numeric, single-precision numeric, and so on) should be the same for corresponding arguments and parameters. (See Section 2.5.4, "Checking Arguments with the DECLARE Statement," for more information on how to ensure that arguments and parameters agree in number and type.)

A parameter list consists of any of the following, all separated by commas:

- Valid variable names, except for fixed-length strings

  For example, `x$` and `x AS STRING` are both legal in a parameter list, since they refer to variable-length strings. However, `x AS STRING * 10` refers to a fixed-length string 10 characters long and cannot appear in a parameter list. (Fixed-length strings are perfectly all right as *arguments* passed to procedures. See Chapter 4, "String Processing," for more information on fixed-length and variable-length strings.)

- Array names followed by left and right parentheses

An argument list consists of any of the following, all separated by commas:

- Constants
- Expressions
- Valid variable names
- Array names followed by left and right parentheses

Examples

The following example shows the first line of a subprogram definition with a parameter list:

```
SUB TestSub (A%, Array(), RecVar AS RecType, Cs$)
```

The first parameter, `A%`, is an integer; the second parameter, `Array()`, is a single-precision array, since untyped numeric variables are single precision by default; the third parameter, `RecVar`, is a record of type `RecType`; and the fourth parameter, `Cs$`, is a string.

The `CALL TestSub` line in the next example calls the `TestSub` subprogram and passes it four arguments of the appropriate type:

```
TYPE RecType
   Rank AS STRING * 12
   SerialNum AS LONG
END TYPE

DIM RecVar AS RecType

CALL TestSub (X%, A(), RecVar, "Daphne")
```

2.5.2 Passing Constants and Expressions

Constants—whether string or numeric—can appear in the list of arguments passed to a procedure. Naturally, a string constant must be passed to a string

parameter and a numeric constant to a numeric parameter, as shown in the next example:

```
CONST SCREENWIDTH = 80
CALL PrintBanner (SCREENWIDTH, "Monthly Status Report")
.
.
.
SUB PrintBanner (SW%, Title$)
   .
   .
   .
END SUB
```

If a numeric constant in an argument list does not have the same type as the corresponding parameter in the **SUB** or **FUNCTION** statement, then the constant is coerced to the type of the parameter, as you can see by the output from the next example:

```
CALL test(4.6, 4.1)
END

SUB test (x%, y%)
   PRINT x%, y%
END SUB
```

Output

```
 5             4
```

Expressions resulting from operations on variables and constants can also be passed to a procedure. As is the case with constants, numeric expressions that disagree in type with their parameters are coerced into agreement, as shown here:

```
Checker A! + 25!, NOT BooleanVal%

' In the next call, putting parentheses around the
' long-integer variable Bval& makes it an expression.
' The (Bval&) expression is coerced to a short integer
' in the Checker SUB:
Checker A! / 3.1, (Bval&)
.
.
.
END

SUB Checker (Param1!, Param2%)
   .
   .
   .
END SUB
```

2.5.3 Passing Variables

This section discusses how to pass simple variables, complete arrays, elements of arrays, records, and elements of records to procedures.

2.5.3.1 Passing Simple Variables

In both argument and parameter lists, you can declare the type for a simple variable in one of the following three ways:

1. Append one of the following type-declaration suffixes to the variable name: **%**, **&**, **!**, **#**, or **$**.
2. Declare the variable in a *declare variablename* **AS** *type* clause, where the placeholder *declare* can be either **DIM**, **COMMON**, **REDIM**, **SHARED**, or **STATIC**, and *type* can be either **INTEGER**, **LONG**, **SINGLE**, **DOUBLE**, **STRING**, or **STRING** * *n*. For example:

   ```
   DIM A AS LONG
   ```

3. Use a **DEF***type* statement to set the default type.

No matter which method you choose, corresponding variables must have the same type in both the argument and parameter lists, as shown in the following example.

Example

In this example, two arguments are passed to the **FUNCTION** procedure. The first is an integer giving the length of the string returned by the **FUNCTION**, while the second is a character that is repeated to make the string.

```
FUNCTION CharString$(A AS INTEGER, B$) STATIC
   CharString$ = STRING$(A%, B$)
END FUNCTION

DIM X AS INTEGER
INPUT "Enter a number (1 to 80): ", X
INPUT "Enter a character: ", Y$

' Print a string consisting of the Y$ character, repeated
' X number of times:
PRINT CharString$(X, Y$)
END
```

Output

```
Enter a number (1 to 80): 21
Enter a character: #
#####################
```

2.5.3.2 Passing an Entire Array

To pass all the elements of an array to a procedure, put the array's name, followed by left and right parentheses, in both the argument and parameter lists.

Example

This example shows how to pass all the elements of an array to a procedure:

```
DIM Values(1 TO 5) AS INTEGER

' Note empty parentheses after array name when calling
' procedure and passing array:
CALL ChangeArray (1, 5, Values())
CALL PrintArray (1, 5, Values())
END

' Note empty parentheses after P parameter:

SUB ChangeArray (Min%, Max%, P() AS INTEGER) STATIC
   FOR I% = Min% TO Max%
      P(I%) = I% ^ 3
   NEXT I%
END SUB

SUB PrintArray (Min%, Max%, P() AS INTEGER) STATIC
   FOR I% = Min% TO Max%
      PRINT P(I%)
   NEXT I%
   PRINT
END SUB
```

2.5.3.3 Passing Individual Array Elements

If a procedure does not require an entire array, you can pass individual elements of the array instead. To pass an element of an array, use the array name followed by the appropriate subscripts inside parentheses.

Example

The `SqrVal Array(4,2)` statement in the following example passes the element in row 4, column 2 of the array to the `SqrVal` subprogram (note how the subprogram actually changes the value of this array element):

```
DIM Array(1 TO 5,1 TO 3)

Array(4,2) = -36
PRINT Array(4,2)
SqrVal Array(4,2)
PRINT Array(4,2)      ' The call to SqrVal has changed
                      ' the value of Array(4,2).
END
```

```
SUB SqrVal(A) STATIC
   A = SQR(ABS(A))
END SUB
```

Output

```
-36
 6
```

2.5.3.4 Using Array-Bound Functions

The **LBOUND** and **UBOUND** functions are a useful way to determine the size of an array passed to a procedure. The **LBOUND** function finds the smallest index value of an array subscript, while the **UBOUND** function finds the largest one. These functions save you the trouble of having to pass the upper and lower bounds of each array dimension to a procedure.

Example

The subprogram in the following example uses the **LBOUND** function to initialize the variables `Row` and `Col` to the lowest subscript values in each dimension of `A`. It also uses the **UBOUND** function to limit the number of times the **FOR** loop executes to the number of elements in the array.

```
SUB PrintOut(A(2)) STATIC
   FOR Row = LBOUND(A,1) TO UBOUND(A,1)
      FOR Col = LBOUND(A,2) TO UBOUND(A,2)
         PRINT A(Row,Col)
      NEXT Col
   NEXT Row
END SUB
```

2.5.3.5 Passing an Entire Record

To pass a complete record (a variable declared as having a user-defined type) to a procedure, complete the following steps:

1. Define the type (`StockItem` in this example):

```
TYPE StockItem
   PartNumber  AS STRING * 6
   Description AS STRING * 20
   UnitPrice   AS SINGLE
   Quantity    AS INTEGER
END TYPE
```

2. Declare a variable (`StockRecord`) with that type:

```
DIM StockRecord AS StockItem
```

3. Call a procedure (`FindRecord`) and pass it the variable you have declared:

```
CALL FindRecord(StockRecord)
```

4. In the procedure definition, give the parameter the same type as the variable:

```
SUB FindRecord (RecordVar AS StockItem) STATIC
   .
   .
   .
END SUB
```

2.5.3.6 Passing Individual Elements of a Record

To pass an individual element in a record to a procedure, put the name of the element (*recordname.elementname*) in the argument list. Be sure, as always, that the corresponding parameter in the procedure definition agrees with the type of that element.

Example

The following example shows how to pass the two elements in the record variable `StockItem` to the `PrintTag` **SUB** procedure. Note how each parameter in the **SUB** procedure agrees with the type of the individual record elements.

```
TYPE StockType
   PartNumber AS STRING * 6
   Descrip AS STRING * 20
   UnitPrice AS SINGLE
   Quantity AS INTEGER
END TYPE

DIM StockItem AS StockType

CALL PrintTag (StockItem.Descrip, StockItem.UnitPrice)
.
.
.
END

SUB PrintTag (Desc$, Price AS SINGLE)
   .
   .
   .
END SUB
```

2.5.4 Checking Arguments with the DECLARE Statement

If you are using QuickBASIC to write your program, you will notice that QuickBASIC automatically inserts a **DECLARE** statement for each procedure whenever you save the program. Each **DECLARE** statement consists of the word

DECLARE, followed by the words **SUB** or **FUNCTION**, the name of the procedure, and a set of parentheses. If the procedure has no parameters, then the parentheses are empty. If the procedure has parameters, then the parentheses enclose a *parameterlist* that specifies the number and type of the arguments to be passed to the procedure. This *parameterlist* has the same format as the list in the definition line of the **SUB** or **FUNCTION**.

The purpose of the *parameterlist* in a **DECLARE** statement is to turn on "type checking" of arguments passed to the procedure. That is, every time the procedure is called with variable arguments, those variables are checked to be sure they agree with the number and type of the parameters in the **DECLARE** statement.

QuickBASIC puts all procedure definitions at the end of a module when it saves a program. Therefore, if there were no **DECLARE** statements, when you tried to compile this program with the BC command you would run into a problem known as "forward reference" (calling a procedure before it is defined). By generating a prototype of the procedure definition, **DECLARE** statements allow your program to call procedures that are defined later in a module, or in another module altogether.

Examples

The next example shows an empty parameter list in the **DECLARE** statement, since no arguments are passed to `GetInput$`:

```
DECLARE FUNCTION GetInput$ ()
X$ = GetInput$

FUNCTION GetInput$ STATIC
   GetInput$ = INPUT$(10)
END FUNCTION
```

The next example shows a parameter list in the **DECLARE** statement, since an integer argument is passed to this version of `GetInput$`:

```
DECLARE FUNCTION GetInput$ (X%)
X$ = GetInput$ (5)

FUNCTION GetInput$ (X%) STATIC
   GetInput$ = INPUT$(X%)
END FUNCTION
```

2.5.4.1 When QuickBASIC Does Not Generate a DECLARE Statement

In certain instances, QuickBASIC does not generate **DECLARE** statements in the module that calls a procedure.

QuickBASIC cannot generate a **DECLARE** statement in one module for a **FUNCTION** procedure defined in another module if the module is not loaded.

In such a case, you must type the **DECLARE** statement yourself at the beginning of the module where the **FUNCTION** is called; otherwise, QuickBASIC considers the call to the **FUNCTION** to be a variable name.

QuickBASIC does not generate a **DECLARE** statement for a **SUB** procedure in another module, whether that module is loaded or not. The **DECLARE** statement is not needed unless you want to call the **SUB** procedure without using the keyword **CALL**, however.

QuickBASIC also cannot generate a **DECLARE** statement for any procedure in a Quick library. You must add one to the program yourself.

2.5.4.2 Developing Programs outside the QuickBASIC Environment

If you are writing your programs with your own text editor and then compiling them outside the QuickBASIC environment with the BC and LINK commands, be sure to put **DECLARE** statements in the following three locations:

1. At the beginning of any module that calls a **FUNCTION** procedure before it is defined:

```
DECLARE FUNCTION Hypot (X!, Y!)

INPUT X, Y
PRINT Hypot(X, Y)
END

FUNCTION Hypot (A, B) STATIC
   Hypot = SQR(A ^ 2 + B ^ 2)
END FUNCTION
```

2. At the beginning of any module that calls a **SUB** procedure before it is defined and does not use **CALL** when calling the **SUB**:

```
DECLARE SUB PrintString (X, Y)
INPUT X, Y

PrintString X, Y      ' Note: no parentheses around
                      ' arguments
END

SUB PrintString (A,B) STATIC

   ' Convert the numbers to strings, remove any leading
   ' blanks from the second number, and print:
   PRINT STR$(A) + LTRIM$(STR$(B))
END SUB
```

When you call a **SUB** procedure with **CALL**, you don't have to declare the **SUB** first:

```
A$ = "466"
B$ = "123"
CALL PrintString(A$, B$)
END

SUB PrintString (X$, Y$) STATIC
   PRINT VAL(X$) + VAL(Y$)
END SUB
```

3. At the beginning of any module that calls a **SUB** or **FUNCTION** procedure defined in another module (an "external procedure")

If your procedure has no parameters, remember to put empty parentheses after the name of the procedure in the **DECLARE** statement, as in the next example:

```
DECLARE FUNCTION GetHour$ ()
PRINT GetHour$
END

FUNCTION GetHour$ STATIC
   GetHour$ = LEFT$(TIME$,2)
END FUNCTION
```

Remember, a **DECLARE** statement can appear only at the module level, not the procedure level. A **DECLARE** statement affects the entire module in which it appears.

2.5.4.3 Using Include Files for Declarations

If you have created a separate procedure-definition module that defines one or more **SUB** or **FUNCTION** procedures, it is a good idea to make an include file to go along with this module. This include file should contain the following:

- **DECLARE** statements for all the module's procedures.
- **TYPE...END TYPE** record definitions for any record parameters in this module's **SUB** or **FUNCTION** procedures.
- **COMMON** statements listing variables shared between this module and other modules in the program. (See Section 2.6.3, "Sharing Variables with Other Modules," for more information on using **COMMON** for this purpose.)

Every time you use the definition module in one of your programs, insert a **$INCLUDE** metacommand at the beginning of any module that invokes procedures in the definition module. When your program is compiled, the actual contents of the include file are substituted for the **$INCLUDE** metacommand.

A simple rule of thumb is to make an include file for every module and then use the module and the include file together as outlined above. The following list itemizes some of the benefits of this technique:

- A module containing procedure definitions remains truly modular—that is, you don't have to copy all the **DECLARE** statements for its procedures every time you call them from another module; instead, you can just substitute one **$INCLUDE** metacommand.
- In QuickBASIC, using an include file for procedure declarations suppresses automatic generation of **DECLARE** statements when you save a program.
- Using an include file for declarations avoids problems with getting one module to recognize a **FUNCTION** in another module. (See Section 2.5.4.1, "When QuickBASIC Does Not Generate a DECLARE Statement," for more information.)

You can take advantage of QuickBASIC's facility for generating **DECLARE** statements when creating your include file. The following steps show you how to do this:

1. Create your module.
2. Within that module, call any **SUB** or **FUNCTION** procedures you have defined.
3. Save the module to get automatic **DECLARE** statements for all the procedures.
4. Reedit the module, removing the procedure calls and moving the **DECLARE** statements to a separate include file.

See Appendix F, "Metacommands," for more information on the syntax and usage of the **$INCLUDE** metacommand.

Example

The following fragments illustrate how to use a definition module and an include file together:

```
' ========================================================
'                          MODDEF.BAS
' This module contains definitions for the PROMPTER and
' MAX! procedures.
' ========================================================

FUNCTION Max! (X!, Y!) STATIC
   IF X! > Y! THEN Max! = X! ELSE Max! = Y!
END FUNCTION
```

```
SUB Prompter (Row%, Column%, RecVar AS RecType) STATIC
   LOCATE Row%, Column%
   INPUT "Description: ", RecVar.Description
   INPUT "Quantity:    ", RecVar.Quantity
END SUB

' ============================================================
'                          MODDEF.BI
'   This is an include file that contains DECLARE statements
'   for the PROMPTER and MAX! procedures (as well as a TYPE
'   statement defining the RecType user type). Use this file
'   whenever you use the MODDEF.BAS module.
' ============================================================

TYPE RecType
   Description AS STRING * 15
   Quantity AS INTEGER
END TYPE

DECLARE FUNCTION Max! (X!, Y!)
DECLARE SUB Prompter (Row%, Column%, RecVar AS RecType)

'
============================================================
'                          SAMPLE.BAS
'    This module is linked with the MODDEF.BAS module, and
'    calls the PROMPTER and MAX! procedures in MODDEF.BAS.
============================================================

' The next line makes the contents of the MODDEF.BI include
' file part of this module as well:
' $INCLUDE: 'MODDEF.BI'
.
.
.
INPUT A, B
PRINT Max!(A, B)         ' Call the Max! FUNCTION in MODDEF.BAS
.
.
.
Prompter 5, 5, RecVar    ' Call the Prompter SUB in MODDEF.BAS
.
.
.
```

IMPORTANT *While it is good programming practice to put procedure declarations in an include file, do not put the procedures themselves (**SUB...END SUB** or **FUNCTION...END FUNCTION** blocks) in an include file. Procedure definitions are not allowed inside include files in QuickBASIC Version 4.5. If you have used include files to define **SUB** procedures in programs written with QuickBASIC Versions 2.0 or 3.0, either put these definitions in a separate module or incorporate them into the module where they are called.*

2.5.4.4 Declaring Procedures in Quick Libraries

A convenient programming practice is to put all the declarations for procedures in a Quick library into one include file. With the **$INCLUDE** metacommand you can then incorporate this include file into programs using the library. This saves you the trouble of copying all the relevant **DECLARE** statements every time you use the library.

2.5.5 Passing Arguments by Reference

By default, variables—whether simple scalar variables, arrays and array elements, or records—are passed "by reference" to **FUNCTION** and **SUB** procedures. Here is what is meant by passing variables by reference:

- Each program variable has an address, or a location in memory where its value is stored.
- Calling a procedure and passing variables to it by reference calls the procedure and passes it the address of each variable. Thus, the address of the variable and the address of its corresponding parameter in the procedure are one and the same.
- Therefore, if the procedure modifies the value of the parameter, it also modifies the value of the variable that is passed.

If you do not want a procedure to change the value of a variable, pass the procedure the value contained in the variable, not the address. This way, changes are made only to a copy of the variable, not the variable itself. See the next section for a discussion of this alternative way of passing variables.

Example

In the following program, changes made to the parameter `A$` in the `Replace` procedure also change the argument `Test$`:

```
Test$ = "a string with all lowercase letters."
PRINT "Before subprogram call: "; Test$
CALL Replace (Test$, "a")
PRINT "After subprogram call: "; Test$
END

SUB Replace (A$, B$) STATIC
   Start = 1
   DO

      ' Look for B$ in A$, starting at the character
      ' with position "Start" in A$:
      Found = INSTR(Start, A$, B$)
```

```
      ' Make every occurrence of B$ in A$
      ' an uppercase letter:
      IF Found > 0 THEN
         MID$(A$,Found) = UCASE$(B$)
         Start = Start + 1
      END IF
   LOOP WHILE Found > 0
END SUB
```

Output

```
Before subprogram call: a string with all lowercase letters.
After subprogram call: A string with All lowercAse letters.
```

2.5.6 Passing Arguments by Value

Passing an argument "by value" means the value of the argument is passed, rather than its address. In BASIC procedures, passing a variable by value is simulated by copying the variable into a temporary location, then passing the address of this temporary location. Since the procedure does not have access to the address of the original variable, it cannot change the original variable; it makes all changes to the copy instead.

You can pass expressions as arguments to procedures, as in the following:

```
' A + B is an expression; the values of A and B
' are not affected by this procedure call:
CALL Mult(A + B, B)
```

Expressions are always passed by value (see Section 2.5.2, "Passing Constants and Expressions," for more information).

Example

One way to pass a variable by value is to enclose it in parentheses, thus making it an expression. As you can see from the output that follows, changes to the **SUB** procedure's local variable `Y` are passed back to the module-level code as changes to the variable `B`. However, changes to `X` in the procedure do not affect the value of `A`, since `A` is passed by value.

```
A = 1
B = 1
PRINT "Before subprogram call, A ="; A; ", B ="; B

' A is passed by value, and B is passed by reference:
CALL Mult((A), B)
PRINT "After subprogram call, A ="; A; ", B ="; B
END
```

```
SUB Mult (X, Y) STATIC
   X = 2 * X
   Y = 3 * Y
   PRINT "In subprogram, X ="; X; ", Y ="; Y
END SUB
```

Output

```
Before subprogram call, A = 1 , B = 1
In subprogram, X = 2 , Y = 3
After subprogram call, A = 1 , B = 3
```

2.6 Sharing Variables with SHARED

In addition to passing variables through argument and parameter lists, procedures can also share variables with other procedures and with code at the module level (that is, code within a module but outside of any procedure) in one of the two ways listed below:

1. Variables listed in a **SHARED** statement within a procedure are shared only between that procedure and the module-level code. Use this method when different procedures in the same module need different combinations of module-level variables.
2. Variables listed in a module-level **COMMON SHARED**, **DIM SHARED**, or **REDIM SHARED** statement are shared between the module-level code and all procedures within that module. This method is most useful when all procedures in a module use a common set of variables.

You can also use the **COMMON** or **COMMON SHARED** statement to share variables among two or more modules. Sections 2.6.1–2.6.3 discuss these three ways to share variables.

2.6.1 Sharing Variables with Specific Procedures in a Module

If different procedures within a module need to share different variables with the module-level code, use the **SHARED** statement within each procedure.

Arrays in **SHARED** statements consist of the array name followed by a set of empty parentheses ( ):

```
SUB JustAnotherSub STATIC
   SHARED ArrayName ()
   .
   .
   .
```

If you give a variable its type in an **AS** *type* clause, then the variable must also be typed with the **AS** *type* clause in a **SHARED** statement:

```
DIM Buffer AS STRING * 10
.
.
.
END

SUB ReadRecords STATIC
   SHARED Buffer AS STRING * 10
   .
   .
   .
END SUB
```

Example

In the next example, the **SHARED** statements in the `GetRecords` and `InventoryTotal` procedures show the format of a shared variable list:

```
' ============================================================
'                     MODULE-LEVEL CODE
' ============================================================
TYPE RecType
   Price AS SINGLE
   Desc AS STRING * 35
END TYPE

DIM RecVar(1 TO 100) AS RecType    ' Array of records

INPUT "File name: ", FileSpec$
CALL GetRecords
PRINT InventoryTotal
END

' ============================================================
'                     PROCEDURE-LEVEL CODE
' ============================================================
SUB GetRecords STATIC

' Both FileSpec$ and the RecVar array of records
' are shared with the module-level code above:
   SHARED FileSpec$, RecVar() AS RecType
   OPEN FileSpec$ FOR RANDOM AS #1
   .
   .
   .
END SUB
```

```
FUNCTION InventoryTotal STATIC

' Only the RecVar array is shared with the module-level
' code:
   SHARED RecVar() AS RecType
   .
   .
   .
END FUNCTION
```

2.6.2 Sharing Variables with All Procedures in a Module

If variables are declared at the module level with the **SHARED** attribute in a **COMMON**, **DIM**, or **REDIM** statement (for example, by using a statement of the form **COMMON SHARED** *variablelist*), then all procedures within that module have access to those variables; in other words, the **SHARED** attribute makes variables global throughout a module.

The **SHARED** attribute is convenient when you need to share large numbers of variables among all procedures in a module.

Examples

These statements declare variables shared among all procedures in one module:

```
COMMON SHARED A, B, C
DIM SHARED Array(1 TO 10, 1 TO 10) AS UserType
REDIM SHARED Alpha(N%)
```

In the following example, the module-level code shares the string array `StrArray` and the integer variables `Min` and `Max` with the two **SUB** procedures `FillArray` and `PrintArray`:

```
' ========================================================
'                      MODULE-LEVEL CODE
' ========================================================
DECLARE SUB FillArray ()
DECLARE SUB PrintArray ()

' The following DIM statements share the Min and Max
' integer variables and the StrArray string array
' with any SUB or FUNCTION in this module:
DIM SHARED StrArray (33 TO 126) AS STRING * 5
DIM SHARED Min AS INTEGER, Max AS INTEGER

Min = LBOUND(StrArray)
Max = UBOUND(StrArray)

FillArray               ' Note the absence of argument lists.
PrintArray
END
```

```
' ============================================================
'                      PROCEDURE-LEVEL CODE
' ============================================================
SUB FillArray STATIC

   ' Load each element of the array from 33 to 126
   ' with a 5-character string, each character of which
   ' has the ASCII code I%:
   FOR I% = Min TO Max
      StrArray(I%) = STRING$(5, I%)
   NEXT

END SUB

SUB PrintArray STATIC
   FOR I% = Min TO Max
      PRINT StrArray(I%)
   NEXT
END SUB
```

Partial output

```
!!!!!
"""""
#####
$$$$$
%%%%%
&&&&&
'''''
(((((
.
.
.
```

If you are using your own text editor to write your programs and directly compiling those programs outside the QuickBASIC development environment, note that variable declarations with the **SHARED** attribute must precede the procedure definition. Otherwise, the value of any variable declared with **SHARED** is not available to the procedure, as shown by the output from the next example. (If you are using QuickBASIC to create your programs, this sequence is not required, since QuickBASIC automatically saves programs in the correct order.)

```
DEFINT A-Z

FUNCTION Adder (X, Y) STATIC
   Adder = X + Y + Z
END FUNCTION

DIM SHARED Z
Z = 2
PRINT Adder (1, 3)
END
```

Output

```
 4
```

The next example shows how you should save the module shown above, with the definition of `Adder` following the `DIM SHARED Z` statement:

```
DEFINT A-Z

DECLARE FUNCTION Adder (X, Y)

' The variable Z is now shared with Adder:
DIM SHARED Z
Z = 2
PRINT Adder (1, 3)
END

FUNCTION Adder (X, Y) STATIC
   Adder = X + Y + Z
END FUNCTION
```

Output

```
 6
```

2.6.3 Sharing Variables with Other Modules

If you want to share variables across modules in your program, list the variables in **COMMON** or **COMMON SHARED** statements at the module level in each module.

Examples

The following example shows how to share variables between modules by using a **COMMON** statement in the module that calls the **SUB** procedures, as well as a **COMMON SHARED** statement in the module that defines the procedures. With **COMMON SHARED**, all procedures in the second module have access to the common variables:

```
' ============================================================
'                        MAIN MODULE
' ============================================================

COMMON A, B
A = 2.5
B = 1.2
CALL Square
CALL Cube
END
```

```
' ============================================================
'          Module with Cube and Square Procedures
' ============================================================

' NOTE: The names of the variables (X, Y) do not have to be
' the same as in the other module (A, B). Only the types
' have to be the same.

COMMON SHARED X, Y  ' This statement is at the module level.
                    ' Both X and Y are shared with the CUBE
                    ' and SQUARE procedures below.
SUB Cube STATIC
   PRINT "A cubed   ="; X ^ 3
   PRINT "B cubed   ="; Y ^ 3
END SUB

SUB Square STATIC
   PRINT "A squared ="; X ^ 2
   PRINT "B squared ="; Y ^ 2
END SUB
```

The following example uses named COMMON blocks at the module levels and SHARED statements within procedures to share different sets of variables with each procedure:

```
' ============================================================
'                          MAIN MODULE
' Prints the volume and density of a filled cylinder given
' the input values
' ============================================================

COMMON /VolumeValues/ Height, Radius, Volume
COMMON /DensityValues/ Weight, Density

INPUT "Height of cylinder in centimeters: ", Height
INPUT "Radius of cylinder in centimeters: ", Radius
INPUT "Weight of filled cylinder in grams: ", Weight

CALL VolumeCalc
CALL DensityCalc

PRINT "Volume is"; Volume; "cubic centimeters."
PRINT "Density is"; Density; "grams/cubic centimeter."
END

' ============================================================
'     Module with DensityCalc and VolumeCalc Procedures
' ============================================================

COMMON /VolumeValues/ H, R, V
COMMON /DensityValues/ W, D
```

```
SUB VolumeCalc STATIC

   ' Share the Height, Radius, and Volume variables
   ' with this procedure:
   SHARED H, R, V
   CONST PI = 3.141592653589#
   V = PI * H * (R ^ 2)
END SUB

SUB DensityCalc STATIC

   ' Share the Weight, Volume, and Density variables
   ' with this procedure:
   SHARED W, V, D
   D = W / V
END SUB
```

Output

```
Height of cylinder in centimeters: 100
Radius of cylinder in centimeters: 10
Weight of filled cylinder in grams: 10000
Volume is 31415.93 cubic centimeters.
Density is .3183099 grams/cubic centimeter.
```

2.6.4 The Problem of Variable Aliasing

"Variable aliasing" is sometimes a problem in long programs containing many variables and procedures. Variable aliasing is the situation where two or more names refer to the same location in memory. It occurs:

- When the same variable appears more than once in the list of arguments passed to a procedure.
- When a variable passed in an argument list is also accessed by the procedure by means of the **SHARED** statement or the **SHARED** attribute.

To avoid aliasing problems, double-check variables shared with a procedure to make sure they don't also appear in a procedure call's argument list. Also, don't pass the same variable twice, as in the next statement:

```
' X is passed twice; this will lead to aliasing problems
' in the Test procedure:
CALL Test(X, X, Y)
```

Example

The following example illustrates how variable aliasing can occur. Here the variable `A` is shared between the module-level code and the **SUB** procedure with the `DIM SHARED` statement. However, `A` is also passed by reference to the

SUB as an argument. Therefore, in the subprogram, A and X both refer to the same location in memory. Thus, when the subprogram modifies X, it is also modifying A, and vice versa.

```
DIM SHARED A
A = 4
CALL PrintHalf(A)
END

SUB PrintHalf (X) STATIC
   PRINT "Half of"; X; "plus half of"; A; "equals";
   X = X / 2              ' X and A now both equal 2.
   A = A / 2              ' X and A now both equal 1.
   PRINT A + X
END SUB
```

Output

```
Half of 4 plus half of 4 equals 2
```

2.7 Automatic and STATIC Variables

When the **STATIC** attribute appears on a procedure-definition line, it means that local variables within the procedure are **STATIC**; that is, their values are preserved between calls to the procedure.

Leaving off the **STATIC** attribute makes local variables within the procedure "automatic" by default; that is, you get a fresh set of local variables each time the procedure is called.

You can override the effect of leaving off the **STATIC** attribute by using the **STATIC** statement within the procedure, thus making some variables automatic and others **STATIC** (see Section 2.8 for more information).

NOTE *The **SHARED** statement also overrides the default for variables in a procedure (local **STATIC** or local automatic), since any variable appearing in a **SHARED** statement is known at the module level and thus is not local to the procedure.*

2.8 Preserving Values of Local Variables with the STATIC Statement

Sometimes you may want to make some local variables in a procedure **STATIC** while keeping the rest automatic. List those variables in a **STATIC** statement within the procedure.

Also, putting a variable name in a **STATIC** statement is a way of making absolutely sure that the variable is local, since a **STATIC** statement overrides the effect of a module-level **SHARED** statement.

A **STATIC** statement can appear only within a procedure. An array name in a **STATIC** statement must be followed by a set of empty parentheses ( ). Also, you must dimension any array that appears in a **STATIC** statement before using the array, as shown in the next example:

```
SUB SubProg2
   STATIC Array() AS INTEGER
   DIM Array(-5 TO 5, 1 TO 25) AS INTEGER
   .
   .
   .
END SUB
```

NOTE *If you give a variable its type in an* ***AS*** *type clause, then the* ***AS*** *type clause must appear along with the variable's name in both the* ***STATIC*** *and* ***DIM*** *statements.*

Example

The following example shows how a **STATIC** statement preserves the value of the string variable `Y$` throughout successive calls to `TestSub`:

```
DECLARE SUB TestSub ()
FOR I% = 1 TO 5
   TestSub                ' Call TestSub five times.
NEXT I%
END

SUB TestSub               ' Note: no STATIC attribute

  ' Both X$ and Y$ are local variables in TestSub (that is,
  ' their values are not shared with the module-level code).
  ' However since X$ is an automatic variable, it is
  ' reinitialized to a null string every time TestSub is
  ' called. In contrast, Y$ is STATIC, so it retains the
  ' value it had from the last call:
   STATIC Y$
   X$ = X$ + "*"
   Y$ = Y$ + "*"
   PRINT X$, Y$
END SUB
```

Output

```
*              *
*              **
*              ***
*              ****
*              *****
```

2.9 Recursive Procedures

Procedures in BASIC can be recursive. A recursive procedure is one that can call itself or call other procedures that in turn call the first procedure.

2.9.1 The Factorial Function

A good way to illustrate recursive procedures is to consider the factorial function from mathematics. One way to define n! ("n factorial") is with the following formula:

n! = n * (n–1) * (n–2) * ... * 2 * 1

For example, 5 factorial is evaluated as follows:

5! = 5 * 4 * 3 * 2 * 1 = 120

NOTE *Do not confuse the mathematical factorial symbol (!) used in this discussion with the single-precision type-declaration suffix used by BASIC.*

Factorials lend themselves to a recursive definition as well:

n! = n * (n–1)!

This leads to the following progression:

5! = 5 * 4!
4! = 4 * 3!
3! = 3 * 2!
2! = 2 * 1!
1! = 1 * 0!

Recursion must always have a terminating condition. With factorials this terminating condition occurs when 0! is evaluated—by definition, 0! is equal to 1.

NOTE *Although a recursive procedure can have* ***STATIC*** *variables by default (as in the next example), it is often preferable to let automatic variables be the default instead. In this way, recursive calls will not overwrite variable values from a preceding call.*

Example

The following example uses a recursive **FUNCTION** procedure to calculate factorials:

```
DECLARE FUNCTION Factorial# (N%)
Format$ = "###_! = ####################"
DO
   INPUT "Enter number from 0 - 20 (or -1 to end): ", Num%
   IF Num% >= 0 AND Num% <= 20 THEN
      PRINT USING Format$; Num%; Factorial#(Num%)
   END IF
LOOP WHILE Num% >= 0
END

FUNCTION Factorial# (N%) STATIC

   IF N% > 0 THEN  ' Call Factorial# again
                   ' if N is greater than zero.
      Factorial# = N% * Factorial#(N% - 1)

   ELSE            ' Reached the end of recursive calls
                   ' (N% = 0), so "climb back up the ladder."
      Factorial# = 1
   END IF
END FUNCTION
```

2.9.2 Adjusting the Size of the Stack

Recursion can eat up a lot of memory, since each set of automatic variables in a **SUB** or **FUNCTION** procedure is saved on the stack. (Saving variables this way allows a procedure to continue with the correct variable values after control returns from a recursive call.)

If you have a recursive procedure with many automatic variables, or a deeply nested recursive procedure, you may need to adjust the size of the stack with a **CLEAR , ,** *stacksize* statement, where *stacksize* is the number of bytes from the stack you want to reserve. Otherwise, while your program is running you may get the error message `Out of stack space`.

The following steps outline one way to estimate the amount of memory a recursive procedure needs:

1. Insert a single quotation mark to temporarily turn the recursive call into a comment line so that the procedure will be invoked only once when the program runs.
2. Call the **FRE(–2)** function (which returns the total unused stack space) just before you call the recursive procedure. Also call the **FRE(–2)** function right at the end of the recursive procedure. Use **PRINT** statements to display the returned values.

3. Run the program. The difference in values is the amount of stack space (in bytes) used by one call to the procedure.
4. Estimate the maximum number of times the procedure is likely to be invoked, then multiply this value by the stack space consumed by one call to the procedure. The result is *totalbytes*.
5. Reserve the amount of stack space calculated in step 4:

   CLEAR , , *totalbytes*

2.10 Transferring Control to Another Program with CHAIN

Unlike procedure calls, which occur within the same program, the **CHAIN** statement simply starts a new program. When a program chains to another program, the following sequence occurs:

1. The first program stops running.
2. The second program is loaded into memory.
3. The second program starts running.

The advantage of using **CHAIN** is that it enables you to split a program with large memory requirements into several smaller programs.

The **COMMON** statement allows you to pass variables from one program to another program in a chain. A common programming practice is to put these **COMMON** statements in an include file, and then use the **$INCLUDE** metacommand at the beginning of each program in the chain.

NOTE *Don't use a* **COMMON** */blockname /variablelist statement (a "named* **COMMON** *block") to pass variables to a chained program, since variables listed in named* **COMMON** *blocks are not preserved when chaining. Use a blank* **COMMON** *block (***COMMON** *variablelist) instead.*

Example

This example, which shows a chain connecting three separate programs, uses an include file to declare variables passed in common among the programs:

```
' ============ CONTENTS OF INCLUDE FILE COMMONS.BI ========
DIM Values(10)
COMMON Values(), NumValues

' ======================== MAIN.BAS ========================

' Read in the contents of the COMMONS.BI file:
' $INCLUDE: 'COMMONS.BI'

   ' Input the data:
   INPUT "Enter number of data values (<=10): ", NumValues
   FOR I = 1 TO NumValues
      Prompt$ = "Value ("+LTRIM$(STR$(I))+")? "
      PRINT Prompt$;
      INPUT "", Values(I)
   NEXT I

   ' Have the user specify the calculation to do:
   INPUT "Calculation (1=st. dev., 2=mean)? ", Choice

   ' Now, chain to the correct program:
   SELECT CASE Choice

      CASE 1:           ' Standard Deviation
         CHAIN "STDEV"

      CASE 2:           ' Mean
         CHAIN "MEAN"
   END SELECT
END

' ======================== STDEV.BAS ========================
' Calculates the standard deviation of a set of data
' ===========================================================

' $INCLUDE: 'COMMONS.BI'

   Sum   = 0   ' Normal sum
   SumSq = 0   ' Sum of values squared

   FOR I = 1 TO NumValues
      Sum   = Sum   + Values(I)
      SumSq = SumSq + Values(I) ^ 2
   NEXT I

   Stdev = SQR(SumSq / NumValues - (Sum / NumValues) ^ 2)
   PRINT "The Standard Deviation of the samples is: " Stdev
END
```

```
' ========================= MEAN.BAS ========================
' Calculates the mean (average) of a set of data
' ===========================================================

' $INCLUDE: 'COMMONS.BI'

   Sum = 0

   FOR I = 1 TO NumValues
      Sum = Sum + Values(I)
   NEXT

   Mean = Sum / NumValues
   PRINT "The mean of the samples is: " Mean
END
```

2.11 Sample Application: Recursive Directory Search (WHEREIS.BAS)

The following program uses a recursive **SUB** procedure, `ScanDir`, to scan a disk for the file name input by the user. Each time this program finds the given file, it prints the complete directory path to the file.

Statements and Functions Used

This program demonstrates the following statements and keywords discussed in this chapter:

- **DECLARE**
- **FUNCTION...END FUNCTION**
- **STATIC**
- **SUB...END SUB**

Program Listing

```
DEFINT A-Z

' Declare symbolic constants used in program:
CONST EOFTYPE = 0, FILETYPE = 1, DIRTYPE = 2, ROOT = "TWH"

DECLARE SUB ScanDir (PathSpec$, Level, FileSpec$, Row)

DECLARE FUNCTION MakeFileName$ (Num)
DECLARE FUNCTION GetEntry$ (FileNum, EntryType)
```

```
CLS
INPUT "File to look for"; FileSpec$
PRINT
PRINT "Enter the directory where the search should start"
PRINT "(optional drive + directories). Press <ENTER> to "
PRINT "begin search in root directory of current drive."
PRINT
INPUT "Starting directory"; PathSpec$
CLS

RightCh$ = RIGHT$(PathSpec$, 1)

IF PathSpec$ = "" OR RightCh$ = ":" OR RightCh$ <> "\" THEN
   PathSpec$ = PathSpec$ + "\"
END IF

FileSpec$ = UCASE$(FileSpec$)
PathSpec$ = UCASE$(PathSpec$)
Level = 1
Row = 3

' Make the top level call (level 1) to begin the search:
ScanDir PathSpec$, Level, FileSpec$, Row

KILL ROOT + ".*"        ' Delete all temporary files created
                        ' by the program.

LOCATE Row + 1, 1: PRINT "Search complete."
END

' ======================== GETENTRY ========================
'    This procedure processes entry lines in a DIR listing
'    saved to a file.

'    This procedure returns the following values:

'       GetEntry$       A valid file or directory name
'       EntryType       If equal to 1, then GetEntry$
'                       is a file.
'                       If equal to 2, then GetEntry$
'                       is a directory.
' ==========================================================
```

```
FUNCTION GetEntry$ (FileNum, EntryType) STATIC

   ' Loop until a valid entry or end-of-file (EOF) is read:
   DO UNTIL EOF(FileNum)
      LINE INPUT #FileNum, EntryLine$
      IF EntryLine$ <> "" THEN

         ' Get first character from the line for test:
         TestCh$ = LEFT$(EntryLine$, 1)
         IF TestCh$ <> " " AND TestCh$ <> "." THEN EXIT DO
      END IF
   LOOP

   ' Entry or EOF found, decide which:
   IF EOF(FileNum) THEN          ' EOF, so return EOFTYPE
      EntryType = EOFTYPE        ' in EntryType.
      GetEntry$ = ""

   ELSE                          ' Not EOF, so it must be a
                                 ' file or a directory.

      ' Build and return the entry name:
      EntryName$ = RTRIM$(LEFT$(EntryLine$, 8))

      ' Test for extension and add to name if there is one:
      EntryExt$ = RTRIM$(MID$(EntryLine$, 10, 3))
      IF EntryExt$ <> "" THEN
         GetEntry$ = EntryName$ + "." + EntryExt$
      ELSE
         GetEntry$ = EntryName$
      END IF

      ' Determine the entry type, and return that value
      ' to the point where GetEntry$ was called:
      IF MID$(EntryLine$, 15, 3) = "DIR" THEN
         EntryType = DIRTYPE            ' Directory
      ELSE
         EntryType = FILETYPE           ' File
      END IF

   END IF

END FUNCTION
```

```
' ===================== MAKEFILENAME$ ====================
'    This procedure makes a file name from a root string
'    ("TWH," defined as a symbolic constant at the module
'    level) and a number passed to it as an argument (Num).
' ========================================================

FUNCTION MakeFileName$ (Num) STATIC

   MakeFileName$ = ROOT + "." + LTRIM$(STR$(Num))

END FUNCTION

' ====================== SCANDIR ========================
'   This procedure recursively scans a directory for the
'   file name entered by the user.

'   NOTE: The SUB header doesn't use the STATIC keyword
'         since this procedure needs a new set of variables
'         each time it is invoked.
' ========================================================

SUB ScanDir (PathSpec$, Level, FileSpec$, Row)

   LOCATE 1, 1: PRINT "Now searching"; SPACE$(50);
   LOCATE 1, 15: PRINT PathSpec$;

   ' Make a file specification for the temporary file:
   TempSpec$ = MakeFileName$(Level)

   ' Get a directory listing of the current directory,
   ' and save it in the temporary file:
   SHELL "DIR " + PathSpec$ + " > " + TempSpec$

   ' Get the next available file number:
   FileNum = FREEFILE

   ' Open the DIR listing file and scan it:
   OPEN TempSpec$ FOR INPUT AS #FileNu
```

```
' Process the file, one line at a time:
   DO

      ' Input an entry from the DIR listing file:
      DirEntry$ = GetEntry$(FileNum, EntryType)

      ' If entry is a file:
      IF EntryType = FILETYPE THEN

         ' If the FileSpec$ string matches,
         ' print entry and exit this loop:
         IF DirEntry$ = FileSpec$ THEN
            LOCATE Row, 1: PRINT PathSpec$; DirEntry$;
            Row = Row + 1
            EntryType = EOFTYPE
         END IF

      ' If the entry is a directory, then make a recursive
      ' call to ScanDir with the new directory:
      ELSEIF EntryType = DIRTYPE THEN
         NewPath$ = PathSpec$ + DirEntry$ + "\"
         ScanDir NewPath$, Level + 1, FileSpec$, Row
         LOCATE 1, 1: PRINT "Now searching"; SPACE$(50);
         LOCATE 1, 15: PRINT PathSpec$;
      END IF

   LOOP UNTIL EntryType = EOFTYPE

   ' Scan on this DIR listing file is finished, so close it:
   CLOSE FileNum
END SUB
```

CHAPTER 3

File and Device I/O

3

This chapter shows you how to use BASIC input and output (I/O) functions and statements. These statements permit your programs to access data stored in files and to communicate with devices attached to your system.

The chapter includes material on a variety of programming tasks related to retrieving, storing, and formatting information. The relationship between data files and physical devices such as screens and keyboards is also covered.

When you are finished with this chapter, you will know how to perform the following programming tasks:

- Print text on the screen
- Get input from the keyboard for use in a program
- Create data files on disk
- Store records in data files
- Read records from data files
- Read or modify data in files that are not in American Standard Code for Information Interchange (ASCII) format
- Communicate with other computers through the serial port

3.1 Printing Text on the Screen

This section explains how to accomplish the following tasks:

- Display text on the screen with **PRINT**
- Display formatted text on the screen with **PRINT USING**
- Skip spaces in a row of printed text with **SPC**
- Skip to a given column in a row of printed text with **TAB**
- Change the number of rows or columns appearing on the screen with **WIDTH**
- Open a text viewport with **VIEW PRINT**

NOTE *Output that appears on the screen is sometimes referred to as "standard output." You can redirect standard output by using the DOS command-line symbols > or >>, thus sending output that would have gone to the screen to a different output device (such as a printer) or to a disk file. (See your DOS documentation for more information on redirecting output.)*

3.1.1 Screen Rows and Columns

To understand how text is printed on the screen, it helps to think of the screen as a grid of "rows" and "columns." The height of one row slightly exceeds the height of a line of printed output, while the width of one column is just wider than the width of one character. A standard screen configuration in text mode (nongraphics) is 80 columns wide by 25 rows high. Figure 3.1 shows how each character printed on the screen occupies a unique cell in the grid, a cell that can be identified by pairing a row argument with a column argument.

The bottom row of the screen is not usually used for output, unless you use a **LOCATE** statement to display text there. (See Section 3.3, "Controlling the Text Cursor," for more information on **LOCATE**.)

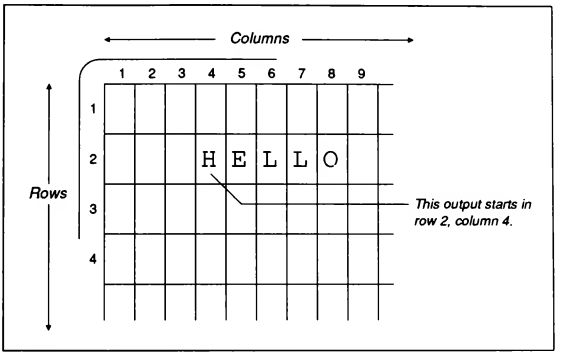


Figure 3.1 Text Output on Screen

3.1.2 Displaying Text and Numbers with PRINT

By far the most commonly used statement for output to the screen is the **PRINT** statement. With **PRINT**, you can display numeric or string values, or a mixture of the two. In addition, **PRINT** with no arguments prints a blank line.

The following are some general comments about **PRINT**:

- **PRINT** always prints numbers with a trailing blank space. If the number is positive, the number is also preceded by a space; if the number is negative, the number is preceded by a minus sign.
- The **PRINT** statement can be used to print lists of expressions. Expressions in the list can be separated from other expressions by commas, semicolons, one or more blank spaces, or one or more tab characters. A comma causes **PRINT** to skip to the beginning of the next "print zone," or block of 14 columns, on the screen. A semicolon (or any combination of spaces and/or tabs) between two expressions prints the expressions on the screen next to each other, with no spaces in between (except for the built-in spaces for numbers).

- Ordinarily, **PRINT** ends each line of output with a new-line sequence (a carriage-return and line-feed). However, a comma or semicolon at the end of the list of expressions suppresses this; the next printed output from the program appears on the same line unless it is too long to fit on that line.
- **PRINT** wraps an output line that exceeds the width of the screen onto the next line. For example, if you try to print a line that is 100 characters long on an 80-column screen, the first 80 characters of the line show up on one row, followed by the next 20 characters on the next row. If the 100-character line didn't start at the left edge of the screen (for example, if it followed a **PRINT** statement ending in a comma or semicolon), then the line would print until it reached the 80th column of one row and continue in the first column of the next row.

Example

The output from the following program shows some of the different ways you can use **PRINT**:

```
A = 2
B = -1
C = 3
X$ = "over"
Y$ = "there"

PRINT A, B, C
PRINT B, A, C
PRINT A; B; C
PRINT X$; Y$
PRINT X$, Y$;
PRINT A, B
PRINT
FOR I = 1 TO 8
   PRINT X$,
NEXT
```

Output

```
 2             -1             3
-1              2             3
 2 -1  3
overthere
over          there 2        -1

over          over           over          over          over
over          over           over
```

3.1.3 Displaying Formatted Output with PRINT USING

The **PRINT USING** statement gives greater control than **PRINT** over the appearance of printed data, especially numeric data. Through the use of special characters embedded in a format string, **PRINT USING** allows you to specify information such as how many digits from a number (or how many characters from a string) are displayed, whether or not a plus (+) sign or a dollar sign ($) appears in front of a number, and so forth.

Example

The example that follows gives you a sample of what can be done with **PRINT USING**. You can list more than one expression after the **PRINT USING** format string. As is the case with **PRINT**, the expressions in the list can be separated from one another by commas, semicolons, spaces, or tab characters.

```
X = 441.2318

PRINT USING "The number with 3 decimal places ###.###";X
PRINT USING "The number with a dollar sign $$##.##";X
PRINT USING "The number in exponential format #.###^^^^";X
PRINT USING "Numbers with plus signs +###  "; X; 99.9
```

Output

```
The number with 3 decimal places 441.232
The number with a dollar sign $441.23
The number in exponential format 0.441E+03
Numbers with plus signs +441  Numbers with plus signs +100
```

Consult the QB Advisor for more on **PRINT USING**.

3.1.4 Skipping Spaces and Advancing to a Specific Column

By using the **SPC(*n*)** statement in a **PRINT** statement, you can skip *n* spaces in a row of printed output, as shown by the output from the next example:

```
PRINT "         1         2         3"
PRINT "123456789012345678901234567890"
PRINT "First Name"; SPC(10); "Last Name"
```

Output

```
1         2         3
123456789012345678901234567890
First Name          Last Name
```

By using the **TAB**(*n*) statement in a **PRINT** statement, you can skip to the *n*th column (counting from the left side of the screen) in a row of printed output. The following example uses **TAB** to produce the same output as that shown above:

```
PRINT "         1         2         3"
PRINT "123456789012345678901234567890"
PRINT "First Name"; TAB(21); "Last Name"
```

Neither **SPC** nor **TAB** can be used by itself to position printed output on the screen; they can only appear in **PRINT** statements.

3.1.5 Changing the Number of Columns or Rows

You can control the maximum number of characters that appear in a single row of output by using the **WIDTH** *columns* statement. The **WIDTH** *columns* statement actually changes the size of characters that are printed on the screen, so that more or fewer characters can fit on a row. For example, **WIDTH 40** makes characters wider, so the maximum row length is 40 characters. **WIDTH 80** makes characters narrower, so the maximum row length is 80 characters. The numbers 40 and 80 are the only valid values for the *columns* argument.

On machines equipped with an Enhanced Graphics Adapter (EGA) or Video Graphics Adapter (VGA), the **WIDTH** statement can also control the number of rows that appear on the screen as follows:

WIDTH [[*columns*]] [[, *rows*]]

The value for *rows* may be 25, 30, 43, 50, or 60, depending on the type of display adapter you use and the screen mode set in a preceding **SCREEN** statement.

3.1.6 Creating a Text Viewport

So far, the entire screen has been used for text output. However, with the **VIEW PRINT** statement, you can restrict printed output to a "text viewport," a horizontal slice of the screen. The syntax of the **VIEW PRINT** statement is:

VIEW PRINT [[*topline* **TO** *bottomline*]]

The values for *topline* and *bottomline* specify the locations where the viewport will begin and end.

A text viewport also gives you control over on-screen scrolling. Without a viewport, once printed output reaches the bottom of the screen, text or graphics output that was at the top of the screen scrolls off and is lost. However, after a **VIEW PRINT** statement, scrolling takes place only between the top and bottom lines of the viewport. This means you can label the displayed output at the top and/or bottom of the screen without having to worry that the labeling will scroll it off if too many lines of data appear. You can also use the **CLS 2** statement to

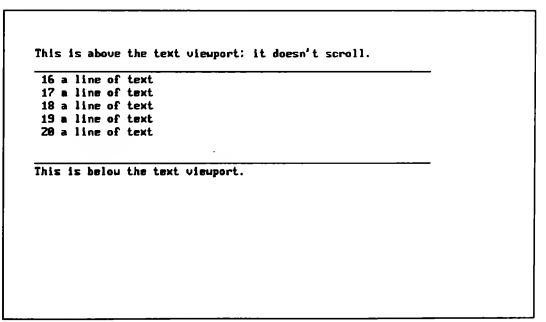
clear just the text viewport, leaving the contents of the rest of the screen intact. See Section 5.5, "Defining a Graphics Viewport," to learn how to create a viewport for graphics output on the screen.

Example

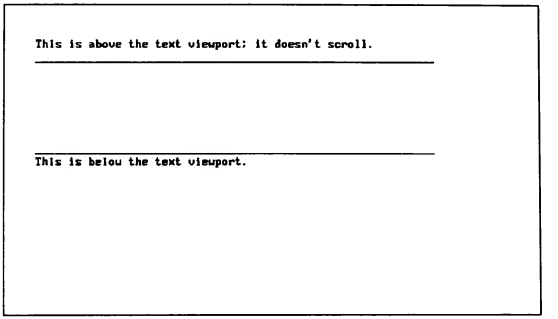
You can see the effects of a **VIEW PRINT** statement by examining the output from the next example:

```
CLS
LOCATE 3, 1
PRINT "This is above the text viewport; it doesn't scroll."

LOCATE 4, 1
PRINT STRING$(60, "_")        ' Print horizontal lines above
LOCATE 11, 1
PRINT STRING$(60, "_")        ' and below the text viewport.

PRINT "This is below the text viewport."

VIEW PRINT 5 TO 10            ' Text viewport extends from
                              ' lines 5 to 10.

FOR I = 1 TO 20               ' Print numbers and text in
   PRINT I; "a line of text" ' the viewport.
NEXT

DO: LOOP WHILE INKEY$ = ""    ' Wait for a key press.
CLS 2                         ' Clear just the viewport.
END
```

Output (Before User Presses Key)



Output (After User Presses Key)



3.2 Getting Input from the Keyboard

This section shows you how to use the following statements and functions to enable your BASIC programs to accept input entered from the keyboard:

- **INPUT**
- **LINE INPUT**
- **INPUT$**
- **INKEY$**

NOTE *Input typed at the keyboard is often referred to as "standard input." You can use the DOS symbol < to direct standard input to your program from a file or other input device instead of from the keyboard. (See your DOS documentation for more information on redirecting input.)*

3.2.1 The INPUT Statement

The **INPUT** statement takes information typed by the user and stores it in a list of variables, as shown in the following example:

```
INPUT A%, B, C$
INPUT D$
PRINT A%, B, C$, D$
```

Output

```
? 6.6,45,a string
? "two, three"
 7             45            a string       two, three
```

Here are some general comments about **INPUT**:

- An **INPUT** statement by itself prompts the user with a question mark (?) followed by a blinking cursor.
- The **INPUT** statement is followed by one or more variable names. If there are more than one variable, they are separated by commas.
- The number of constants entered by the user after the **INPUT** prompt must be the same as the number of variables in the **INPUT** statement itself.
- The values the user enters must agree in type with the variables in the list following **INPUT**. In other words, enter a number if the variable is designated as having the type integer, long integer, single precision, or double precision. Enter a string if the variable is designated as having the type string.
- Since constants in an input list must be separated by commas, an input string constant containing one or more commas should be enclosed in double quotes. The double quotes ensure that the string is treated as a unit and not broken into two or more parts.

If the user breaks any of the last three rules, BASIC prints the error message `Redo from start`. This message reappears until the input agrees in number and type with the variable list.

If you want your input prompt to be more informative than a simple question mark, you can make a prompt appear, as in the following example:

```
INPUT "What is the correct time (hour, min)"; Hr$, Min$
```

This prints the following prompt:

```
What is the correct time (hour, min)?
```

Note the semicolon between the prompt and the input variables. This semicolon causes a question mark to appear as part of the prompt. Sometimes you may want to eliminate the question mark altogether; in this case, put a comma between the prompt and the variable list:

```
INPUT "Enter the time (hour, min): ", Hr$, Min$
```

This prints the following prompt:

```
Enter the time (hour, min):
```

3.2.2 The LINE INPUT Statement

If you want your program to accept lines of text with embedded commas, leading blanks, or trailing blanks, yet you do not want to have to remind the user to enclose the input in double quotes, use the **LINE INPUT** statement. The **LINE INPUT** statement, as its name implies, accepts a line of input (terminated by pressing the ENTER key) from the keyboard and stores it in a single string variable. Unlike **INPUT**, the **LINE INPUT** statement does not print a question mark by default to prompt for input; it does, however, allow you to display a prompt string.

The following example shows the difference between **INPUT** and **LINE INPUT**:

```
' Assign the input to three separate variables:
INPUT "Enter three values separated by commas: ", A$, B$, C$

' Assign the input to one variable (commas not treated
' as delimiters between input):
LINE INPUT "Enter the same three values: ", D$

PRINT "A$ = "; A$
PRINT "B$ = "; B$
PRINT "C$ = "; C$
PRINT "D$ = "; D$
```

Output

```
Enter 3 values separated by commas: by land, air, and sea
Enter the same three values: by land, air, and sea
A$ = by land
B$ = air
C$ = and sea
D$ = by land, air, and sea
```

With both **INPUT** and **LINE INPUT**, input is terminated when the user presses the ENTER key, which also advances the cursor to the next line. As the next example shows, a semicolon between the **INPUT** keyword and the prompt string keeps the cursor on the same line:

```
INPUT "First value: ", A
INPUT; "Second value: ", B
INPUT "     Third value: ", C
```

The following shows some sample input to the preceding program and the positions of the prompts:

```
First value: 5
Second value: 4     Third value: 3
```

3.2.3 The INPUT$ Function

Both **INPUT** and **LINE INPUT** wait for the user to press the ENTER key before they store what is typed; that is, they read a line of input, then assign it to program variables. In contrast, the **INPUT$**(*number*) function doesn't wait for the enter key to be pressed; it just reads a specified number of characters. For example, the following line in a program reads three characters typed by the user, then stores the three-character string in the variable `Test$`:

```
Test$ = INPUT$(3)
```

Unlike the **INPUT** statement, the **INPUT$** function does not prompt the user for data, nor does it echo input characters on the screen. Also, since **INPUT$** is a function, it cannot stand by itself as a complete statement. **INPUT$** must appear in an expression, as in the following:

```
INPUT x                 ' INPUT is a statement.

PRINT INPUT$(1)         ' INPUT$ is a function, so it must
Y$ = INPUT$(1)          ' appear in an expression.
```

The **INPUT$** function reads input from the keyboard as an unformatted stream of characters. Unlike **INPUT** or **LINE INPUT**, **INPUT$** accepts any key pressed, including control keys like ESC or BACKSPACE. For example, pressing the ENTER key five times assigns five carriage-return characters to the `Test$` variable in the next line:

```
Test$ = INPUT$(5)
```

3.2.4 The INKEY$ Function

The **INKEY$** function completes the list of BASIC's keyboard-input functions and statements. When BASIC encounters an expression containing the **INKEY$** function, it checks to see if the user has pressed a key since one of the following:

- The last time it found an expression with **INKEY$**
- The beginning of the program, if this is the first time **INKEY$** appears

If no key has been pressed since the last time the program checked, **INKEY$** returns a null string (""). If a key has been pressed, **INKEY$** returns the character corresponding to that key.

Example

The most important difference between **INKEY$** and the other statements and functions discussed in this section is that **INKEY$** lets your program continue doing other things while it checks for input. In contrast, **LINE INPUT**,

INPUT$, and **INPUT** suspend program execution until there is input, as show in this example:

```
PRINT "Press any key to start. Press any key to end."

' Don't do anything else until the user presses a key:
Begin$ = INPUT$(1)

I = 1

' Print the numbers from one to one million.
' Check for a key press while the loop is executing:
DO
   PRINT I
   I = I + 1

' Continue looping until the value of the variable I is
' greater than one million or until a key is pressed:
LOOP UNTIL I > 1000000 OR INKEY$ <> ""
```

3.3 Controlling the Text Cursor

When you display printed text on the screen, the text cursor marks the place on the screen where output from the program—or input typed by the user—will appear next. In the next example, after the **INPUT** statement displays its 12-letter prompt, `First name:`, the cursor waits for input in row 1 at column 13:

```
' Clear the screen; start printing in row one, column one:
CLS
INPUT "First name: ", FirstName$
```

In the next example, the semicolon at the end of the second **PRINT** statement leaves the cursor in row 2 at column 27:

```
CLS
PRINT

' Twenty-six characters are in the next line:
PRINT "Press any key to continue.";
PRINT INPUT$(1)
```

Sections 3.3.1–3.3.3 show how to control the location of the text cursor, change its shape, and get information about its location.

3.3.1 Positioning the Cursor

The input and output statements and functions discussed so far do not allow much control over where output is displayed or where the cursor is located after the output is displayed. Input prompts or output always start in the far left

column of the screen and descend one row at a time from top to bottom unless a semicolon is used in the **PRINT** or **INPUT** statements to suppress the carriage-return and line-feed sequence.

The **SPC** and **TAB** statements, discussed in Section 3.1.4, "Skipping Spaces and Advancing to a Specific Column," give some control over the location of the cursor by allowing you to move it to any column within a given row.

The **LOCATE** statement extends this control one step further. The syntax for **LOCATE** is

LOCATE [[*row*][,[*column*][,[*cursor*][,[*start*][,*stop*]]]]]

Using the **LOCATE** statement allows you to position the cursor in any *row* or *column* on the screen, as shown by the output from the next example:

```
CLS
FOR Row = 9 TO 1 STEP -2
   Column = 2 * Row
   LOCATE Row, Column
   PRINT "12345678901234567890";
NEXT
```

Output

```
12345678901234567890

    12345678901234567890

        12345678901234567890

            12345678901234567890

                12345678901234567890
```

3.3.2 Changing the Cursor's Shape

The optional *cursor*, *start*, and *stop* arguments shown in the syntax for the **LOCATE** statement also allow you to change the shape of the cursor and make it visible or invisible. A value of 1 for *cursor* makes the cursor visible, while a value of 0 makes the cursor invisible. The *start* and *stop* arguments control the height of the cursor, if it is on, by specifying the top and bottom "pixel" lines, respectively, for the cursor. (Any character on the screen is composed of lines of pixels, which are dots of light on the screen.) If a cursor spans the height of one row of text, then the line of pixels at the top of the cursor has the value 0, while the line of pixels at the bottom has a value of 7 or 13, depending on the type of display adapter you have. (For monochrome the value is 13; for color it is 7.)

You can turn the cursor on and change its shape without specifying a new location for it. For example, the following statement keeps the cursor wherever it is

at the completion of the next **PRINT** or **INPUT** statement, then makes it half a character high:

```
LOCATE , , 1, 2, 5  ' Row and column arguments both optional
```

The following examples show different cursor shapes produced using different *start* and *stop* values on a color display. Each **LOCATE** statement shown in the left column is followed by the statement

```
INPUT "PROMPT:", X$
```

Statement	Input Prompt and Cursor Shape
`LOCATE , , 1, 0, 7`	PROMPT:▮
`LOCATE , , 1, 0, 3`	PROMPT:■
`LOCATE , , 1, 4, 7`	PROMPT:▁
`LOCATE , , 1, 6, 2`	PROMPT:▀

In the preceding examples, note that making the *start* argument bigger than the *stop* argument results in a two-piece cursor.

3.3.3 Getting Information about the Cursor's Location

You can think of the functions **CSRLIN** and **POS(*n*)** as the inverses of the **LOCATE** statement: whereas **LOCATE** tells the cursor where to go, **CSRLIN** and **POS(*n*)** tell your program where the cursor is. The **CSRLIN** function returns the current row and the **POS(*n*)** function returns the current column of the cursor's position.

The argument *n* for **POS(*n*)** is what is known as a "dummy" argument; that is, *n* is a placeholder that can be any numeric expression. For example, `POS(0)` and `POS(1)` both return the same value.

Example

The following example uses the **POS(*n*)** function to print 50 asterisks in rows of 13 asterisks:

```
FOR I% = 1 TO 50
   PRINT "*";                    ' Print an asterisk and keep
                                 ' the cursor on the same line.
   IF POS(1) > 13 THEN PRINT     ' If the cursor's position
                                 ' is past column 13, advance
                                 ' to the next line.
NEXT
```

Output

```
**********
**********
**********
**********
```

3.4 Working with Data Files

Data files are physical locations on your disk where information is permanently stored. The following three tasks are greatly simplified by using data files in your BASIC programs:

1. Creating, manipulating, and storing large amounts of data
2. Accessing several sets of data with one program
3. Using the same set of data in several different programs

The sections that follow introduce the concepts of records and fields and contrast different ways to access data files from BASIC. When you have completed Sections 3.4.1– 3.4.7, you should know how to do the following:

- Create new data files
- Open existing files and read their contents
- Add new information to an existing data file
- Change the contents of an existing data file

3.4.1 How Data Files Are Organized

A data file is a collection of related blocks of information, or "records." Each record in a data file is further subdivided into "fields" or regularly recurring items of information within each record. If you compare a data file with a more old-fashioned way of storing information—for example, a folder containing application forms filled out by job applicants at a particular company—then a record is analogous to one application form in that folder. To carry the comparison one step further, a field is analogous to an item of information included on every application form, such as a Social Security number.

NOTE *If you do not want to access a file using records but instead want to treat it as an unformatted sequence of bytes, then read Section 3.4.7, "Binary File I/O."*

3.4.2 Sequential and Random-Access Files

The terms "sequential file" and "random-access file" refer to two different ways to store and access data on disk from your BASIC programs. A simplified way to think of these two kinds of files is with the following analogy: a sequential file is like a cassette tape, while a random-access file is like an LP record. To find a song on a cassette tape, you have to start at the beginning and fast-forward through the tape sequentially until you find the song you are looking for—there is no way to jump right to the song you want. This is similar to the way you have to find information in a sequential file: to get to the 500th record, you first have to read records 1 through 499.

In contrast, if you want to play a certain song on an LP, all you have to do is lift the tone arm of the record player and put the needle down right on the song: you can randomly access anything on the LP without having to play all the songs before the one you want. Similarly, you can call up any record in a random-access file just by specifying its number, greatly reducing access time.

NOTE *Although there is no way to jump directly to a specific record in a sequential file, the* ***SEEK*** *statement lets you jump directly to a specific byte in the file (to "fast forward" or "rewind" the tape, to extend the preceding analogy). See Section 3.4.7, "Binary File I/O," for more information on how to do this.*

3.4.3 Opening a Data File

Before your program can read, modify, or add to a data file, it must first open the file. BASIC does this with the **OPEN** statement. The **OPEN** statement can be used to create a new file. The following list describes the various uses of the **OPEN** statement:

- Create a new data file and open it so records can be added to it

```
' No file named PRICE.DAT is in the current directory:
OPEN "Price.Dat" FOR OUTPUT AS #1
```

- Open an existing data file so new records overwrite any data already in the file

```
' A file named PRICE.DAT is already in the current
' directory; new records can be written to it, but all
' old records are lost:
OPEN "Price.Dat" FOR OUTPUT AS #1
```

- Open an existing data file so new records are added to the end of the file, preserving data already in the file

```
OPEN "Price.Dat" FOR APPEND AS #1
```

The **APPEND** mode will also create a new file if a file with the given name does not already appear in the current directory.

- Open an existing data file so old records can be read from it

```
OPEN "Price.Dat" FOR INPUT AS #1
```

See Section 3.4.5, "Using Sequential Files," for more information about the **INPUT**, **OUTPUT**, and **APPEND** modes.

- Open an existing data file (or create a new one if a file with that name doesn't exist), then read or write fixed-length records to and from the file

```
OPEN "Price.Dat" FOR RANDOM AS #1
```

See Section 3.4.6, "Using Random-Access Files," for more information about this mode.

- Open an existing data file (or create a new one if a file with that name doesn't exist), then read data from the file or add new data to the file, starting at any byte position in the file

```
OPEN "Price.Dat" FOR BINARY AS #1
```

See Section 3.4.7, "Binary File I/O," for more information about this mode.

3.4.3.1 File Numbers in BASIC

The **OPEN** statement does more than just specify a mode for data I/O for a particular file (**OUTPUT**, **INPUT**, **APPEND**, **RANDOM**, or **BINARY**); it also associates a unique file number with that file. This file number, which can be any integer from 1 to 255, is then used by subsequent file I/O statements in the program as a shorthand way to refer to the file. As long as the file is open, this number remains associated with the file. When the file is closed, the file number is freed for use with another file. (See Section 3.4.4 for information on how files are closed.) Your BASIC programs can open more than one file at a time.

The **FREEFILE** function can help you find an unused file number. This function returns the next available number that can be associated with a file in an **OPEN** statement. For example, **FREEFILE** might return the value 3 after the following **OPEN** statements:

```
OPEN "Test1.Dat" FOR RANDOM AS #1
OPEN "Test2.Dat" FOR RANDOM AS #2
FileNum = FREEFILE
OPEN "Test3.Dat" FOR RANDOM AS #FileNum
```

The **FREEFILE** function is particularly useful when you create your own library procedures that open files. With **FREEFILE**, you don't have to pass information about the number of open files to these procedures.

3.4.3.2 File Names in BASIC

File names in **OPEN** statements can be any string expression, composed of any combination of the following characters:

- The letters a–z and A–Z
- The numbers 0–9
- The following special characters:
  ( ) { } @ # $ % ^ & ! - _ ' ~

The string expression can also contain an optional drive specification, as well as a complete or partial path specification. This means your BASIC program can work with data files on another drive or in a directory other than the one where the program is running. For example, the following **OPEN** statements are all valid:

```
OPEN "..\Grades.Qtr" FOR INPUT AS #1

OPEN "a:\salaries\1987.man" FOR INPUT AS #2

FileName$ = "TempFile"
OPEN FileName$ FOR OUTPUT AS #3

BaseName$ = "Invent"
OPEN BaseName$ + ".DAT" FOR OUTPUT AS #4
```

DOS also imposes its own restrictions on file names: you can use no more than eight characters for the base name (everything to the left of an optional period) and no more than three characters for the extension (everything to the right of an optional period). Long file names in BASIC programs are truncated in the following fashion:

File Name in Program	Resulting File Name in DOS
`Prog@Data@File`	`PROG@DAT.A@F` The BASIC name is more than 11 characters long, so BASIC takes the first 8 letters for the base name, inserts a period (.), and uses the next 3 letters as the extension. Everything else is discarded.

`Mail#.Version1`	`MAIL#.VER` The base name (`Mail#`) is shorter than eight characters, but the extension (`Version1`) is longer than three, so the extension is shortened to three characters.
`RELEASE_NOTES.BAK`	Gives the run-time error message `Bad file name`. The base name must be shorter than eight characters if you are going to include an explicit extension (`.BAK` in this case).

DOS is not case sensitive, so lowercase letters in file names are converted to all uppercase (capital) letters. Therefore, you should not rely on the mixing of lowercase and uppercase to distinguish between files. For example, if you already had a file on the disk named INVESTOR.DAT, the following **OPEN** statement would overwrite that file, destroying any information already stored in it:

```
OPEN "Investor.Dat" FOR OUTPUT AS #1
```

3.4.4 Closing a Data File

Closing a data file has two important results: first, it writes any data currently in the file's buffer (a temporary holding area in memory) to the file; second, it frees the file number associated with that file for use by another **OPEN** statement.

Use the **CLOSE** statement within a program to close a file. For example, if the file `Price.Dat` is opened with the statement

```
OPEN "Price.Dat" FOR OUTPUT AS #1
```

then the statement `CLOSE #1` ends output to `Price.Dat`. If `Price.Dat` is opened with

```
OPEN "Price.Dat" FOR OUTPUT AS #2
```

then the appropriate statement for ending output is `CLOSE #2`. A **CLOSE** statement with no file-number arguments closes all open files.

A data file is also closed when either of the following occurs:

- The BASIC program performing I/O ends (program termination always closes all open data files).
- The program performing I/O transfers control to another program with the **RUN** statement.

3.4.5 Using Sequential Files

This section discusses how records are organized in sequential data files and then shows how to read data from, or write data to, an open sequential file.

3.4.5.1 Records in Sequential Files

Sequential files are ASCII (text) files. This means you can use any word processor to view or modify a sequential file. Records are stored in sequential files as a single line of text, terminated by a carriage-return and line-feed (CR-LF) sequence. Each record is divided into fields, or repeated chunks of data that occur in the same order in every record. Figure 3.2 shows how three records might appear in a sequential file.

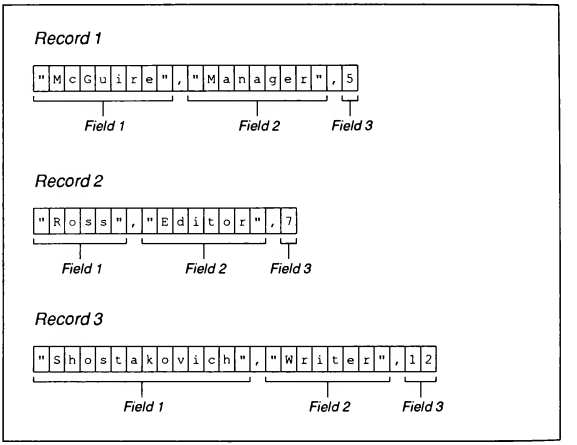


Figure 3.2 Records in Sequential Files

Note that each record in a sequential file can be a different length; moreover, fields can be different lengths in different records.

The kind of variable in which a field is stored determines where that field begins and ends. (See Sections 3.4.5.2– 3.4.5.6 for examples of reading and storing fields from records.) For example, if your program reads a field into a string variable, then any of the following can signal the end of that field:

- Double quotes (") if the string begins with double quotes
- Comma (,) if the string does not begin with double quotes
- CR-LF if the field is at the end of the record

On the other hand, if your program reads a field into a numeric variable, then any of the following can signal the end of that field:

- Comma
- One or more spaces
- CR-LF

3.4.5.2 Putting Data in a New Sequential File

You can add data to a new sequential file after first opening it to receive records with an **OPEN***filename* **FOR OUTPUT** statement. Use the **WRITE #** statement to write records to the file.

You can open sequential files for reading or for writing but not for both at the same time. If you are writing to a sequential file and want to read back the data you stored, you must first close the file, then reopen it for input.

Example

The following short program creates a sequential file named `Price.Dat`, then adds data entered at the keyboard to the file. The **OPEN** statement in this program both creates the file and readies it to receive records. The **WRITE #** then writes each record to the file. Note that the number used in the **WRITE #** statement is the same number given to the file name `Price.Dat` in the **OPEN** statement.

```
' Create a file named Price.Dat
' and open it to receive new data:

OPEN "Price.Dat" FOR OUTPUT AS #1

DO
   ' Continue putting new records in Price.Dat until the
   ' user presses ENTER without entering a company name:
   INPUT "Company (press <ENTER> to quit): ", Company$

   IF Company$ <> "" THEN

      ' Enter the other fields of the record:
      INPUT "Style: ", Style$
      INPUT "Size: ", Size$
      INPUT "Color: ", Clr$
      INPUT "Quantity: ", Qty
```

```
      ' Put the new record in the file
      ' with the WRITE # statement:
      WRITE #1, Company$, Style$, Size$, Clr$, Qty
   END IF
LOOP UNTIL Company$ = ""

' Close Price.Dat (this ends output to the file):
CLOSE #1
END
```

WARNING *If, in the preceding example, you already had a file named* `Price.Dat` *on the disk, the* ***OUTPUT*** *mode given in the* ***OPEN*** *statement would erase the existing contents of* `Price.Dat` *before writing any new data to it. If you want to add new data to the end of an existing file without erasing what is already in it, use the* ***APPEND*** *mode of* ***OPEN****. See Section 3.4.5.4, "Adding Data to a Sequential File," for more information on this mode.*

3.4.5.3 Reading Data from a Sequential File

You can read data from a sequential file after first opening it with the statement **OPEN** *filename* **FOR INPUT**. Use the **INPUT #** statement to read records from the file one field at a time. (See Section 3.4.5.6, "Other Ways to Read Data from a Sequential File," for information on other file-input statements and functions you can use with a sequential file.)

Example

The following program opens the `Price.Dat` data file created in the previous example and reads the records from the file, displaying the complete record on the screen if the quantity for the item is less than the input amount.

The `INPUT #1` statement reads one record at a time from `Price.Dat`, assigning the fields in the record to the variables `Company$`, `Style$`, `Size$`, `Clr$`, and `Qty`. Since this is a sequential file, the records are read in order from the first one entered to the last one entered.

The **EOF** (end-of-file) function tests whether the last record has been read by **INPUT #**. If the last record has been read, **EOF** returns the value –1 (true), and the loop for getting data ends; if the last record has not been read, **EOF** returns the value 0 (false), and the next record is read from the file.

```
OPEN "Price.Dat" FOR INPUT AS #1

INPUT "Display all items below what level"; Reorder

DO UNTIL EOF(1)
   INPUT #1, Company$, Style$, Size$, Clr$, Qty
   IF Qty < Reorder THEN
      PRINT  Company$, Style$, Size$, Clr$, Qty
   END IF
LOOP
```

```
CLOSE #1
END
```

3.4.5.4 Adding Data to a Sequential File

As mentioned earlier, if you have a sequential file on disk and want to add more data to the end of it, you cannot simply open the file in output mode and start writing data. As soon as you open a sequential file in output mode, you destroy its current contents. You must use the append mode instead, as shown in the next example:

```
OPEN "Price.Dat" FOR APPEND AS #1
```

In fact, **APPEND** is always a safe alternative to **OUTPUT**, since the append mode creates a new file if one with the name specified doesn't already exist. For example, if a file named `Price.Dat` did not reside on disk, the example statement above would make a new file with that name.

3.4.5.5 Other Ways to Write Data to a Sequential File

The preceding examples all use the **WRITE #** statement to write records to a sequential file. There is, however, another statement you can use to write sequential file records: **PRINT #**.

The best way to show the difference between these two data-storage statements is to examine the contents of a file created with both. The following short fragment opens a file named `Test.Dat` then places the same record in it twice, once with **WRITE #** and once with **PRINT #**. After running this program you can examine the contents of `Test.Dat` with the DOS TYPE commands:

```
OPEN "Test.Dat" FOR OUTPUT AS #1
Nm$ = "Penn, Will"
Dept$ = "User Education"
Level = 4
Age = 25
WRITE #1, Nm$, Dept$, Level, Age
PRINT #1, Nm$, Dept$, Level, Age
CLOSE #1
```

Output

```
"Penn, Will","User Education",4,25
Penn, Will    User Education              4              25
```

The record stored with **WRITE #** has commas that explicitly separate each field of the record, as well as quotes enclosing each string expression. On the other hand, **PRINT #** has written an image of the record to the file exactly as it would appear on screen with a simple **PRINT** statement. The commas in the **PRINT #** statement are interpreted as meaning "advance to the next print zone" (a new print zone occurs every 14 spaces, starting at the beginning of a line), and quotes are not placed around the string expressions.

At this point, you may be wondering what difference these output statements make, except in the appearance of the data within the file. The answer lies in what happens when your program reads the data back from the file with an **INPUT #** statement. In the following example, the program reads the record stored with **WRITE #** and prints the values of its fields without any problem:

```
OPEN "Test.Dat" FOR INPUT AS #1

' Input the first record,
' and display the contents of each field:
INPUT #1, Nm$, Dept$, Level, Age
PRINT Nm$, Dept$, Level, Age

' Input the second record,
' and display the contents of each field:
INPUT #1, Nm$, Dept$, Level, Age
PRINT Nm$, Dept$, Level, Age

CLOSE #1
```

Output

```
Penn, Will     User Education              4              25
```

However, when the program tries to input the next record stored with **PRINT #**, it produces the error message `Input past end of file`. Without double quotes enclosing the first field, the **INPUT #** statement sees the comma between `Penn` and `Will` as a field delimiter, so it assigns only the last name `Penn` to the variable `Nm$`. **INPUT #** then reads the rest of the line into the variable `Dept$`. Since all of the record has now been read, there is nothing left to put in the variables `level` and `age`. The result is the error message `Input past end of file`.

If you are storing records that have string expressions and you want to read these records later with the **INPUT #** statement, follow one of these two rules of thumb:

1. Use the **WRITE #** statement to store the records.
2. If you want to use the **PRINT #** statement, remember it does not put commas in the record to separate fields, nor does it put quotes around strings. You have to put these field separators in the **PRINT #** statement yourself.

For example, you can avoid the problems shown in the preceding example by using **PRINT #** with quotation marks surrounding each field containing a string expression, as in the example below.

Example

```
' 34 is ASCII value for double-quote character:

Q$ = CHR$(34)

' The next four statements all write the record to the
' file with double quotes around each string field:

PRINT #1, Q$ Nm$ Q$ Q$ Dept$ Q$ Level Age
PRINT #1, Q$ Nm$ Q$;Q$ Dept$ Q$;Level;Age
PRINT #1, Q$ Nm$ Q$,Q$ Dept$ Q$,Level,Age
WRITE #1, Nm$, Dept$, Level, Age
```

Output to File

```
"Penn, Will""User Education" 4  25
"Penn, Will""User Education" 4  25
"Penn, Will"  "User Education"      4             25
"Penn, Will","User Education",4,25
```

3.4.5.6 Other Ways to Read Data from a Sequential File

In the preceding sections, **INPUT #** is used to read a record (one line of data from a file), assigning different fields in the record to the variables listed after **INPUT #**. This section explores alternative ways to read data from sequential files, both as records (**LINE INPUT #**) and as unformatted sequences of bytes (**INPUT$**).

The LINE INPUT # Statement With the **LINE INPUT #** statement, your program can read a line of text exactly as it appears in a file without interpreting commas or quotes as field delimiters. This is particularly useful in programs that work with ASCII text files.

The **LINE INPUT #** statement reads an entire line from a sequential file (up to a carriage-return and line-feed sequence) into a single string variable.

Examples

The following short program reads each line from the file `Chap1.Txt` and then echoes that line to the screen:

```
' Open Chap1.Txt for sequential input:
OPEN "Chap1.Txt" FOR INPUT AS #1

' Keep reading lines sequentially from the file until
' there are none left in the file:
```

```
DO UNTIL EOF(1)

   ' Read a line from the file and store it
   ' in the variable LineBuffer$:
   LINE INPUT #1, LineBuffer$

   ' Print the line on the screen:
   PRINT LineBuffer$
LOOP
```

The preceding program is easily modified to a file-copying utility that prints each line read from the specified input file to another file, instead of to the screen:

```
' Input names of input and output files:

INPUT "File to copy: ", FileName1$
IF FileName1$ = "" THEN END
INPUT "Name of new file: ", FileName2$
IF FileName2$ = "" THEN END

' Open first file for sequential input:
OPEN FileName1$ FOR INPUT AS #1

' Open second file for sequential output:
OPEN FileName2$ FOR OUTPUT AS #2

' Keep reading lines sequentially from first file
' until there are none left in the file:
DO UNTIL EOF(1)

   ' Read a line from first file and store it in the
   ' variable LineBuffer$:
   LINE INPUT #1, LineBuffer$

   ' Write LineBuffer$ to the second file:
   PRINT #2, LineBuffer$

LOOP
```

The INPUT$ Function Yet another way to read data from sequential files (and, in fact, from any file) is to use the **INPUT$** function. Whereas **INPUT #** and **LINE INPUT #** read a line at a time from a sequential file, **INPUT$** reads a specified number of characters from a file, as shown in the following examples:

Statement	Action
`X$ = INPUT$(100, #1)`	Reads 100 characters from file number 1 and assigns all of them to the string variable `X$`
`Test$ = INPUT$(1, #2)`	Reads one character from file number 2 and assigns it to the string variable `Test$`

The **INPUT$** function without a file number always reads input from standard input (usually the keyboard).

The **INPUT$** function does what is known as "binary input"; that is, it reads a file as an unformatted stream of characters. For example, it does not see a carriage-return and line-feed sequence as signaling the end of an input operation. Therefore, **INPUT$** is the best choice when you want your program to read every single character from a file or when you want it to read a binary, or non-ASCII, file.

Example

The following program copies the named binary file to the screen, printing only alphanumeric and punctuation characters in the ASCII range 32 to 126, as well as tabs, carriage returns, and line feeds:

```
' 9 is ASCII value for horizontal tab, 10 is ASCII value
' for line feed, and 13 is ASCII value for carriage return:
CONST LINEFEED = 10, CARRETURN = 13, TABCHAR = 9

INPUT "Print which file: ", FileName$
IF FileName$ = "" THEN END

OPEN FileName$ FOR INPUT AS #1

DO UNTIL EOF(1)
   Character$ = INPUT$(1, #1)
   CharVal = ASC(Character$)
   SELECT CASE CharVal
      CASE 32 TO 126
         PRINT Character$;
      CASE TABCHAR, CARRETURN
         PRINT Character$;
      CASE LINEFEED
         IF OldCharVal <> CARRETURN THEN PRINT Character$;
      CASE ELSE
         ' This is not one of the characters this program
         ' is interested in, so don't print anything.
    END SELECT
    OldCharVal = CharVal
LOOP
```

3.4.6 Using Random-Access Files

This section discusses how records are organized in random-access data files, then shows you how to read data from and write data to a file opened for random access.

3.4.6.1 Records in Random-Access Files

Random-access records are stored quite differently from sequential records. Each random-access record is defined with a fixed length, as is each field within the record. These fixed lengths determine where a record or field begins and ends, as there are no commas separating fields, and no carriage-return and line-feed sequences between records. Figure 3.3 shows how three records might appear in a random-access file.

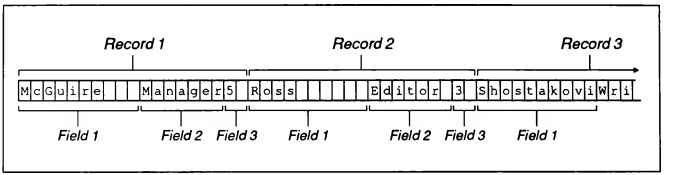


Figure 3.3 Records in a Random-Access File

If you are storing records containing numbers, using random-access files saves disk space when compared with using sequential files. This is because sequential files save numbers as a sequence of ASCII characters representing each digit, whereas random-access files save numbers in a compressed binary format.

For example, the number 17,000 is represented in a sequential file using five bytes, one for each digit. However, if 17,000 is stored in an integer field of a random-access record, it takes only two bytes of disk space.

In general, integers in random-access files take two bytes, long integers and single-precision numbers take four bytes, and double-precision numbers take eight bytes.

3.4.6.2 Adding Data to a Random-Access File

To write a program that adds data to a random-access file, follow these steps:

1. Define the fields of each record.
2. Open the file in random-access mode and specify the length of each record.
3. Get input for a new record and store the record in the file.

Each of these steps is now considerably easier than it was in BASICA, as you can see from the examples that follow.

Defining Fields You can define your own record with a **TYPE...END TYPE** statement, which allows you to create a composite data type that mixes string and numeric elements. This is a big advantage over the earlier method of setting up records with a **FIELD** statement, which required that each field be defined as a string. By defining a record with **TYPE...END TYPE**, you eliminate the need to use the functions that convert numeric data to strings (**MK***type***$**, **MKSMBF$**, and **MKDMBF$**) and strings to numeric data (**CV***type*, **CVSMBF**, and **CVDMBF**).

The following fragments contrast these two methods of defining records:

- Record defined with **TYPE...END TYPE**

```
' Define the RecordType structure:
TYPE RecordType
   Name AS STRING * 30
   Age AS INTEGER
   Salary AS SINGLE
END TYPE

' Declare the variable RecordVar
' as having the type RecordType:
DIM RecordVar AS RecordType

.
.
.
```

- Record defined with **FIELD**

```
' Define the lengths of the fields
' in the temporary storage buffer:
FIELD #1,30 AS Name$,2 AS Age$,4 AS Salary$
.
.
.
```

Opening the File and Specifying Record Length Since the length of a random-access record is fixed, you should let your program know how long you want each record to be; otherwise, record length defaults to 128 bytes.

To specify record length, use the **LEN** = clause in the **OPEN** statement. The next two fragments, which continue the contrasting examples started above, show how to use **LEN** = .

- Specifying the record length for a record that is defined with the statement **TYPE...END TYPE**

```
.
.
.
' Open the random-access file and specify the length
' of one record as being equal to the length of the
' RecordVar variable:
OPEN "EMPLOYEE.DAT" FOR RANDOM AS #1 LEN = LEN(RecordVar)
.
.
.
```

- Specifying record length for a record defined with **FIELD**

```
.
.
.
' Open the random-access file and specify the length
' of a record:
OPEN "EMPLOYEE.DAT" FOR RANDOM AS #1 LEN = 31
.
.
.
```

As you can see, when you use **FIELD**, you have to add the lengths of each field yourself (`FirstName$` is 10 bytes, `LastName$` is 15 bytes, `Age$` is 2 bytes, `Salary$` is 4 bytes, so the record is 10+15+2+4 or 31 bytes long). With **TYPE...END TYPE**, you no longer have to do these calculations. Instead, just use the **LEN** function to calculate the length of the variable you have created to hold your records (`RecordVar`, in this case).

Inputting Data and Storing the Record You can input data directly into the elements of a user-defined record without having to worry about left- or right-justification of input within a field with **LSET** or **RSET**. Compare the following two fragments, which continue the examples started above, to see the amount of code this approach saves you.

- Entering data for a random-access record and storing the record using **TYPE...END TYPE**.

```
' Enter the data:
INPUT "First name"; RecordVar.FirstName
INPUT "Last name"; RecordVar.LastName
INPUT "Age"; RecordVar.Age
INPUT "Salary"; RecordVar.Salary
```

```
' Store the record:
PUT #1, , RecordVar

.
.
.
```

- Entering data for a random-access record and storing the record using FIELD

```
' Enter the data:
INPUT "First name"; FirstNm$
INPUT "Last name"; LastNm$
INPUT "Age"; AgeVal%
INPUT "Salary"; SalaryVal!

' Left justify the data in the storage-buffer fields,
' using the MKI$ and MKS$ functions to convert numbers
' to file strings:
LSET FirstName$ = FirstNm$
LSET LastName$ = LastNm$
LSET Age$ = MKI$(AgeVal%)
LSET Salary$ = MKS$(SalaryVal!)

' Store the record:
PUT #1
```

Putting It All Together The following program puts together all the steps outlined above—defining fields, specifying record length, inputting data, and storing the input data—to open a random-access data file named `STOCK.DAT` and add records to it:

```
DEFINT A-Z

' Define structure of a single record in the random-access
' file. Each record will consist of four fixed-length fields
' ("PartNumber", "Description", "UnitPrice", "Quantity"):
TYPE StockItem
   PartNumber AS STRING * 6
   Description AS STRING * 20
   UnitPrice AS SINGLE
   Quantity AS INTEGER
END TYPE
```

```
' Declare a variable (StockRecord) using the above type:
DIM StockRecord AS StockItem

' Open the random-access file, specifying the length of one
' record as the length of the StockRecord variable:
OPEN "STOCK.DAT" FOR RANDOM AS #1 LEN=LEN(StockRecord)

' Use LOF() to calculate the number of records already in
' the file, so new records will be added after them:
RecordNumber = LOF(1) \ LEN(StockRecord)

' Now, add new records:
DO

   ' Input data for a stock record from keyboard and store
   ' in the different elements of the StockRecord variable:
   INPUT "Part Number? ", StockRecord.PartNumber
   INPUT "Description? ", StockRecord.Description
   INPUT "Unit Price ? ", StockRecord.UnitPrice
   INPUT "Quantity   ? ", StockRecord.Quantity

   RecordNumber = RecordNumber + 1

   ' Write data in StockRecord to a new record in the file:
   PUT #1, RecordNumber, StockRecord

   ' Check to see if more data are to be read:
   INPUT "More (Y/N)? ", Resp$
LOOP UNTIL UCASE$(Resp$) = "N"

' All done; close the file and end:
CLOSE  #1
END
```

If the `STOCK.DAT` file already existed, this program would add more records to the file without overwriting any that were already in the file. The following key statement makes this work:

```
RecordNumber = LOF(1) \ LEN(StockRecord)
```

Here is what happens:

1. The `LOF(1)` function calculates the total number of bytes in the file `STOCK.DAT`. If `STOCK.DAT` is new or has no records in it, `LOF(1)` returns the value 0.
2. The `LEN(StockRecord)` function calculates the number of bytes in one record. (`StockRecord` is defined as having the same structure as the user-defined type `StockItem`.)
3. Therefore, the number of records is equal to the total bytes in the file divided by the bytes in one record.

This is another advantage of having a fixed-length record. Since each record is the same size, you can always use the above formula to calculate the number of records in the file. Obviously, this would not work with a sequential file, since sequential records can have different lengths.

3.4.6.3 Reading Data Sequentially

Using the technique outlined in the preceding section for calculating the number of records in a random-access file, you can write a program that reads all the records in that file.

Example

The following program reads records sequentially (from the first record stored to the last) from the STOCK.DAT file created in the previous section:

```
' Define a record structure (type) for random-access
' file records:
TYPE StockItem
   PartNumber AS STRING * 6
   Description AS STRING * 20
   UnitPrice AS SINGLE
   Quantity AS INTEGER
END TYPE

' Declare a variable (StockRecord) using the above type:
DIM StockRecord AS StockItem

' Open the random-access file:
OPEN "STOCK.DAT" FOR RANDOM AS #1 LEN = LEN(StockRecord)

' Calculate number of records in the file:
NumberOfRecords = LOF(1) \ LEN(StockRecord)

' Read the records and write the data to the screen:
FOR RecordNumber = 1 TO NumberOfRecords

   ' Read the data from a new record in the file:
   GET #1, RecordNumber, StockRecord

   ' Print the data to the screen:
   PRINT "Part Number: ", StockRecord.PartNumber
   PRINT "Description: ", StockRecord.Description
   PRINT "Unit Price : ", StockRecord.UnitPrice
   PRINT "Quantity   : ", StockRecord.Quantity

NEXT

' All done; close the file and end:
CLOSE #1
END
```

It is not necessary to close STOCK.DAT before reading from it. Opening a file for random access lets you write to or read from the file with a single **OPEN** statement.

3.4.6.4 Using Record Numbers to Retrieve Records

You can read any record from a random-access file by specifying the record's number in a **GET** statement. You can write to any record in a random-access file by specifying the record's number in a **PUT** statement. This is one of the major advantages that random-access files have over sequential files, since sequential files do not permit direct access to a specific record.

The sample-application program INDEX.BAS, listed in Section 3.6.2, shows a technique that quickly finds a particular record by searching an index of record numbers.

Example

The following fragment shows how to use **GET** with a record number:

```
DEFINT A-Z                ' Default variable type is integer.
CONST FALSE = 0, TRUE = NOT FALSE

TYPE StockItem
   PartNumber AS STRING * 6
   Description AS STRING * 20
   UnitPrice AS SINGLE
   Quantity AS INTEGER
END TYPE

DIM StockRecord AS StockItem

OPEN "STOCK.DAT" FOR RANDOM AS #1 LEN=LEN(StockRecord)

NumberOfRecords = LOF(1) \ LEN(StockRecord)
GetMoreRecords = TRUE

DO
   PRINT "Enter record number for part you want to see ";
   PRINT "(0 to end): ";
   INPUT "", RecordNumber

   IF RecordNumber>0 AND RecordNumber<NumberOfRecords THEN

      ' Get the record whose number was entered and store
      ' it in StockRecord:
      GET #1, RecordNumber, StockRecord

      ' Display the record:
      .
      .
      .
```

```
    ELSEIF RecordNumber = 0 THEN
      GetMoreRecords = FALSE
   ELSE
      PRINT "Input value out of range."
   END IF
LOOP WHILE GetMoreRecords
END
```

3.4.7 Binary File I/O

Binary access is a third way—in addition to random access and sequential access—to read or write a file's data. Use the following statement to open a file for binary I/O:

OPEN*file* **FOR BINARY AS #***filenum*

Binary access is a way to get at the raw bytes of any file, not just an ASCII file. This makes it very useful for reading or modifying files saved in a non-ASCII format, such as Microsoft Word files or executable (.EXE) files.

Files opened for binary access are treated as an unformatted sequence of bytes. Although you can read or write a record (a variable declared as having a user-defined type) to a file opened in the binary mode, the file itself does not have to be organized into fixed-length records. In fact, binary I/O does not have to deal with records at all, unless you consider each byte in a file as a separate record.

3.4.7.1 Comparing Binary Access and Random Access

The binary-access mode is similar to the random-access mode in that you can both read from and write to a file after a single **OPEN** statement. (Binary thus differs from sequential access, where you must first close a file and then reopen it if you want to switch between reading and writing.) Also, like random access, binary access lets you move backwards and forwards within an open file. Binary access even supports the same statements used for reading and writing random-access files:

{**GET** | **PUT**} [[**#**]]*filenumber*, [[*position*]], *variable*

Here, *variable* can have any type, including a variable-length string or a user-defined type, and *position* points to the place in the file where the next **GET** or **PUT** operation will take place. (The *position* is relative to the beginning of the file; that is, the first byte in the file has *position* one, the second byte has *position* two, and so on.) If you leave off the *position* argument, successive **GET** and **PUT** operations move the file pointer sequentially through the file from the first byte to the last.

The **GET** statement reads a number of bytes from the file equal to the length of *variable*. Similarly, the **PUT** statement writes a number of bytes to the file equal to the length of *variable*. For example, if *variable* has integer type, then **GET** reads two bytes into *variable*; if *variable* has single-precision type, **GET** reads

four bytes. Therefore, if you don't specify a *position* argument in a **GET** or **PUT** statement, the file pointer is moved a distance equal to the length of *variable*.

The **GET** statement and **INPUT$** function are the only ways to read data from a file opened in binary mode. The **PUT** statement is the only way to write data to a file opened in binary mode.

Binary access, unlike random access, enables you to move to any byte position in a file and then read or write any number of bytes you want. In contrast, random access can only move to a record boundary and read a fixed number of bytes (the length of a record) each time.

3.4.7.2 Positioning the File Pointer with SEEK

If you want to move the file pointer to a certain place in a file without actually performing any I/O, use the **SEEK** statement. Its syntax is

SEEK *filenumber, position*

After a **SEEK** statement, the next read or write operation in the file opened with *filenumber* begins at the byte noted in *position*.

The counterpart to the **SEEK** statement is the **SEEK** function, with this syntax:

SEEK(*filenumber*)

The **SEEK** function tells you the byte position where the very next read or write operation begins. (If you are using binary I/O to access a file, the **LOC** and **SEEK** functions give similar results, but **LOC** returns the position of the last byte read or written, while **SEEK** returns the position of the next byte to be read or written.)

The **SEEK** statement and function also work on files opened for sequential or random access. With sequential access, both the statement and the function behave the same as they do with binary access; that is, the **SEEK** statement moves the file pointer to specific byte positions, and the **SEEK** function returns information about the next byte to read or write.

However, if a file is opened for random access, the **SEEK** statement can move the file pointer only to the beginning of a record, not to a byte within a record. Also, when used with random-access files, the **SEEK** function returns the number of the next record rather than the position of the next byte.

Example

The following program opens a QuickBASIC Quick library, then reads and prints the names of BASIC procedures and other external symbols contained in the library. This program is in the file named QLBDUMP.BAS on the QuickBASIC distribution disks.

```
' This program prints the names of Quick library procedures.
DECLARE SUB DumpSym (SymStart AS INTEGER, QHdrPos AS LONG)

TYPE ExeHdr                     ' Part of DOS .EXE header
    other1    AS STRING * 8     ' Other header information
    CParHdr   AS INTEGER        ' Size of header in paragraphs
    other2    AS STRING * 10    ' Other header information
    IP        AS INTEGER        ' Initial IP value
    CS        AS INTEGER        ' Initial (relative) CS value
END TYPE

TYPE QBHdr                      ' QLB header
    QBHead    AS STRING * 6     ' QB specific heading
    Magic     AS INTEGER        ' Magic word: identifies file as a Quick library
    SymStart  AS INTEGER        ' Offset from header to first code symbol
    DatStart  AS INTEGER        ' Offset from header to first data symbol
END TYPE

TYPE QbSym                      ' QuickLib symbol entry
    Flags     AS INTEGER        ' Symbol flags
    NameStart AS INTEGER        ' Offset into name table
    other     AS STRING * 4     ' Other header information
END TYPE

DIM EHdr AS ExeHdr, Qhdr AS QBHdr, QHdrPos AS LONG

INPUT "Enter Quick library file name: ", FileName$
FileName$ = UCASE$(FileName$)
IF INSTR(FileName$,".QLB") = 0 THEN FileName$ = FileName$ + ".QLB"
INPUT "Enter output file name or press ENTER for screen: ", OutFile$
OutFile$ = UCASE$(OutFile$)
IF OutFile$ = "" THEN OutFile$ = "CON"
OPEN FileName$ FOR BINARY AS #1
OPEN OutFile$ FOR OUTPUT AS #2

GET #1, , EHdr                  ' Read the EXE format header.
QHdrPos = (EHdr.CParHdr + EHdr.CS) * 16 + EHdr.IP + 1

GET #1, QHdrPos, Qhdr           ' Read the QuickLib format header.
IF Qhdr.Magic <> &H6C75 THEN PRINT "Not a QB UserLibrary": END

PRINT #2, "Code Symbols:": PRINT #2,
DumpSym Qhdr.SymStart, QHdrPos  ' dump code symbols
PRINT #2,
PRINT #2, "Data Symbols:": PRINT #2, ""
DumpSym Qhdr.DatStart, QHdrPos  ' dump data symbols
PRINT #2,

END

SUB DumpSym (SymStart AS INTEGER, QHdrPos AS LONG)
   DIM QlbSym AS QbSym
   DIM NextSym AS LONG, CurrentSym AS LONG

   ' Calculate the location of the first symbol entry,
   ' then read that entry:
   NextSym = QHdrPos + Qhdr.SymStart
   GET #1, NextSym, QlbSym
```

```
DO
      NextSym = SEEK(1)          ' Save the location of the next symbol.
      CurrentSym = QHdrPos + QlbSym.NameStart
      SEEK #1, CurrentSym        ' Use SEEK to move to the name
                                 ' for the current symbol entry.
      Prospect$ = INPUT$(40, 1) ' Read the longest legal string,
                                 ' plus one additional byte for
                                 ' the final null character (CHR$(0)).

      ' Extract the null-terminated name:
      SName$ = LEFT$(Prospect$, INSTR(Prospect$, CHR$(0)))

      ' Print only those names that do not begin with "__", "$", or "b$"
      ' as these names are usually considered reserved:
      T$ = LEFT$(SName$, 2)
      IF T$ <> "__" AND LEFT$(SName$, 1) <> "$" AND UCASE$(T$) <> "B$" THEN
         PRINT #2, "  " + SName$
      END IF

      GET #1, NextSym, QlbSym    ' Read a symbol entry.
   LOOP WHILE QlbSym.Flags       ' Flags=0 (false) means end of table.

END SUB
```

3.5 Working with Devices

Microsoft BASIC supports device I/O. This means certain computer peripherals can be opened for I/O just like data files on disk. Input from or output to these devices can be done with the statements and functions listed in Table 10.4, "Statements and Functions Used for Data-File I/O," with the exceptions noted in Section 3.5.1, "Differences between Device I/O and File I/O." Table 3.1 lists the devices supported by BASIC.

Table 3.1 Devices Supported by BASIC for I/O

Name	Device	I/O Mode Supported
COM1:	First serial port	Input and output
COM2:	Second serial port	Input and output
CONS:	Screen	Output only
KYBD:	Keyboard	Input only
LPT1:	First printer	Output only
LPT2:	Second printer	Output only
LPT3:	Third printer	Output only
SCRN:	Screen	Output only

3.5.1 Differences between Device I/O and File I/O

Certain functions and statements used for file I/O are not allowed for device I/O, while other statements and functions behave differently. These are some of the differences:

- The CONS:, SCRN:, and LPT*n*: devices cannot be opened in the input or append modes.
- The KYBD: device cannot be opened in the output or append modes.
- The **EOF**, **LOC**, and **LOF** functions cannot be used with the CONS:, KYBD:, LPT*n*:, or SCRN: devices.
- The **EOF**, **LOC**, and **LOF** functions can be used with the **COM***n*: serial device; however, the values these functions return have a different meaning than the values they return when used with data files. (See Section 3.5.2 for an explanation of what these functions do when used with **COM***n*:.)

Example

The following program shows how the devices LPT1: or SCRN: can be opened for output using the same syntax as that for data files. This program reads all the lines from the file chosen by the user and then prints the lines on the screen or the printer according to the user's choice.

```
CLS
' Input the name of the file to look at:
INPUT "Name of file to display: ", FileNam$

' If no name is entered, end the program;
' otherwise, open the given file for reading (INPUT):
IF FileNam$ = "" THEN END ELSE OPEN FileNam$ FOR INPUT AS #1

' Input choice for displaying file (Screen or Printer);
' continue prompting for input until either the "S" or "P"
' keys are pressed:
DO
   ' Go to row 2, column 1 on the screen and print prompt:
   LOCATE 2, 1, 1
   PRINT "Send output to screen (S), or to printer (P): ";

   ' Print over anything after the prompt:
   PRINT SPACE$(2);

   ' Relocate cursor after the prompt, and make it visible:
   LOCATE 2, 47, 1
   Choice$ = UCASE$(INPUT$(1))      ' Get input.
   PRINT Choice$
LOOP WHILE Choice$ <> "S" AND Choice$ <> "P"
```

```
' Depending on the key pressed, open either the printer
' or the screen for output:
SELECT CASE Choice$
   CASE "P"
      OPEN "LPT1:" FOR OUTPUT AS #2
      PRINT "Printing file on printer."
   CASE "S"
      CLS
      OPEN "SCRN:" FOR OUTPUT AS #2
END SELECT

' Set the width of the chosen output device to 80 columns:
WIDTH #2, 80

' As long as there are lines in the file, read a line
' from the file and print it on the chosen device:
DO UNTIL EOF(1)
   LINE INPUT #1, LineBuffer$
   PRINT #2, LineBuffer$
LOOP

CLOSE       ' End input from the file and output to the device.
END
```

3.5.2 Communications through the Serial Port

The **OPEN "COM*n*:"** statement (where *n* can be 1 or, if you have two serial ports, 2) allows you to open your computer's serial port(s) for serial (bit-by-bit) communication with other computers or with peripheral devices such as modems or serial printers. The following are some of the parameters you can specify:

- Rate of data transmission, measured in "baud" (bits per second)
- Whether or not to detect transmission errors and how those errors will be detected
- How many stop bits (1, 1.5, or 2) are to be used to signal the end of a transmitted byte
- How many bits in each byte of data transmitted or received constitute actual data

When the serial port is opened for communication, an input buffer is set aside to hold the bytes being read from other device. This is because, at high baud rates, characters arrive faster than they can be processed. The default size for this buffer is 512 bytes, and it can be modified with the **LEN** = *numbytes* option of the **OPEN "COM*n*:"** statement. The values returned by the **EOF**, **LOC**, and **LOF** functions when used with a communications device return information about the size of this buffer, as shown in the following list:

Function	Information Returned
EOF	Whether or not any characters are waiting to be read from the input buffer
LOC	The number of characters waiting in the input buffer
LOF	The amount of space remaining (in bytes) in the output buffer

Since every character is potentially significant data, both **INPUT #** and **LINE INPUT #** have serious drawbacks for getting input from another device. This is because **INPUT #** stops reading data into a variable when it encounters a comma or new line (and, sometimes, a space or double quote), and **LINE INPUT #** stops reading data when it encounters a new line. This makes **INPUT$** the best function to use for input from a communications device, since it reads all characters.

The following line uses the **LOC** function to check the input buffer for the number of characters waiting there from the communications device opened as file #1; it then uses the **INPUT$** function to read those characters, assigning them to a string variable named `ModemInput$`:

```
ModemInput$ = INPUT$(LOC(1), #1)
```

3.6 Sample Applications

The sample applications listed in this section include a screen-handling program that prints a calendar for any month in any year from 1899 to 2099, a file I/O program that builds and searches an index of record numbers from a random-access file, and a communications program that makes your PC behave like a terminal when connected to a modem.

3.6.1 Perpetual Calendar (CAL.BAS)

After prompting the user to input a month from 1 to 12 and a year from 1899 to 2099, the following program prints the calendar for the given month and year. The `IsLeapYear` procedure makes appropriate adjustments to the calendar for months in a leap year.

Statements and Functions Used

This program demonstrates the following screen-handling functions and statements discussed in Sections 3.1– 3.3:

- **INPUT**
- **INPUT$**

- **LOCATE**
- **POS(0)**
- **PRINT**
- **PRINT USING**
- **TAB**

Program Listing

The perpetual calendar program CAL.BAS is listed below.

```
DEFINT A-Z                ' Default variable type is integer.

' Define a data type for the names of the months and the
' number of days in each:
TYPE MonthType
   Number AS INTEGER      ' Number of days in the month
   MName AS STRING * 9    ' Name of the month
END TYPE

' Declare procedures used:
DECLARE FUNCTION IsLeapYear% (N%)
DECLARE FUNCTION GetInput% (Prompt$, Row%, LowVal%, HighVal%)

DECLARE SUB PrintCalendar (Year%, Month%)
DECLARE SUB ComputeMonth (Year%, Month%, StartDay%, TotalDays%)

DIM MonthData(1 TO 12) AS MonthType

' Initialize month definitions from DATA statements below:
FOR I = 1 TO 12
   READ MonthData(I).MName, MonthData(I).Number
NEXT

' Main loop, repeat for as many months as desired:
DO
   CLS

   ' Get year and month as input:
   Year = GetInput("Year (1899 to 2099): ", 1, 1899, 2099)
   Month = GetInput("Month (1 to 12): ", 2, 1, 12)

   ' Print the calendar:
   PrintCalendar Year, Month
' Another Date?
   LOCATE 13, 1           ' Locate in 13th row, 1st column.
   PRINT "New Date? ";    ' Keep cursor on same line.
   LOCATE , , 1, 0, 13    ' Turn cursor on and make it one
                          ' character high.
   Resp$ = INPUT$(1)      ' Wait for a key press.
   PRINT Resp$            ' Print the key pressed.

LOOP WHILE UCASE$(Resp$) = "Y"
END
```

```
' Data for the months of a year:
DATA January, 31, February, 28, March, 31
DATA April, 30, May, 31, June, 30, July, 31, August, 31
DATA September, 30, October, 31, November, 30, December, 31

' ====================== COMPUTEMONTH ======================
'  Computes the first day and the total days in a month
' ==========================================================

SUB ComputeMonth (Year, Month, StartDay, TotalDays) STATIC
   SHARED MonthData() AS MonthType

   CONST LEAP   = 366 MOD 7
   CONST NORMAL = 365 MOD 7

   ' Calculate total number of days (NumDays) since 1/1/1899:

   ' Start with whole years:
   NumDays = 0
   FOR I = 1899 TO Year - 1
      IF IsLeapYear(I) THEN         ' If leap year,
         NumDays = NumDays + LEAP   ' add 366 MOD 7.
      ELSE                          ' If normal year,
         NumDays = NumDays + NORMAL ' add 365 MOD 7.
      END IF
   NEXT

   ' Next, add in days from whole months:
   FOR I = 1 TO Month - 1
      NumDays = NumDays + MonthData(I).Number
   NEXT

   ' Set the number of days in the requested month:
   TotalDays = MonthData(Month).Number

   ' Compensate if requested year is a leap year:
   IF IsLeapYear(Year) THEN

      ' If after February, add one to total days:
      IF Month > 2 THEN
         NumDays = NumDays + 1

      ' If February, add one to the month's days:
      ELSEIF Month = 2 THEN
         TotalDays = TotalDays + 1
      END IF
   END IF

   ' 1/1/1899 was a Sunday, so calculating "NumDays MOD 7"
   ' gives the day of week (Sunday = 0, Monday = 1, Tuesday
   ' = 2, and so on) for the first day of the input month:
   StartDay = NumDays MOD 7
END SUB
```

```
' ======================== GETINPUT ========================
'       Prompts for input, then tests for a valid range
' ==========================================================

FUNCTION GetInput (Prompt$, Row, LowVal, HighVal) STATIC

   ' Locate prompt at specified row, turn cursor on and
   ' make it one character high:
   LOCATE Row, 1, 1, 0, 13
   PRINT Prompt$;

   ' Save column position:
   Column = POS(0)

   ' Input value until it's within range:
   DO
      LOCATE Row, Column    ' Locate cursor at end of prompt.
      PRINT SPACE$(10)      ' Erase anything already there.
      LOCATE Row, Column    ' Relocate cursor at end of prompt.
      INPUT "", Value       ' Input value with no prompt.
   LOOP WHILE (Value < LowVal OR Value > HighVal

   ' Return valid input as value of function:
   GetInput = Value

END FUNCTION

' ======================== ISLEAPYEAR ======================
'   Determines if a year is a leap year or not
' ==========================================================

FUNCTION IsLeapYear (N) STATIC

   ' If the year is evenly divisible by 4 and not divisible
   ' by 100, or if the year is evenly divisible by 400,
   ' then it's a leap year:
   IsLeapYear = (N MOD 4=0 AND N MOD 100<>0) OR (N MOD 400=0)
END FUNCTION

' ======================== PRINTCALENDAR ===================
'   Prints a formatted calendar given the year and month
' ==========================================================

SUB PrintCalendar (Year, Month) STATIC
SHARED MonthData() AS MonthType

   ' Compute starting day (Su M Tu ...)
   ' and total days for the month:
   ComputeMonth Year, Month, StartDay, TotalDays
   CLS
   Header$ = RTRIM$(MonthData(Month).MName) + "," + STR$(Year)

   ' Calculate location for centering month and year:
   LeftMargin = (35 - LEN(Header$)) \ 2
```

```
' Print header:
   PRINT TAB(LeftMargin); Header$
   PRINT
   PRINT "Su    M    Tu    W    Th    F    Sa"
   PRINT

   ' Recalculate and print tab
   ' to the first day of the month (Su M Tu ...):
   LeftMargin = 5 * StartDay + 1
   PRINT TAB(LeftMargin);

   ' Print out the days of the month:
   FOR I = 1 TO TotalDays
      PRINT USING "##"; I;

      ' Advance to the next line
      ' when the cursor is past column 32:
      IF POS(0) > 32 THEN PRINT
   NEXT

END SUB
```

Output

```
              June, 1988

Su    M    Tu    W    Th    F    Sa

                   1     2     3     4
 5     6     7     8     9    10    11
12    13    14    15    16    17    18
19    20    21    22    23    24    25
26    27    28    29    30


New Date?
```

3.6.2 Indexing a Random-Access File (INDEX.BAS)

The following program uses an indexing technique to store and retrieve records in a random-access file. Each element of the `Index` array has two parts: a string field (`PartNumber`) and an integer field (`RecordNumber`). This array is sorted alphabetically on the `PartNumber` field, which allows the array to be rapidly searched for a specific part number using a binary search.

The `Index` array functions much like the index to a book. When you want to find the pages in a book that deal with a particular topic, you look up an entry for that topic in the index. The entry then points to a page number in the book. Similarly, this program looks up a part number in the alphabetically sorted `Index` array. Once it finds the part number, the associated record number in the `RecordNumber` field points to the record containing all the information for that part.

Statements and Functions Used

This program demonstrates the following functions and statements used in accessing random-access files:

- **TYPE...END TYPE**
- **OPEN...FOR RANDOM**
- **GET #**
- **PUT #**
- **LOF**

Program Listing

The random-access file indexing program INDEX.BAS is listed below.

```
DEFINT A-Z

' Define the symbolic constants used globally in the program:
CONST FALSE = 0, TRUE = NOT FALSE

' Define a record structure for random-file records:
TYPE StockItem
   PartNumber AS STRING * 6
   Description AS STRING * 20
   UnitPrice AS SINGLE
   Quantity AS INTEGER
END TYPE

' Define a record structure for each element of the index:
TYPE IndexType
   RecordNumber AS INTEGER
   PartNumber AS STRING * 6
END TYPE

' Declare procedures that will be called:
DECLARE FUNCTION Filter$ (Prompt$)
DECLARE FUNCTION FindRecord% (PartNumber$, RecordVar AS StockItem)

DECLARE SUB AddRecord (RecordVar AS StockItem)
DECLARE SUB InputRecord (RecordVar AS StockItem)
DECLARE SUB PrintRecord (RecordVar AS StockItem)
DECLARE SUB SortIndex ()
DECLARE SUB ShowPartNumbers ()
```

```
' Define a buffer (using the StockItem type)
' and define and dimension the index array:
DIM StockRecord AS StockItem, Index(1 TO 100) AS IndexType

' Open the random-access file:
OPEN "STOCK.DAT" FOR RANDOM AS #1 LEN = LEN(StockRecord)

' Calculate number of records in the file:
NumberOfRecords = LOF(1) \ LEN(StockRecord)

' If there are records, read them and build the index:
IF NumberOfRecords <> 0 THEN
   FOR RecordNumber = 1 TO NumberOfRecords

      ' Read the data from a new record in the file:
      GET #1, RecordNumber, StockRecord

      ' Place part number and record number in index:
      Index(RecordNumber).RecordNumber = RecordNumber
      Index(RecordNumber).PartNumber = StockRecord.PartNumber
   NEXT

   SortIndex            ' Sort index in part-number order.
END IF

DO                      ' Main-menu loop.
   CLS
   PRINT "(A)dd records."
   PRINT "(L)ook up records."
   PRINT "(Q)uit program."
   PRINT
   LOCATE , , 1
   PRINT "Type your choice (A, L, or Q) here: ";

   ' Loop until user presses, A, L, or Q:
   DO
      Choice$ = UCASE$(INPUT$(1))
   LOOP WHILE INSTR("ALQ", Choice$) = 0

   ' Branch according to choice:
   SELECT CASE Choice$
      CASE "A"
         AddRecord StockRecord
      CASE "L"
         IF NumberOfRecords = 0 THEN
            PRINT : PRINT "No records in file yet. ";
            PRINT "Press any key to continue.";
            Pause$ = INPUT$(1)
         ELSE
            InputRecord StockRecord
         END IF
      CASE "Q"          ' End program.
   END SELECT
LOOP UNTIL Choice$ = "Q"

CLOSE #1                ' All done, close file and end.
END
```

```
' ======================== ADDRECORD ======================
' Adds records to the file from input typed at the keyboard
' =========================================================

SUB AddRecord (RecordVar AS StockItem) STATIC
   SHARED Index() AS IndexType, NumberOfRecords
   DO
      CLS
      INPUT "Part Number: ", RecordVar.PartNumber
      INPUT "Description: ", RecordVar.Description

      ' Call the Filter$ FUNCTION to input price & quantity:
      RecordVar.UnitPrice = VAL(Filter$("Unit Price : "))
      RecordVar.Quantity = VAL(Filter$("Quantity   : "))

      NumberOfRecords = NumberOfRecords + 1

      PUT #1, NumberOfRecords, RecordVar

      Index(NumberOfRecords).RecordNumber = NumberOfRecords
      Index(NumberOfRecords).PartNumber = RecordVar.PartNumber
      PRINT : PRINT "Add another? ";
      OK$ = UCASE$(INPUT$(1))
   LOOP WHILE OK$ = "Y"

   SortIndex            ' Sort index file again.
END SUB

' ========================= FILTER ========================
'       Filters all non-numeric characters from a string
'       and returns the filtered string
' =========================================================

FUNCTION Filter$ (Prompt$) STATIC
   ValTemp2$ = ""
   PRINT Prompt$;                      ' Print the prompt passed.
   INPUT "", ValTemp1$                 ' Input a number as
                                       ' a string.
   StringLength = LEN(ValTemp1$)       ' Get the string's length.
   FOR I% = 1 TO StringLength          ' Go through the string,
      Char$ = MID$(ValTemp1$, I%, 1) ' one character at a time.

      ' Is the character a valid part of a number (i.e.,
      ' a digit or a decimal point)?  If yes, add it to
      ' the end of a new string:
      IF INSTR(".0123456789", Char$) > 0 THEN
         ValTemp2$ = ValTemp2$ + Char$

      ' Otherwise, check to see if it's a lowercase "l",
      ' since typewriter users may enter a one that way:
      ELSEIF Char$ = "l" THEN
         ValTemp2$ = ValTemp2$ + "1" ' Change the "l" to a "1".
      END IF
   NEXT I%

   Filter$ = ValTemp2$                 ' Return filtered string.

END FUNCTION
```

```
' ======================== FINDRECORD ====================
'  Uses a binary search to locate a record in the index
' ========================================================

FUNCTION FindRecord% (Part$, RecordVar AS StockItem) STATIC
   SHARED Index() AS IndexType, NumberOfRecords

   ' Set top and bottom bounds of search:
   TopRecord = NumberOfRecords
   BottomRecord = 1

   ' Search until top of range is less than bottom:
   DO UNTIL (TopRecord < BottomRecord)

      ' Choose midpoint:
      Midpoint = (TopRecord + BottomRecord) \ 2

      ' Test to see if it's the one wanted (RTRIM$()
      ' trims trailing blanks from a fixed string):
      Test$ = RTRIM$(Index(Midpoint).PartNumber)

      ' If it is, exit loop:
      IF Test$ = Part$ THEN
         EXIT DO

      ' Otherwise, if what we're looking for is greater,
      ' move bottom up:
      ELSEIF Part$ > Test$ THEN
         BottomRecord = Midpoint + 1

      ' Otherwise, move the top down:
      ELSE
         TopRecord = Midpoint - 1
      END IF
   LOOP

   ' If part was found, input record from file using
   ' pointer in index and set FindRecord% to TRUE:
   IF Test$ = Part$ THEN
      GET #1, Index(Midpoint).RecordNumber, RecordVar
      FindRecord% = TRUE

   ' Otherwise, if part was not found, set FindRecord%
   ' to FALSE:
   ELSE
      FindRecord% = FALSE
   END IF
END FUNCTION

' ======================== INPUTRECORD ====================
'    First, INPUTRECORD calls SHOWPARTNUMBERS, which prints
'    a menu of part numbers on the top of the screen. Next,
'    INPUTRECORD prompts the user to enter a part number.
'    Finally, it calls the FINDRECORD and PRINTRECORD
'    procedures to find and print the given record.
' ========================================================
```

```
SUB InputRecord (RecordVar AS StockItem) STATIC
   CLS
   ShowPartNumbers      ' Call the ShowPartNumbers SUB.

   ' Print data from specified records
   ' on the bottom part of the screen:
   DO
      PRINT "Type a part number listed above ";
      INPUT "(or Q to quit) and press <ENTER>: ", Part$
      IF UCASE$(Part$) <> "Q" THEN
         IF FindRecord(Part$, RecordVar) THEN
            PrintRecord RecordVar
         ELSE
            PRINT "Part not found."
         END IF
      END IF
      PRINT STRING$(40, "_")
   LOOP WHILE UCASE$(Part$) <> "Q"

   VIEW PRINT   ' Restore the text viewport to entire screen.
END SUB

' ======================= PRINTRECORD =====================
'               Prints a record on the screen
' =========================================================

SUB PrintRecord (RecordVar AS StockItem) STATIC
   PRINT "Part Number: "; RecordVar.PartNumber
   PRINT "Description: "; RecordVar.Description
   PRINT USING "Unit Price :$$###.##"; RecordVar.UnitPrice
   PRINT "Quantity   :"; RecordVar.Quantity
END SUB

' ===================== SHOWPARTNUMBERS ===================
' Prints an index of all the part numbers in the upper part
' of the screen
' =========================================================

SUB ShowPartNumbers STATIC
   SHARED Index() AS IndexType, NumberOfRecords

   CONST NUMCOLS = 8, COLWIDTH = 80 \ NUMCOLS

   ' At the top of the screen, print a menu indexing all
   ' the part numbers for records in the file. This menu is
   ' printed in columns of equal length (except possibly the
   ' last column, which may be shorter than the others):
   ColumnLength = NumberOfRecords
   DO WHILE ColumnLength MOD NUMCOLS
      ColumnLength = ColumnLength + 1
   LOOP
   ColumnLength = ColumnLength \ NUMCOLS
   Column = 1
   RecordNumber = 1
```

```
DO UNTIL RecordNumber > NumberOfRecords
      FOR Row = 1 TO ColumnLength
         LOCATE Row, Column
         PRINT Index(RecordNumber).PartNumber
         RecordNumber = RecordNumber + 1
         IF RecordNumber > NumberOfRecords THEN EXIT FOR
      NEXT Row
      Column = Column + COLWIDTH
   LOOP

   LOCATE ColumnLength + 1, 1
   PRINT STRING$(80, "_")       ' Print separator line.

   ' Scroll information about records below the part-number
   ' menu (this way, the part numbers are not erased):
   VIEW PRINT ColumnLength + 2 TO 24
END SUB

' ========================= SORTINDEX =====================
'                Sorts the index by part number
' =========================================================

SUB SortIndex STATIC
   SHARED Index() AS IndexType, NumberOfRecords

   ' Set comparison offset to half the number of records
   ' in index:
   Offset = NumberOfRecords \ 2

   ' Loop until offset gets to zero:
   DO WHILE Offset > 0
      Limit = NumberOfRecords - Offset
      DO

         ' Assume no switches at this offset:
         Switch = FALSE

         ' Compare elements and switch ones out of order:
         FOR I = 1 TO Limit
            IF Index(I).PartNumber > Index(I + Offset).PartNumber THEN
               SWAP Index(I), Index(I + Offset)
               Switch = I
            END IF
         NEXT I

         ' Sort on next pass only to where
         ' last switch was made:
         Limit = Switch
      LOOP WHILE Switch

      ' No switches at last offset, try one half as big:
      Offset = Offset \ 2
   LOOP
END SUB
```

Output

```
0235      0417      0583      8721

Type a part number listed above (or Q to quit) and press <ENTER>: 0417
Part Number: 0417
Description: oil filter
Unit Price :   $5.99
Quantity   : 12

Type a part number listed above (or Q to quit) and press <ENTER>: 8721
Part Number: 8721
Description: chamois cloth
Unit Price :   $8.99
Quantity   : 23

Type a part number listed above (or Q to quit) and press <ENTER>:
```

3.6.3 Terminal Emulator (TERMINAL.BAS)

The following simple program turns your computer into a "dumb" terminal; that is, it makes your computer function solely as an I/O device in tandem with a modem. This program uses the **OPEN "COM1:"** statement and associated device I/O functions to do the following two things:

1. Send the characters you type to the modem
2. Print characters returned by the modem on the screen

Note that typed characters displayed on the screen are first sent to the modem and then returned to your computer through the open communications channel. You can verify this if you have a modem with Receive Detect (RD) and Send Detect (SD) lights—they will flash in sequence as you type.

To dial a number, you would have to enter the Attention Dial Touch-Tone (ATDT) or Attention Dial Pulse (ATDP) commands at the keyboard (assuming you have a Hayes®-compatible modem).

Any other commands sent to the modem would also have to be entered at the keyboard.

Statements and Functions Used

This program demonstrates the following functions and statements used in communicating with a modem through your computer's serial port:

- **OPEN "COM1:"**
- **EOF**
- **INPUT$**
- **LOC**

Program Listing

```
DEFINT A-Z

DECLARE SUB Filter (InString$)

COLOR 7, 1                          ' Set screen color.
CLS

Quit$ = CHR$(0) + CHR$(16)          ' Value returned by INKEY$
                                    ' when ALT+q is pressed.

' Set up prompt on bottom line of screen and turn cursor on:
LOCATE 24, 1, 1
PRINT STRING$(80, "_");
LOCATE 25, 1
PRINT TAB(30); "Press ALT+q to quit";

VIEW PRINT 1 TO 23                  ' Print between lines 1 & 23.

' Open communications (1200 baud, no parity, 8-bit data,
' 1 stop bit, 256-byte input buffer):
OPEN "COM1:1200,N,8,1" FOR RANDOM AS #1LEN = 256

DO                                  ' Main communications loop.

   KeyInput$ = INKEY$               ' Check the keyboard.

   IF KeyInput$ = Quit$ THEN        ' Exit the loop if the user
      EXIT DO                       ' pressed ALT+q.

   ELSEIF KeyInput$ <> "" THEN      ' Otherwise, if the user has
      PRINT #1, KeyInput$;          ' pressed a key, send the
   END IF                           ' character typed to modem.
```

```
   ' Check the modem. If characters are waiting (EOF(1) is
   ' true), get them and print them to the screen:
   IF NOT EOF(1) THEN

       ' LOC(1) gives the number of characters waiting:
       ModemInput$ = INPUT$(LOC(1), #1)

       Filter ModemInput$          ' Filter out line feeds and
       PRINT ModemInput$;          ' backspaces, then print.
    END IF
LOOP

CLOSE                              ' End communications.
CLS
END

'
' ========================= FILTER ========================
'               Filters characters in an input string
' =========================================================
'
SUB Filter (InString$) STATIC

   ' Look for backspace characters and recode
   ' them to CHR$(29) (the LEFT cursor key):
   DO
      BackSpace = INSTR(InString$, CHR$(8))
      IF BackSpace THEN
      MID$(InString$, BackSpace) = CHR$(29)
      END IF
   LOOP WHILE BackSpace

   ' Look for line-feed characters and
   ' remove any found:
   DO
      LnFd = INSTR(InString$, CHR$(10))
      IF LnFd THEN
   InString$=LEFT$(InString$,LnFd-1)+MID$(InString$,LnFd+1)
      END IF
   LOOP WHILE LnFd

END SUB
```

CHAPTER 4

String Processing

4

This chapter shows you how to manipulate sequences of ASCII characters, known as strings. String manipulation is indispensable when you are processing text files, sorting data, or modifying string-data input.

When you are finished with this chapter, you will know how to perform the following string-processing tasks:

- Declare fixed-length strings
- Compare strings and sort them alphabetically
- Search for one string inside another
- Get part of a string
- Trim spaces from the beginning or end of a string
- Generate a string by repeating a single character
- Change uppercase letters in a string to lowercase and vice versa
- Convert numeric expressions to string representations and vice versa

4.1 Strings Defined

A string is a sequence of contiguous characters. Examples of characters are the letters of the alphabet (a–z and A–Z), punctuation symbols such as commas (,) or question marks (?), and other symbols from the fields of math and finance such as plus (+) or percent (%) signs.

In this chapter, the term "string" can refer to any of the following:

- A string constant
- A string variable
- A string expression

String constants are declared in one of two ways:

1. By enclosing a sequence of characters between double quotes, as in the following **PRINT** statement:

   ```
   PRINT "Processing the file. Please wait."
   ```

   This is known as a "literal string constant."

2. By setting a name equal to a literal string in a **CONST** statement, as in the following:

   ```
   ' Define the string constant MESSAGE:
   CONST MESSAGE = "Drive door open."
   ```

   This is known as a "symbolic string constant."

String variables can be declared in one of three ways:

1. By appending the string-type suffix ($) to the variable name:

   ```
   LINE INPUT "Enter your name: "; Buffer$
   ```

2. By using the **DEFSTR** statement:

   ```
   ' All variables beginning with the letter "B"
   ' are strings, unless they end with one of the
   ' numeric-type suffixes (%, &, !, or #):
   DEFSTR B
   .
   .
   .
   Buffer = "Your name here"  ' Buffer is a string variable
   ```

3. By declaring the string name in an **AS STRING** statement:

   ```
   DIM Buffer1 AS STRING       ' Buffer1 is a variable-
                               ' length string.
   DIM Buffer2 AS STRING*10    ' Buffer2 is a fixed-length
                               ' string 10 bytes long.
   ```

A string expression is a combination of string variables, constants, and/or string functions.

Of course, the character components of strings are not stored in your computer's memory in a form generally recognizable to humans. Instead, each character is translated to a number known as the ASCII code for that character. For example, capital "A" is stored as decimal 65 (or hexadecimal 41H), while lowercase "a" is stored as decimal 97 (or hexadecimal 61H). See Appendix D, "Keyboard Scan Codes and ASCII Character Codes," for a complete list of the ASCII codes for each character.

You can also use the BASIC **ASC** function to determine the ASCII code for a character; for example, `ASC("A")` returns the value 65. The inverse of the **ASC** function is the **CHR$** function. **CHR$** takes an ASCII code as its argument and returns the character with that code; for example, the statement `PRINT CHR$(64)` displays the character `@`.

4.2 Variable- and Fixed-Length Strings

In previous versions of BASIC, strings were always variable length. BASIC now supports both variable-length strings and fixed-length strings.

4.2.1 Variable-Length Strings

Variable-length strings are "elastic"; that is, they expand and contract to store different numbers of characters (from none to a maximum of 32,767). For example, the length of the variable `Temp$` in the following example varies according to the size of what is stored in `Temp$`:

```
Temp$ = "1234567"

' LEN function returns length of string
'   (number of characters it contains):
PRINT LEN(Temp$)

Temp$ = "1234"
PRINT LEN(Temp$)
```

Output

```
 7
 4
```

4.2.2 Fixed-Length Strings

Fixed-length strings are commonly used as record elements in a **TYPE...END TYPE** user-defined data type. However, they can also be declared by themselves in **COMMON, DIM, REDIM, SHARED,** or **STATIC** statements, as in the following statement:

```
DIM Buffer AS STRING * 10
```

As their name implies, fixed-length strings have a constant length, regardless of the length of the string stored in them. This is shown by the output from the following example:

```
DIM LastName AS STRING * 12
DIM FirstName AS STRING * 10

LastName = "Huntington-Ashley"
FirstName = "Claire"

PRINT "123456789012345678901234567890"
PRINT FirstName; LastName
PRINT LEN(FirstName)
PRINT LEN(LastName)
```

Output

```
123456789012345678901234567890
Claire     Huntington-A
 10
 12
```

Note that the lengths of the string variables `FirstName` and `LastName` did not change, as demonstrated by the values returned by the **LEN** function (as well as the four spaces following the six-letter name, `Claire`).

The output from the above example also illustrates how values assigned to fixed-length variables are left-justified and, if necessary, truncated on the right. It is not necessary to use the **LSET** function to left-justify values in fixed-length strings; this is done automatically. If you want to right-justify a string inside a fixed-length string, use **RSET**, as shown here:

```
DIM NameBuffer AS STRING * 10
RSET NameBuffer = "Claire"
PRINT "1234567890"
PRINT NameBuffer
```

Output

```
1234567890
    Claire
```

4.3 Combining Strings

Two strings can be combined with the plus (+) operator. The string following the plus operator is appended to the string preceding the plus operator, as shown in the next example:

```
A$ = "first string"
B$ = "second string"
C$ = A$ + B$
PRINT C$
```

Output

```
first stringsecond string
```

The process of combining strings in this way is called "concatenation," which means linking together.

Note that in the previous example, the two strings are combined without any intervening spaces. If you want a space in the combined string, you could pad one of the strings `A$` or `B$`, like this:

```
B$ = " second string"          ' Leading blank in B$
```

Because values are left-justified in fixed-length strings, you may find extra spaces when you concatenate them, as in this example:

```
TYPE NameType
   First AS STRING * 10
   Last  AS STRING * 12
END TYPE

DIM NameRecord AS NameType

' The constant "Ed" is left-justified in the variable
' NameRecord.First, so there are eight trailing blanks:
NameRecord.First = "Ed"
NameRecord.Last  = "Feldstein"

' Print a line of numbers for reference:
PRINT "12345678901234567890123456789O"

WholeName$ = NameRecord.First + NameRecord.Last
PRINT WholeName$
```

Output

```
12345678901234567890123456789O
Ed        Feldstein
```

The **LTRIM$** function returns a string with its leading spaces stripped away, while the **RTRIM$** function returns a string with its trailing spaces stripped away. The original string is unaltered. (See Section 4.6, "Retrieving Parts of Strings," for more information on these functions.)

4.4 Comparing Strings

Strings are compared with the following relational operators:

Operator	Meaning
< >	Not equal
=	Equal
<	Less than
>	Greater than
<=	Less than or equal to
>=	Greater than or equal to

A single character is greater than another character if its ASCII value is greater. For example, the ASCII value of the letter "B" is greater than the ASCII value of the letter "A," so the expression `"B" > "A"` is true.

When comparing two strings, BASIC looks at the ASCII values of corresponding characters. The first character where the two strings differ determines the alphabetical order of the strings. For instance, the strings `"doorman"` and `"doormats"` are the same up to the seventh character in each (`"n"` and `"t"`). Since the ASCII value of `"n"` is less than the ASCII value of `"t"`, the expression `"doorman" < "doormats"` is true. Note that the ASCII values for letters follow alphabetical order from A to Z and from a to z, so you can use the relational operators listed above to alphabetize strings. Moreover, uppercase letters have smaller ASCII values than lowercase letters, so in a sorted list of strings, `"ASCII"` would come before `"ascii"`.

If there is no difference between corresponding characters of two strings and they are the same length, then the two strings are equal. If there is no difference between corresponding characters of two strings, but one of the strings is longer, then the longer string is greater than the shorter string. For example `"abc" = "abc"` and `"aaaaaaa" > "aaa"` are both true expressions.

Leading and trailing blank spaces are significant in string comparisons. For example, the string `" test"` is less than the string `"test"`, since a blank space (`" "`) is less than a `"t"`; on the other hand, the string `"test "` is greater than the string `"test"`. Keep this in mind when comparing fixed-length and variable-length strings, since fixed-length strings may contain trailing spaces.

4.5 Searching for Strings

One of the most common string-processing tasks is searching for a string inside another string. The **INSTR**(*string1*, *string2*) function tells you whether or not *string2* is contained in *string1* by returning the position of the first character in *string1* (if any) where the match begins, as shown in this next example:

```
String1$ = "A line of text with 37 letters in it."
String2$ = "letters"

PRINT "          1         2         3         4"
PRINT "1234567890123456789012345678901234567890"
PRINT String1$
PRINT String2$
PRINT INSTR(String1$, String2$)
```

Output

```
1         2         3         4
1234567890123456789012345678901234567890
A line of text with 37 letters in it.
letters
 24
```

If no match is found (that is, *string2* is not a substring of *string1*), **INSTR** returns the value 0.

A variation of the syntax shown above, **INSTR**(*position*, *string1*, *string2*), allows you to specify where you want the search to start in *string1*. To illustrate, making the following modification to the preceding example changes the output:

```
' Start looking for a match at the 30th
' character of String1$:
PRINT INSTR(30, String1$, String2$)
```

Output

```
1         2         3         4
1234567890123456789012345678901234567890
A line of text with 37 letters in it.
letters
 0
```

In other words, the string `"letters"` does not appear after the 30th character of `String1$`.

The **INSTR(** *position, string1, string2*) variation is useful for finding every occurrence of *string2* in *string1* instead of just the first occurrence, as shown in the next example:

```
String1$ = "the dog jumped over the broken saxophone."
String2$ = "the"
PRINT String1$

Start      = 1
NumMatches = 0

DO
   Match = INSTR(Start, String1$, String2$)
   IF Match > 0 THEN
      PRINT TAB(Match); String2$
      Start      = Match + 1
      NumMatches = NumMatches + 1
   END IF
LOOP WHILE MATCH

PRINT "Number of matches ="; NumMatches
```

Output

```
the dog jumped over the broken saxophone.
the
                              the
Number of matches = 2
```

4.6 Retrieving Parts of Strings

Section 4.3, "Combining Strings," shows how to put strings together (concatenate them) by using the + operator. Several functions are available in BASIC that take strings apart, returning pieces of a string, either from the left side, the right side, or the middle of a target string.

4.6.1 Retrieving Characters from the Left Side of a String

The **LEFT$** and **RTRIM$** functions return characters from the left side of a string. The **LEFT$(***string, number*) function returns the specified *number* of characters from the left side of the *string*. For example:

```
Test$ = "Overtly"

' Print the leftmost 4 characters from Test$:
PRINT LEFT$(Test$, 4)
```

Output

```
Over
```

Note that **LEFT$**, like the other functions described in this chapter, does not change the original string `Test$`; it merely returns a different string, a copy of part of `Test$`.

The **RTRIM$** function returns the left part of a string after removing any blank spaces at the end. For example, contrast the output from the two **PRINT** statements in the following example:

```
PRINT "a left-justified string      "; "next"
PRINT RTRIM$("a left-justified string      "); "next"
```

Output

```
a left-justified string      next
a left-justified stringnext
```

The **RTRIM$** function is useful for comparing fixed-length and variable-length strings, as in the next example:

```
DIM NameRecord AS STRING * 10
NameRecord = "Ed"

NameTest$ = "Ed"

' Use RTRIM$ to trim all the blank spaces from the right
' side of the fixed-length string NameRecord, then compare
' it with the variable-length string NameTest$:
IF RTRIM$(NameRecord) = NameTest$ THEN
   PRINT "They're equal"
ELSE
   PRINT "They're not equal"
END IF
```

Output

```
They're equal
```

4.6.2 Retrieving Characters from the Right Side of a String

The **RIGHT$** and **LTRIM$** functions return characters from the right side of a string. The **RIGHT$**(*string*, *number*) function returns the specified *number* of characters from the right side of the *string*. For example:

```
Test$ = "1234567890"

' Print the rightmost 5 characters from Test$:
PRINT RIGHT$(Test$,5)
```

Output

```
67890
```

The **LTRIM$** function returns the right part of a string after removing any blank spaces at the beginning. For example, contrast the output from the next two **PRINT** statements:

```
PRINT "first"; "      a right-justified string"
PRINT "first"; LTRIM$("      a right-justified string")
```

Output

```
first      a right-justified string
firsta right-justified string
```

4.6.3 Retrieving Characters from Anywhere in a String

Use the **MID$** function to retrieve any number of characters from any point in a string. The **MID$(***string, start, number***)** function returns the specified *number* of characters from the *string*, starting at the character with position *start*. For example, the statement

```
MID$("**over the hill**", 12, 4)
```

starts at the twelfth character (or `h`) of the string

```
**over the hill**
```

and returns that character plus the next three characters (`hill`).

The following example shows how to use the **MID$** function to step through a line of text character by character:

```
.
.
.
' Get the number of characters in the string of text:
Length = LEN(TextString$)

FOR I = 1 TO Length

  ' Get the first character, then the second, third,
  ' and so forth, up to the end of the string:
  Char$ = MID$(TextString$, I, 1)

  ' Evaluate that character:
  .
  .
  .
NEXT
```

4.7 Generating Strings

The easiest way to create a string of one character repeated over and over is to use the intrinsic function **STRING$**. The **STRING$**(*number*, *string*) function produces a new string the specified *number* of characters long, each character of which is the first character of the *string* argument. For example, the statement

```
Filler$ = STRING$(27, "*")
```

generates a string of 27 asterisks. For characters that cannot be produced by typing, such as characters whose ASCII values are greater than 127, use the alternative form of this function, **STRING$**(*number*, *code*). This form creates a string the specified *number* of characters long, each character of which has the ASCII code specified by the *code* argument, as in the next example:

```
' Print a string of 10 "@" characters
' (64 is ASCII code for @):
PRINT STRING$(10, 64)
```

Output

```
@@@@@@@@@@
```

The **SPACE$**(*number*) function generates a string consisting of the specified *number* of blank spaces.

4.8 Changing the Case of Letters

You may want to convert uppercase (capital) letters in a string to lowercase or vice versa. This conversion would be useful in a case-insensitive search for a particular string pattern in a large file (in other words, `help`, `HELP`, or `Help` would all be considered the same). These functions would also be handy when you are not sure whether a user will input text in capital or lowercase letters.

The **UCASE$** and **LCASE$** functions make the following conversions to a string:

- **UCASE$** returns a copy of the string passed to it, with all the lowercase letters converted to uppercase.
- **LCASE$** returns a copy of the string passed to it, with all the uppercase letters converted to lowercase.

Example

```
Sample$ = "* The ASCII Appendix: a table of useful codes *"
PRINT Sample$
PRINT UCASE$(Sample$)
PRINT LCASE$(Sample$)
```

Output

```
* The ASCII Appendix: a table of useful codes *
* THE ASCII APPENDIX: A TABLE OF USEFUL CODES *
* the ascii appendix: a table of useful codes *
```

Letters that are already uppercase, as well as characters that are not letters, remain unchanged by **UCASE$**; similarly, lowercase letters and characters that are not letters are unaffected by **LCASE$**.

4.9 Strings and Numbers

BASIC does not allow a string to be assigned to a numeric variable, nor does it allow a numeric expression to be assigned to a string variable. For example, both of these statements result in an error message reading `Type mismatch`:

```
TempBuffer$ = 45
Counter% = "45"
```

Instead, use the **STR$** function to return the string representation of a number or the **VAL** function to return the numeric representation of a string:

```
' The following statements are both valid:
TempBuffer$ = STR$(45)
Counter%  = VAL("45")
```

Note that **STR$** includes the leading blank space that BASIC prints for positive numbers, as this short example shows:

```
FOR I = 0 TO 9
   PRINT STR$(I);
NEXT
```

Output

```
0 1 2 3 4 5 6 7 8 9
```

If you want to eliminate this space, you can use the **LTRIM$** function, as shown in the example below:

```
FOR I = 0 TO 9
   PRINT LTRIM$(STR$(I));
NEXT
```

Output

```
0123456789
```

Another way to format numeric output is with the **PRINT USING** statement (see Section 3.1, "Printing Text on the Screen").

4.10 Changing Strings

The functions mentioned in each of the preceding sections all leave their string arguments unchanged. Changes are made to a copy of the argument.

In contrast, the **MID$** statement (unlike the **MID$** function discussed in Section 4.6.3, "Retrieving Characters from Anywhere in a String") changes its argument by embedding another string in it. The embedded string replaces part or all of the original string, as shown in the following example:

```
Temp$ = "In the sun."
PRINT Temp$

' Replace the "I" with an "O":
MID$(Temp$,1) = "O"

' Replace "sun." with "road":
MID$(Temp$,8) = "road"
PRINT Temp$
```

Output

```
In the sun.
On the road
```

4.11 Sample Application: Converting a String to a Number (STRTONUM.BAS)

The following program takes a number that is input as a string, filters any numeric characters (such as commas) out of the string, then converts the string to a number with the **VAL** function.

Statements and Functions Used

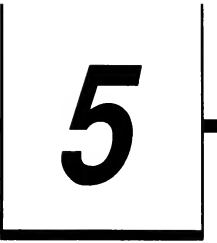
This program demonstrates the following string-handling functions discussed in this chapter:

- **INSTR**
- **LEN**
- **MID$**
- **VAL**

Program Listing

```
DECLARE FUNCTION Filter$ (Txt$, FilterString$)

' Input a line:
LINE INPUT "Enter a number with commas: ", A$

' Look only for valid numeric characters (0123456789.-)
' in the input string:
CleanNum$ = Filter$(A$, "0123456789.-")

' Convert the string to a number:
PRINT "The number's value = " VAL(CleanNum$)
END

' ========================== FILTER =======================
'         Takes unwanted characters out of a string by
'         comparing them with a filter string containing
'         only acceptable numeric characters
' ==========================================================

FUNCTION Filter$ (Txt$, FilterString$) STATIC
   Temp$ = ""
   TxtLength = LEN(Txt$)

   FOR I = 1 TO TxtLength     ' Isolate each character in
      C$ = MID$(Txt$, I, 1)   ' the string.

      ' If the character is in the filter string, save it:
      IF INSTR(FilterString$, C$) <> 0 THEN
         Temp$ = Temp$ + C$
      END IF
   NEXT I

   Filter$ = Temp$
END FUNCTION
```

CHAPTER 5
Graphics



This chapter shows you how to use the graphics statements and functions of BASIC to create a wide variety of shapes, colors, and patterns on your screen. With graphics, you can add a visual dimension to almost any program, whether it's a game, an educational tool, a scientific application, or a financial package.

When you have finished studying this chapter, you will know how to perform the following graphics tasks:

- Use the physical-coordinate system of your personal computer's screen to locate individual pixels, turn those pixels on or off, and change their colors
- Draw lines
- Draw and fill simple shapes, such as circles, ovals, and boxes
- Restrict the area of the screen showing graphics output by using viewports
- Adjust the coordinates used to locate a pixel by redefining screen coordinates
- Use color in graphics output
- Create patterns and use them to fill enclosed figures
- Copy images and reproduce them in another location on the screen
- Animate graphics output

Section 5.1 below contains important information on what you'll need to run the graphics examples shown in this chapter. Read it first.

5.1 What You Need for Graphics Programs

To run the graphics examples shown in this chapter, your computer must have graphics capability, either built in or in the form of a graphics card such as the Color Graphics Adapter (CGA), Enhanced Graphics Adapter (EGA), or Video Graphics Array (VGA). You also need a video display (either monochrome or color) that supports pixel-based graphics.

Also, please keep in mind that these programs all require that your screen be in one of the "screen modes" supporting graphics output. (The screen mode controls the clarity of graphics images, the number of colors available, and the part of the video memory to display.) To select a graphics output mode, use the following statement in your program before using any of the graphics statements or functions described in this chapter:

SCREEN *mode*

Here, *mode* can be either 1, 2, 3, 4, 7, 8, 9, 10, 11, 12, or 13, depending on the monitor/adapter combination installed on your computer.

If you are not sure whether or not the users of your programs have hardware that supports graphics, you can use the following simple test:

```
CONST FALSE = 0, TRUE = NOT FALSE

' Test to make sure user has hardware
' with color/graphics capability:
ON ERROR GOTO Message       ' Turn on error trapping.
SCREEN 1                    ' Try graphics mode one.
ON ERROR GOTO 0             ' Turn off error trapping.
IF NoGraphics THEN END      ' End if no graphics hardware.
.
.
.
END

' Error-handling routine:
Message:
   PRINT "This program requires graphics capability."
   NoGraphics = TRUE
   RESUME NEXT
```

If the user has only a monochrome display with no graphics adapter, the **SCREEN** statement produces an error that in turn triggers a branch to the error-handling routine `Message`. (See Chapter 6, "Error and Event Trapping," for more information on error trapping.)

5.2 Pixels and Screen Coordinates

Shapes and figures on a video display are composed of individual dots of light known as picture elements or "pixels" (or sometimes as "pels") for short. BASIC draws and paints on the screen by turning these pixels on or off and by changing their colors.

A typical screen is composed of a grid of pixels. The exact number of pixels in this grid depends on the hardware you have installed and the screen mode you have selected in the **SCREEN** statement. The larger the number of pixels, the higher the clarity of graphics output. For example, a **SCREEN 1** statement gives a resolution of 320 pixels horizontally by 200 pixels vertically (320 x 200 pixels), while a **SCREEN 2** statement gives a resolution of 640 x 200 pixels. The higher horizontal density in screen mode 2—640 pixels per row versus 320 pixels per row—gives images a sharper, less ragged appearance than they have in screen mode 1.

Depending on the graphics capability of your system, you can use other screen modes that support even higher resolutions (as well as adjust other screen characteristics). Consult the QB Advisor for more information.

When your screen is in one of the graphics modes, you can locate each pixel by means of pairs of coordinates. The first number in each coordinate pair tells the number of pixels from the left side of the screen, while the second number in each pair tells the number of pixels from the top of the screen. For example, in screen mode 2 the point in the extreme upper-left corner of the screen has coordinates (0, 0), the point in the center of the screen has coordinates (320, 100), and the point in the extreme lower-right corner of the screen has coordinates (639, 199), as shown in Figure 5.1.

BASIC uses these screen coordinates to determine where to display graphics (for example, to locate the end points of a line or the center of a circle), as shown in Section 5.3, "Drawing Basic Shapes."

Graphics coordinates differ from text-mode coordinates specified in a **LOCATE** statement. First, **LOCATE** is not as precise: graphics coordinates pinpoint individual pixels on the screen, whereas coordinates used by **LOCATE** are character positions. Second, text-mode coordinates are given in the form "row, column," as in the following:

```
' Move to the 13th row, 15th column,
' then print the message shown:
LOCATE 13, 15
PRINT "This should appear in the middle of the screen."
```

This is the reverse of graphics coordinates, which are given in the form "column, row." A **LOCATE** statement has no effect on the positioning of graphics output on the screen.

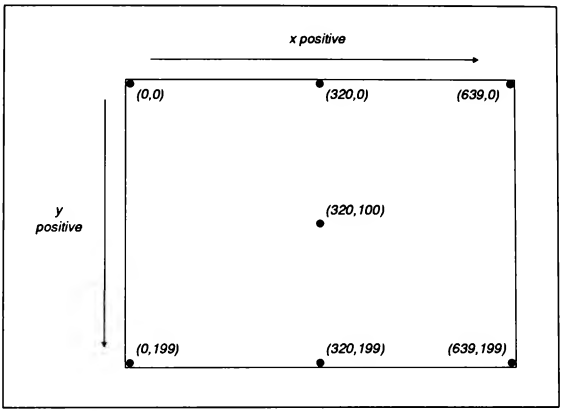


Figure 5.1 Coordinates of Selected Pixels in Screen Mode 2

5.3 Drawing Basic Shapes: Points, Lines, Boxes, and Circles

You can pass coordinate values to BASIC graphics statements to produce a variety of simple figures, as shown in Sections 5.3.1–5.3.2.

5.3.1 Plotting Points with PSET and PRESET

The most fundamental level of control over graphics output is simply turning individual pixels on and off. You do this in BASIC with the **PSET** (for pixel set) and **PRESET** (for pixel reset) statements. The statement **PSET** (*x*, *y*) gives the pixel at location (*x*, *y*) the current foreground color. The **PRESET** (*x*, *y*) statement gives the pixel at location (*x*, *y*) the current background color.

On monochrome monitors, the foreground color is the color that is used for printed text and is typically white, amber, or light green; the background color on monochrome monitors is typically black or dark green. You can choose another color for **PSET** and **PRESET** to use by adding an optional *color* argument. The syntax is then:

PSET (*x*, *y*), *color*

or

PRESET (*x*, *y*), *color*

See Section 5.7 for more information on choosing colors.

Because **PSET** uses the current foreground color by default and **PRESET** uses the current background color by default, **PRESET** without a color argument erases a point drawn with **PSET**, as shown in the next example:

```
SCREEN 2          ' 640 x 200 resolution
PRINT "Press any key to end."

DO

   ' Draw a horizontal line from the left to the right:
   FOR X = 0 TO 640
      PSET (X, 100)
   NEXT

   ' Erase the line from the right to the left:
   FOR X = 640 TO 0 STEP -1
      PRESET (X, 100)
   NEXT

LOOP UNTIL INKEY$ <> ""
END
```

While it is possible to draw any figure using only **PSET** statements to manipulate individual pixels, the output tends to be rather slow, since most pictures consist of many pixels. BASIC has several statements that dramatically increase the speed with which simple figures—such as lines, boxes, and ellipses—are drawn, as shown in Sections 5.3.2 and 5.4.1–5.4.4.

5.3.2 Drawing Lines and Boxes with LINE

When using **PSET** or **PRESET**, you specify only one coordinate pair since you are dealing with only one point on the screen. With **LINE**, you specify two pairs, one for each end of a line segment. The simplest form of the **LINE** statement is as follows:

LINE (*x1*, *y1*) – (*x2*, *y2*)

where (*x1*, *y1*) are the coordinates of one end of a line segment and (*x2*, *y2*) are the coordinates of the other. For example, the following statement draws a straight line from the pixel with coordinates (10, 10) to the pixel with coordinates (150, 200):

```
SCREEN 1
LINE (10, 10)-(150, 200)
```

Note that BASIC does not care about the order of the coordinate pairs: a line from (*x1*, *y1*) to (*x2*, *y2*) is the same as a line from (*x2*, *y2*) to (*x1*, *y1*). This means you could also write the preceding statement as:

```
SCREEN 1
LINE (150, 200)-(10, 10)
```

However, reversing the order of the coordinates could have an effect on graphics statements that follow, as shown in the next section.

5.3.2.1 Using the STEP Option

Up to this point, screen coordinates have been presented as absolute values measuring the horizontal and vertical distances from the extreme upper-left corner of the screen, which has coordinates (0, 0). However, by using the **STEP** option in any of the following graphics statements, you can make the coordinates that follow **STEP** relative to the last point referenced on the screen:

CIRCLE	**GET**
LINE	**PAINT**
PRESET	**PSET**
PUT	

If you picture images as being drawn on the screen by a tiny paintbrush exactly the size of one pixel, then the last point referenced is the location of this paintbrush, or "graphics cursor," when it finishes drawing an image. For example, the following statements leave the graphics cursor at the pixel with coordinates (100, 150):

```
SCREEN 2
LINE (10, 10)-(100, 150)
```

If the next graphics statement in the program is

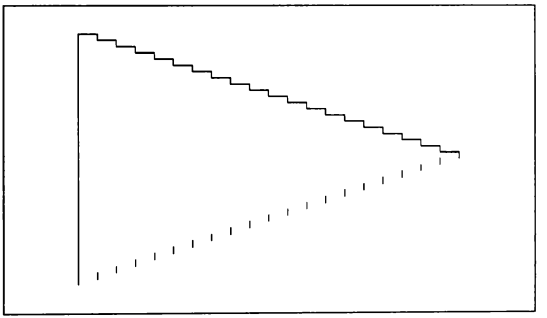
```
PSET STEP(20, 20)
```

then the point plotted by **PSET** is not in the upper-left quadrant of the screen. Instead, the **STEP** option has made the coordinates (20, 20) relative to the last point referenced, which has coordinates (100, 150). This makes the absolute coordinates of the point (100 + 20, 150 + 20) or (120, 170).

In the last example, the last point referenced is determined by a preceding graphics statement. You can also establish a reference point within the same statement, as shown in the next example:

```
' Set 640 x 200 pixel resolution, and make the last
' point referenced the center of the screen:
SCREEN 2

' Draw a line from the lower-left corner of the screen
' to the upper-left corner:
LINE STEP(-310, 100) -STEP(0, -200)

' Draw the "stair steps" down from the upper-left corner
' to the right side of the screen:
FOR I% = 1 TO 20
   LINE -STEP(30, 0)     ' Draw the horizontal line.
   LINE -STEP(0, 5)      ' Draw the vertical line.
NEXT

' Draw the unconnected vertical line segments from the
' right side to the lower-left corner:
FOR I% = 1 TO 20
   LINE STEP(-30, 0) -STEP(0, 5)
NEXT

DO: LOOP WHILE INKEY$ = ""      ' Wait for a keystroke.
```

Output



NOTE *Note the empty* ***DO*** *loop at the end of the last program. If you are running a compiled, stand-alone BASIC program that produces graphics output, your program needs some mechanism like this at the end to hold the output on the screen. Otherwise, it vanishes from the screen before the user has time to notice it.*

5.3.2.2 Drawing Boxes

Using the forms of the **LINE** statement already presented, it is quite easy to write a short program that connects four straight lines to form a box, as shown here:

```
SCREEN 1          ' 320 x 200 pixel resolution

' Draw a box measuring 120 pixels on a side:
LINE (50, 50)-(170, 50)
LINE -STEP(0, 120)
LINE -STEP(-120, 0)
LINE -STEP(0, -120)
```

However, BASIC provides an even simpler way to draw a box, using a single **LINE** statement with the **B** (for box) option. The **B** option is shown in the next example, which produces the same output as the four **LINE** statements in the preceding program:

```
SCREEN 1          ' 320 x 200 pixel resolution

' Draw a box with coordinates (50, 50) for the upper-left
' corner, and (170, 170) for the lower-right corner:
LINE (50, 50)-(170, 170), , B
```

When you add the **B** option, the **LINE** statement no longer connects the two points you specify with a straight line; instead, it draws a rectangle whose opposite corners (upper left and lower right) are at the locations specified.

Two commas precede the **B** in the last example. The first comma functions here as a placeholder for an unused option (the *color* argument), which allows you to pick the color for a line or the sides of a box. (See Section 5.7 for more information on the use of color.)

As before, it does not matter what order the coordinate pairs are given in, so the rectangle from the last example could also be drawn with this statement:

```
LINE (170, 170)-(50, 50), , B
```

Adding the **F** (for fill) option after **B** fills the interior of the box with the same color used to draw the sides. With a monochrome display, this is the same as the foreground color used for printed text. If your hardware has color capabilities, you can change this color with the optional *color* argument (see Section 5.7.1, "Selecting a Color for Graphics Output").

The syntax introduced here for drawing a box is the general syntax used in BASIC to define a rectangular graphics region, and it also appears in the **GET** and **VIEW** statements:

{**GET** | **LINE** | **VIEW** } (*x1*, *y1*) – (*x2*, *y2*), ...

Here, (*x1*, *y1*) and (*x2*, *y2*) are the coordinates of diagonally opposite corners of the rectangle (upper left and lower right). (See Section 5.5, "Defining a Graphics Viewport," for a discussion of **VIEW**, and Section 5.10, "Basic Animation Techniques," for information on **GET** and **PUT**.)

5.3.2.3 Drawing Dotted Lines

The previous sections explain how to use **LINE** to draw solid lines and use them in rectangles; that is, no pixels are skipped. Using yet another option with **LINE**, you can draw dashed or dotted lines instead. This process is known as "line styling." The following is the syntax for drawing a single dashed line from point (*x1*, *y1*) to point (*x2*, *y2*) using the current foreground color:

LINE (*x1*,*y1*) – (*x2*,*y2*), , [[**B**]], *style*

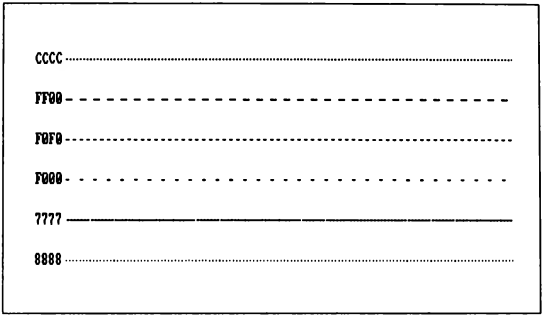
Here *style* is a 16-bit decimal or hexadecimal integer. The **LINE** statement uses the binary representation of the line-style argument to create dashes and blank spaces, with a 1 bit meaning "turn on the pixel," and a 0 bit meaning "leave the pixel off." For example, the hexadecimal integer &HCCCC is equal to the binary integer 1100110011001100, and when used as a *style* argument it draws a line alternating two pixels on, two pixels off.

Example

The following example shows different dashed lines produced using different values for *style*:

```
SCREEN 2          ' 640 x 200 pixel resolution

' Style data:
DATA &HCCCC, &HFF00, &HF0F0
DATA &HF000, &H7777, &H8888

Row% = 4
Column% = 4
XLeft% = 60
XRight% = 600
Y% = 28

FOR I% = 1 TO 6
   READ Style%
   LOCATE Row%, Column%
   PRINT HEX$(Style%)
   LINE (XLeft%, Y%)-(XRight%,Y%), , , Style%
   Row% = Row% + 3
   Y% = Y% + 24
NEXT
```

Output



5.4 Drawing Circles and Ellipses with CIRCLE

The **CIRCLE** statement draws a variety of circular and elliptical, or oval, shapes. In addition, **CIRCLE** draws arcs (segments of circles), and pie-shaped wedges. In graphics mode you can produce just about any kind of curved line with some variation of **CIRCLE**.

5.4.1 Drawing Circles

To draw a circle, you need to know only two things: the location of its center and the length of its radius (the distance from the center to any point on the circle). With this information and a reasonably steady hand (or better yet, a compass), you can produce an attractive circle.

Similarly, BASIC needs only the location of a circle's center and the length of its radius to draw a circle. The simplest form of the **CIRCLE** syntax is

CIRCLE [[**STEP**]] (*x*, *y*), *radius*

where *x*, *y* are the coordinates of the center, and *radius* is the radius of the circle. The next example draws a circle with center (200, 100) and radius 75:

```
SCREEN 2
CIRCLE (200, 100), 75
```

You could rewrite the preceding example as follows, making the same circle but using the **STEP** option to make the coordinates relative to the center rather than to the upper-left corner:

```
SCREEN 2        ' Uses center of screen (320,100) as the
                ' reference point for the CIRCLE statement:
CIRCLE STEP(-120, 0), 75
```

5.4.2 Drawing Ellipses

The **CIRCLE** statement automatically adjusts the "aspect ratio" to make sure that circles appear round and not flattened on your screen. However, you may need to adjust the aspect ratio to make circles come out right on your monitor, or you may want to change the aspect ratio to draw the oval figure known as an ellipse. In either case, use the syntax

CIRCLE [[STEP]] (*x*,*y*), *radius*, , , *aspect*

where *aspect* is a positive real number. (See Section 5.4.5 for more information on the aspect ratio and how to calculate it for different screen modes.)

The extra commas between *radius* and *aspect* are placeholders for other options that tell **CIRCLE** which color to use (if you have a color monitor/adapter and are using one of the screen modes that support color), or whether to draw an arc or wedge. (See Sections 5.4.3, "Drawing Arcs," and 5.7.1, "Selecting a Color for Graphics Output," for more information on these options.)

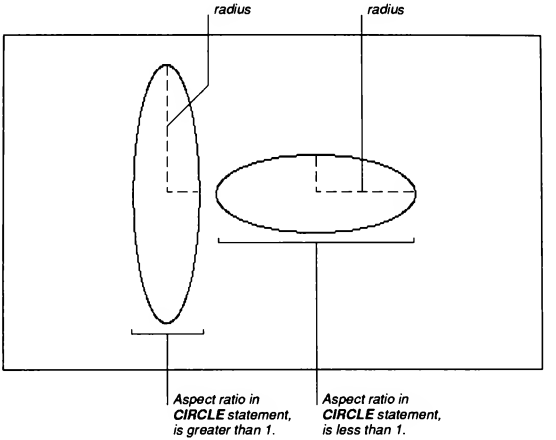
Since the *aspect* argument specifies the ratio of the vertical to horizontal dimensions, large values for *aspect* produce ellipses stretched out along the vertical axis, while small values for *aspect* produce ellipses stretched out along the horizontal axis. Since an ellipse has two radii—one horizontal *x*-radius and one vertical *y*-radius—BASIC uses the single *radius* argument in a **CIRCLE** statement as follows: if *aspect* is less than one, then *radius* is the *x*-radius; if *aspect* is greater than or equal to one, then *radius* is the *y*-radius.

Example

The following example and its output show how different *aspect* values affect whether the **CIRCLE** statement uses the *radius* argument as the *x*-radius or the *y*-radius of an ellipse:

```
SCREEN 1

' This draws the ellipse on the left:
CIRCLE (60, 100), 80, , , , 3

' This draws the ellipse on the right:
CIRCLE (180, 100), 80, , , , 3/10
```

Output



5.4.3 Drawing Arcs

An arc is a segment of a ellipse, in other words a short, curved line. To understand how the **CIRCLE** statement draws arcs, you need to know how BASIC measures angles.

BASIC uses the radian as its unit of angle measure, not only in the **CIRCLE** statement, but also in the intrinsic trigonometric functions such as **COS**, **SIN**, or **TAN**. (The one exception to this use of radians is the **DRAW** statement, which expects angle measurements in degrees. See Section 5.9 for more information about **DRAW**.)

The radian is closely related to the radius of a circle. In fact, the word "radian" is derived from the word "radius." The circumference of a circle equals $2 * \pi *$ *radius*, where π is equal to approximately 3.14159265. Similarly, the number of radians in one complete angle of revolution (or 360) equals $2 * \pi$, or a little more than 6.28. If you are more used to thinking of angles in terms of degrees, here are some common equivalences:

Angle in Degrees	Angle in Radians
360	2π (approximately 6.283)
180	π (approximately 3.142)
90	π/2 (approximately 1.571)
60	π/3 (approximately 1.047)

If you picture a clock face on the screen, **CIRCLE** measures angles by starting at the "3:00" position and rotating counterclockwise, as shown in Figure 5.2:

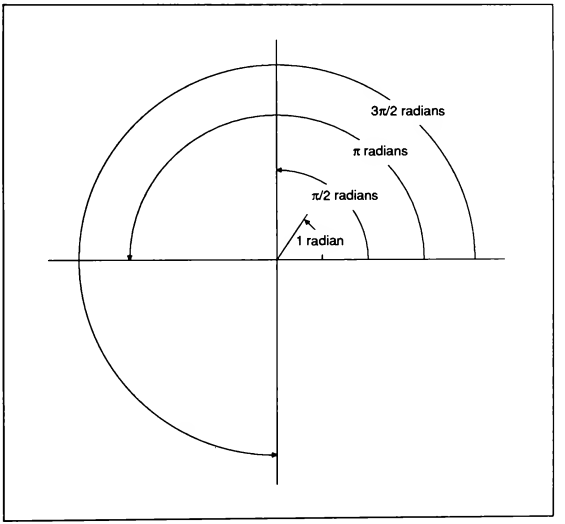


Figure 5.2 How Angles Are Measured for CIRCLE

The general formula for converting from degrees to radians is to multiply degrees by π/180.

To draw an arc, give angle arguments defining the arc's limits:

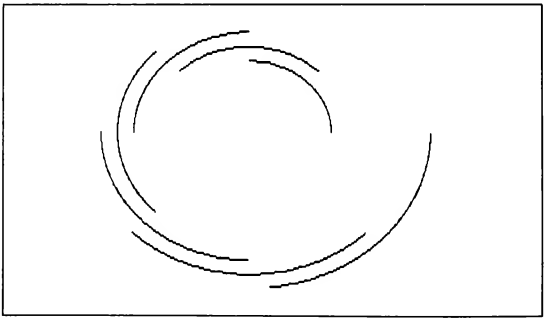
CIRCLE [[**STEP**]] (*x*, *y*), *radius*, [[*color*]], *start*, *end* [[, *aspect*]]

Example

The **CIRCLE** statements in the next example draw seven arcs, with the innermost arc starting at the "3:00" position (0 radians) and the outermost arc starting at the "6:00" position ($3\pi/2$ radians), as you can see from the output:

```
SCREEN 2
CLS

CONST PI = 3.141592653589#      ' Double-precision constant

StartAngle = 0
FOR Radius% = 100 TO 220 STEP 20
   EndAngle = StartAngle + (PI / 2.01)
   CIRCLE (320, 100), Radius%, , StartAngle, EndAngle
   StartAngle = StartAngle + (PI / 4)
NEXT Radius%
```

Output



5.4.4 Drawing Pie Shapes

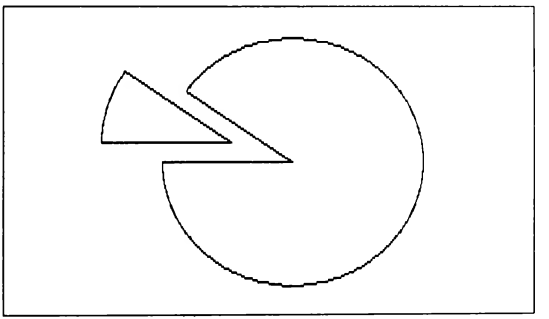
By making either of **CIRCLE**'s *start* or *end* arguments negative, you can connect the arc at its beginning or ending point with the center of the circle. By making both arguments negative, you can draw shapes ranging from a wedge that resembles a slice of pie to the pie itself with the piece missing.

Example

This example draws a pie shape with a piece missing:

```
SCREEN 2

CONST RADIUS = 150, PI = 3.141592653589#

StartAngle = 2.5
EndAngle = PI

' Draw the wedge:
CIRCLE (320, 100), RADIUS, , -StartAngle, -EndAngle

' Swap the values for the start and end angles:
SWAP StartAngle, EndAngle

' Move the center 10 pixels down and 70 pixels to the
' right, then draw the "pie" with the wedge missing:
CIRCLE STEP(70, 10), RADIUS, , -StartAngle, -EndAngle
```

Output



5.4.5 Drawing Shapes to Proportion with the Aspect Ratio

As discussed in Section 5.4.2, "Drawing Ellipses," BASIC's **CIRCLE** statement automatically corrects the aspect ratio, which determines how figures are scaled on the screen. However, with other graphics statements you need to scale horizontal and vertical dimensions yourself to make shapes appear with correct proportions. For example, although the following statement draws a rectangle that measures 100 pixels on all sides, it does not look like a square:

```
SCREEN 1
LINE (0, 0)-(100, 100), , B
```

In fact, this is not an optical illusion; the rectangle really is taller than it is wide. This is because in screen mode 1 there is more space between pixels vertically than horizontally. To draw a perfect square, you have to change the aspect ratio.

The aspect ratio is defined as follows: in a given screen mode consider two lines, one vertical and one horizontal, that appear to have the same length. The aspect ratio is the number of pixels in the vertical line divided by the number of pixels in the horizontal line. This ratio depends on two factors:

1. Because of the way pixels are spaced on most screens, a horizontal row has more pixels than a vertical column of the exact same physical length in all screen modes except modes 11 and 12.
2. The standard personal computer's video-display screen is wider than it is high. Typically, the ratio of screen width to screen height is 4:3.

To see how these two factors interact to produce the aspect ratio, consider a screen after a **SCREEN 1** statement, which gives a resolution of 320 x 200 pixels. If you draw a rectangle from the top of the screen to the bottom, and from the left side of the screen three-fourths of the way across, you have a square, as shown in Figure 5.3.

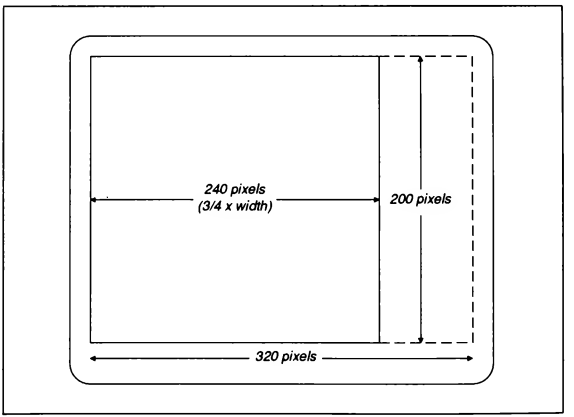


Figure 5.3 The Aspect Ratio in Screen Mode 1

As you can see from the diagram, this square has a height of 200 pixels and a width of 240 pixels. The ratio of the square's height to its width (200 / 240, or when simplified, 5 / 6) is the aspect ratio for this screen resolution. In other words, to draw a square in 320 x 200 resolution, make its height in pixels equal to 5 / 6 times its width in pixels, as shown in the next example:

```
SCREEN 1          ' 320 x 200 pixel resolution

' The height of this box is 100 pixels, and the width is
' 120 pixels, which makes the ratio of the height to the
' width equal to 100/120, or 5/6. The result is a square:
LINE (50, 50) -STEP(120, 100), , B
```

The formula for calculating the aspect ratio for a given screen mode is

(4 / 3) * (*ypixels* / *xpixels*)

where *xpixels* by *ypixels* is the current screen resolution. In screen mode 1, this formula shows the aspect ratio to be (4 / 3) * (200 / 320), or 5 / 6; in screen mode 2, the aspect ratio is (4 / 3) * (200 / 640), or 5 / 12.

If you have a video display with a ratio of width to height that is not equal to 4:3, use the more general form of the formula for computing the aspect ratio:

(*screenwidth* / *screenheight*) * (*ypixels* / *xpixels*)

The **CIRCLE** statement can be made to draw an ellipse by varying the value of the *aspect* argument, as shown above in Section 5.4.2.

5.5 Defining a Graphics Viewport

The graphics examples presented so far have all used the entire video-display screen as their drawing board, with absolute coordinate distances measured from the extreme upper-left corner of the screen.

However, using the **VIEW** statement you can also define a kind of miniature screen (known as a "graphics viewport") inside the boundaries of the physical screen. Once it is defined, all graphics operations take place within this viewport. Any graphics output outside the viewport boundaries is "clipped"; that is, any attempt to plot a point outside the viewport is ignored. There are two main advantages to using a viewport:

1. A viewport makes it easy to change the size and position of the screen area where graphics appear.
2. Using **CLS 1**, you can clear the screen inside a viewport without disturbing the screen outside the viewport.

NOTE *Refer to Section 3.1.6 to learn how to create a "text viewport" for output printed on the screen.*

The general syntax for **VIEW** (note that the **STEP** option is not allowed with **VIEW**) is

VIEW [[**SCREEN**] (*x1*, *y1*) – (*x2*, *y2*)[, [*color*][, [*border*]]]]

where (*x1*, *y1*) and (*x2*, *y2*) define the corners of the viewport, using the standard BASIC syntax for rectangles (see Section 5.3.2.2, "Drawing Boxes"). The optional *color* and *border* arguments allow you to choose a color for the interior and edges, respectively, of the viewport rectangle. See Section 5.7 below for more information on setting and changing colors.

The **VIEW** statement without arguments makes the entire screen the viewport. Without the **SCREEN** option, the **VIEW** statement makes all pixel coordinates relative to the viewport, rather than the full screen. In other words, after the statement

```
VIEW (50, 60)-(150, 175)
```

the pixel plotted with

```
PSET (10, 10)
```

is visible, since it is 10 pixels below and 10 pixels to the right of the upper-left corner of the viewport. Note that this makes the pixel's absolute screen coordinates (50 + 10, 60 + 10) or (60, 70).

In contrast, the **VIEW** statement with the **SCREEN** option keeps all coordinates absolute; that is, coordinates measure distances from the side of the screen, not from the sides of the viewport. Therefore, after the statement

```
VIEW SCREEN (50, 60)-(150, 175)
```

the pixel plotted with

```
PSET (10, 10)
```

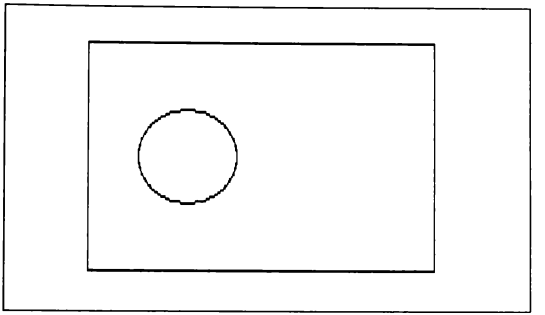
is not visible, since it is 10 pixels below and 10 pixels to the right of the upper-left corner of the screen—outside the viewport.

Examples

The output from the next two examples should further clarify the distinction between **VIEW** and **VIEW SCREEN**:

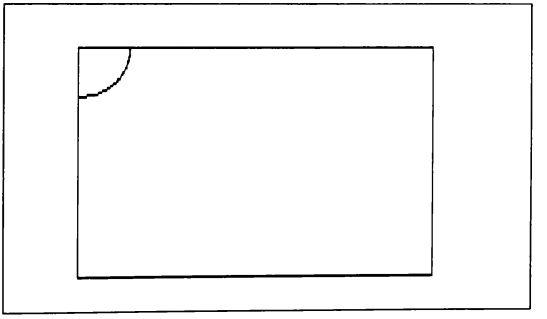
```
SCREEN 2

VIEW (100, 50)-(450, 150), , 1

' This circle's center point has absolute coordinates
' (100 + 100, 50 + 50), or (200, 100):
CIRCLE (100, 50), 50
```

Output using VIEW



```
SCREEN 2

' This circle's center point has absolute coordinates
' (100, 50), so only part of the circle appears
'  within the view port:
VIEW SCREEN (100, 50)-(450, 150), , 1
CIRCLE (100, 50), 50
```

Output using VIEW SCREEN



Note that graphics output located outside the current viewport is clipped by the viewport's edges and does not appear on the screen.

5.6 Redefining Viewport Coordinates with WINDOW

This section shows you how to use the **WINDOW** statement and your own coordinate system to redefine pixel coordinates.

In Sections 5.2–5.5, the coordinates used to locate pixels on the screen represent actual physical distances from the upper-left corner of the screen (or the upper-left corner of the current viewport, if it has been defined with a **VIEW** statement). These are known as "physical coordinates." The "origin," or reference point, for physical coordinates is always the upper-left corner of the screen or viewport, which has coordinates (0, 0).

As you move down the screen and to the right, *x* values (horizontal coordinates) and *y* values (vertical coordinates) get bigger, as shown in the upper diagram of Figure 5.4. While this scheme is standard for video displays, it may seem counterintuitive if you have used other coordinate systems to draw graphs. For example, on the Cartesian grid used in mathematics, *y* values get bigger toward the top of a graph and smaller toward the bottom.

With BASIC's **WINDOW** statement, you can change the way pixels are addressed to use any coordinate system you choose, thus freeing you from the limitations of using physical coordinates.

The general syntax for **WINDOW** is

WINDOW [[[**SCREEN**]] (*x1*, *y1*) – (*x2*, *y2*)]]

where *y1*, *y2*, *x1*, and *x2* are real numbers specifying the top, bottom, left, and right sides of the window, respectively. These numbers are known as "view coordinates." For example, the following statement remaps your screen so that it is bounded on the top and bottom by the lines `y = 10` and `y = -15` and on the left and right by the lines `x = -25` and `x = 5`:

```
WINDOW (-25, -15)-(5, 10)
```

After a **WINDOW** statement, *y* values get bigger toward the top of the screen. In contrast, after a **WINDOW SCREEN** statement, *y* values get bigger toward the bottom of the screen. The bottom half of Figure 5.4 shows the effects of both a **WINDOW** statement and a **WINDOW SCREEN** statement on a line drawn in screen mode 2. Note also how both of these statements change the coordinates of the screen corners. A **WINDOW** statement with no arguments restores the regular physical-coordinate system.

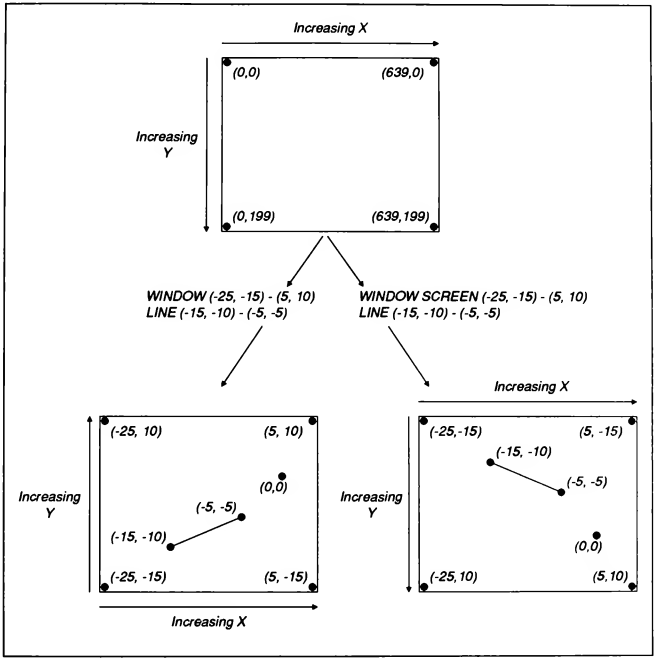


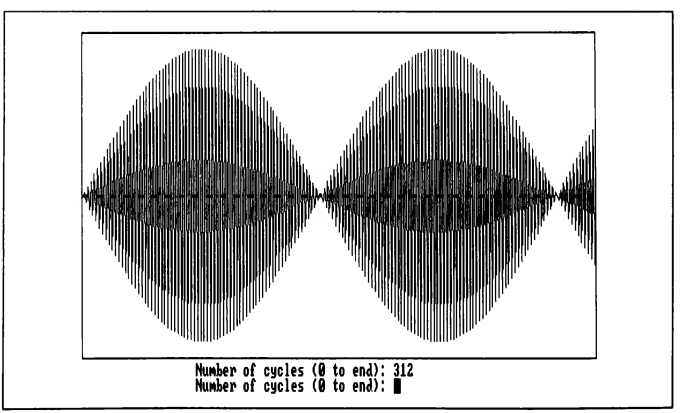
Figure 5.4 WINDOW Contrasted with WINDOW SCREEN

Example

The following example uses both **VIEW** and **WINDOW** to simplify writing a program to graph the sine-wave function for angle values from 0 radians to π radians (or 0° to 180°). This program is in the file named SINEWAVE.BAS on the QuickBASIC distribution disks.

```
SCREEN 2

' Viewport sized to proper scale for graph:
VIEW (20, 2)-(620, 172), , 1
CONST PI = 3.141592653589#

' Make window large enough to graph sine wave from
' 0 radians to pi radians:
WINDOW (0, -1.1)-(PI, 1.1)
Style% = &HFF00                 ' Use to make dashed line.
VIEW PRINT 23 TO 24  ' Scroll printed output in rows 23, 24.
DO
   PRINT TAB(20);
   INPUT "Number of cycles (0 to end): ", Cycles
   CLS
   LINE (PI, 0)-(0, 0), , , Style%' Draw the x axis.
   IF Cycles > 0 THEN

      ' Start at (0,0) and plot the graph:
      FOR X = 0 TO PI STEP .01
         Y = SIN(Cycles * X)     ' Calculate the y coordinate.
         LINE -(X, Y)            ' Draw a line to new point.
      NEXT X
   END IF
LOOP WHILE Cycles > 0
```

Output



5.6.1 The Order of Coordinate Pairs

As with the other BASIC graphics statements that define rectangular areas (**GET**, **LINE**, and **VIEW**), the order of coordinate pairs in a **WINDOW** statement is unimportant. The first pair of statements below has the same effect as the second pair of statements:

```
VIEW (100, 20)-(300, 120)
WINDOW (-4, -3)-(0, 0)

VIEW (300, 120)-(100, 20)
WINDOW (0, 0)-(-4, -3)
```

5.6.2 Keeping Track of View and Physical Coordinates

The **PMAP** and **POINT** functions are useful for keeping track of physical and view coordinates. **POINT**(*number*) tells you the current location of the graphics cursor by returning either the physical or view coordinates (depending on the value for *number*) of the last point referenced in a graphics statement. **PMAP** allows you to translate physical coordinates to view coordinates and vice versa. The physical-coordinate values returned by **PMAP** are always relative to the current viewport.

Examples

The following example shows the different values that are returned by **POINT** (*number*) for *number* values of 0, 1, 2, or 3:

```
SCREEN 2
' Define the view-coordinate window:
WINDOW (-10, -30)-(-5, -10)
' Draw a line from the point with view coordinates (-9,-28)
' to the point with view coordinates (-6,-24):
LINE (-9, -28)-(-6, -24)

PRINT "Physical x-coordinate of the last point = " POINT(0)
PRINT "Physical y-coordinate of the last point = " POINT(1)
PRINT
PRINT "View x-coordinate of the last point  = " POINT(2)
PRINT "View y-coordinate of the last point  = " POINT(3)

END
```

Output

```
Physical x-coordinate of the last point = 511
Physical y-coordinate of the last point = 139

View x-coordinate of the last point  = -6
View y-coordinate of the last point  = -24
```

Given the **WINDOW** statement in the preceding example, the next four **PMAP** statements would print the output that follows:

```
' Map the view x-coordinate -6 to physical x and print:
PhysX% = PMAP(-6, 0)
PRINT PhysX%

' Map the view y-coordinate -24 to physical y and print:
PhysY% = PMAP(-24, 1)
PRINT PhysY%

' Map physical x back to view x and print:
ViewX% = PMAP(PhysX%, 2)
PRINT ViewX%

' Map physical y back to view y and print:
ViewY% = PMAP(PhysY%, 3)
PRINT ViewY%
```

Output

```
 511
 139
-6
-24
```

5.7 Using Colors

If you have a Color Graphics Adapter (CGA), you can choose between the following two graphics modes only:

1. Screen mode 2 has 640 x 200 high resolution, with only one foreground and one background color. This is known as "monochrome," since all graphics output has the same color.
2. Screen mode 1 has 320 x 200 medium resolution with 4 foreground colors and 16 background colors.

There is thus a tradeoff between color and clarity in the two screen modes supported by most color-graphics display adapter hardware. Depending on the graphics capability of your system, you may not have to sacrifice clarity to get a full range of color. However, this section focuses on screen modes 1 and 2.

5.7.1 Selecting a Color for Graphics Output

The following list shows where to put the *color* argument in the graphics statements discussed in previous sections of this chapter. This list also shows other options (such as **BF** with the **LINE** statement or *border* with the **VIEW** statement) that can have a different colors. (Please note that these do not give the complete syntax for some of these statements. This summary is intended to show how to use the *color* option in those statements that accept it.)

PSET (*x*, *y*), *color*
PRESET (*x*, *y*), *color*
LINE (*x1*, *y1*) - (*x2*, *y2*), *color* [, **B**[**F**]]
CIRCLE (*x*, *y*), *radius*, *color*
VIEW (*x1*, *y1*) - (*x2*, *y2*), *color*, *border*

In screen mode 1, the *color* argument is a numeric expression with the value 0, 1, 2, or 3. Each of these values, known as an "attribute," represents a different color, as demonstrated by the following program:

```
' Draw an "invisible" line (same color as background):
LINE (10, 10)-(310, 10), 0

' Draw a light blue (cyan) line:
LINE (10, 30)-(310, 30), 1

' Draw a purple (magenta) line:
LINE (10, 50)-(310, 50), 2

' Draw a white line:
LINE (10, 70)-(310, 70), 3
END
```

As noted in the comments for the preceding example, a *color* value of 0 produces no visible output, since it is always equal to the current background color. At first glance, this may not seem like such a useful color value, but in fact it is useful for erasing a figure without having to clear the entire screen or viewport, as shown in the next example:

```
SCREEN 1

CIRCLE (100, 100), 80, 2, , , 3   ' Draw an ellipse.
Pause$ = INPUT$(1)                ' Wait for a key press.
CIRCLE (100, 100), 80, 0, , , 3   ' Erase the ellipse.
```

5.7.2 Changing the Foreground or Background Color

Section 5.7.1 above describes how to use 4 different foreground colors for graphics output. You have a wider variety of colors in screen mode 1 for the screen's background: 16 in all.

In addition, you can change the foreground color by using a different "palette." In screen mode 1, there are two palettes, or groups of four colors. Each palette assigns a different color to the same attribute; so, for instance, in palette 1 (the default) the color associated with attribute 2 is magenta, while in palette 0 the color associated with attribute 2 is red. If you have a CGA, these colors are predetermined for each palette; that is, the color assigned to number 2 in palette 1 is always magenta, while the color assigned to number 2 in palette 0 is always red.

If you have an Enhanced Graphics Adapter (EGA) or Video Graphics Adapter (VGA), you can use the **PALETTE** statement to choose the color displayed by any attribute. For example, by changing arguments in a **PALETTE** statement, you could make the color displayed by attribute 1 green one time and brown the next. (See Section 5.7.3, "Changing Colors with PALETTE and PALETTE USING," in this manual for more information on reassigning colors.)

In screen mode 1, the **COLOR** statement allows you to control both the background color and the palette for the foreground colors. Here is the syntax for **COLOR** in screen mode 1:

COLOR [[*background*]] [[, *palette*]]

The *background* argument is a numeric expression from 0 to 15, and *palette* is a numeric expression equal to either 0 or 1.

Table 5.1 shows the colors produced with the 4 different foreground numbers in each of the two palettes, while Table 5.2 shows the colors produced with the 16 different background numbers.

Table 5.1 Color Palettes in Screen Mode 1

Foreground Color Number	Color In Palette 0	Color In Palette 1
0	Current background color	Current background color
1	Green	Cyan (bluish green)
2	Red	Magenta (light purple)
3	Brown	White (light grey on some monitors)

Table 5.2 Background Colors in Screen Mode 1

Background Color Number	Color
0	Black
1	Blue
2	Green
3	Cyan
4	Red
5	Magenta
6	Brown (dark yellow on some monitors)
7	White (light grey on some monitors)
8	Dark grey (black on some monitors)
9	Light blue
10	Light green
11	Light cyan
12	Light red
13	Light magenta
14	Light yellow (may have greenish tinge on some monitors)
15	Bright white or very light grey

Example

The following program shows all combinations of the two color palettes with the 16 different background screen colors. This program is in the file named COLORS.BAS on the QuickBASIC distribution disks.

```
SCREEN 1

Esc$ = CHR$(27)
' Draw three boxes and paint the interior
' of each box with a different color:
FOR ColorVal = 1 TO 3
   LINE (X, Y) -STEP(60, 50), ColorVal, BF
   X = X + 61
   Y = Y + 51
NEXT ColorVal

LOCATE 21, 1
PRINT "Press ESC to end."
PRINT "Press any other key to continue."

' Restrict additional printed output to the 23rd line:
```

```
VIEW PRINT 23 TO 23
DO
   PaletteVal = 1
   DO

      ' PaletteVal is either 1 or 0:
      PaletteVal = 1 - PaletteVal

      ' Set the background color and choose the palette:
      COLOR BackGroundVal, PaletteVal
      PRINT "Background ="; BackGroundVal;
      PRINT "Palette ="; PaletteVal;

      Pause$ = INPUT$(1)          ' Wait for a keystroke.
      PRINT

   ' Exit the loop if both palettes have been shown,
   ' or if the user pressed the ESC key:
   LOOP UNTIL PaletteVal = 1 OR Pause$ = Esc$

   BackGroundVal = BackGroundVal + 1

' Exit this loop if all 16 background colors have
' been shown, or if the user pressed the ESC key:
LOOP UNTIL BackGroundVal > 15 OR Pause$ = Esc$

SCREEN 0                      ' Restore text mode and
WIDTH 80                      ' 80-column screen width.
```

5.7.3 Changing Colors with PALETTE and PALETTE USING

The preceding section showed how you can change the color displayed by an attribute simply by specifying a different palette in the **COLOR** statement. However, this restricts you to two fixed color palettes, with just 4 colors in each. Furthermore, each attribute can stand for only one of two possible colors; for example, attribute 1 can signify only green or cyan.

With an EGA or VGA, your choices are potentially much broader. (If you don't have an EGA or VGA, you may want to skip this section.) For instance, depending on the amount of video memory available to your computer, with a VGA you can choose from a palette with as many as 256K (that's right, over 256,000) colors and assign those colors to 256 different attributes. Even an EGA allows you to display up to 16 different colors from a palette of 64 colors.

In contrast to the **COLOR** statement, the **PALETTE** and **PALETTE USING** statements give you a lot more flexibility in manipulating the available color palette. Using these statements, you can assign any color from the palette to any attribute. For example, after the following statement, the output of all graphics statements using attribute 4 appears in light magenta (color 13):

```
PALETTE 4, 13
```

This color change is instantaneous and affects not only subsequent graphics statements but any output already on the screen. In other words, you can draw and paint your screen, then switch the palette to achieve an immediate change of color, as shown by the following example:

```
SCREEN 8
LINE (50, 50)-(150, 150), 4   ' Draws a line in red.
SLEEP 1                       ' Pauses program.
PALETTE 4, 13                 ' Attribute 4 now means color
                              ' 13, so the line drawn in the
                              ' last statement is now light
                              ' magenta.
```

With the **PALETTE** statement's **USING** option, you can change the colors assigned to every attribute all at once.

Example

The following example uses the **PALETTE USING** statement to give the illusion of movement on the screen by constantly rotating the colors displayed by attributes 1 through 15. This program is in the file named PALETTE.BAS on the QuickBASIC distribution disks.

```
DECLARE SUB InitPalette ()
DECLARE SUB ChangePalette ()
DECLARE SUB DrawEllipses ()

DEFINT A-Z
DIM SHARED PaletteArray(15)

SCREEN 8                    ' 640 x 200 resolution; 16 colors

InitPalette                 ' Initialize PaletteArray.
DrawEllipses                ' Draw and paint concentric ellipses.

DO                          ' Shift the palette until a key
   ChangePalette            ' is pressed.
LOOP WHILE INKEY$ = ""

END


' ====================== InitPalette =======================
'    This procedure initializes the integer array used to
'    change the palette.
' ==========================================================
                                                          '

SUB InitPalette STATIC
   FOR I = 0 TO 15
      PaletteArray(I) = I
   NEXT I
END SUB
```

```
' ===================== DrawEllipses =====================
'    This procedure draws 15 concentric ellipses and
'    paints the interior of each with a different color.
' ========================================================

SUB DrawEllipses STATIC
   CONST ASPECT = 1 / 3
   FOR ColorVal = 15 TO 1 STEP -1
      Radius = 20 * ColorVal
      CIRCLE (320, 100), Radius, ColorVal, , , ASPECT
      PAINT (320, 100), ColorVal
   NEXT
END SUB


' ===================== ChangePalette ====================
'  This procedure rotates the palette by one each time it
'  is called. For example, after the first call to
'  ChangePalette, PaletteArray(1) = 2, PaletteArray(2) = 3,
'  . . . , PaletteArray(14) = 15, and PaletteArray(15) = 1
' ========================================================

SUB ChangePalette STATIC
   FOR I = 1 TO 15
      PaletteArray(I) = (PaletteArray(I) MOD 15) + 1
   NEXT I
   PALETTE USING PaletteArray(0) ' Shift the color displayed
                                 ' by each of the attributes.
END SUB
```

5.8 Painting Shapes

Section 5.3.2.2 above shows how to draw a box with the **LINE** statement's **B** option, then paint the box by appending the **F** (for fill) option:

```
SCREEN 1

' Draw a square, then paint the interior with color 1
' (cyan in the default palette):
LINE (50, 50)-(110, 100), 1, BF
```

With BASIC's **PAINT** statement, you can fill any enclosed figure with any color you choose. **PAINT** also allows you to fill enclosed figures with your own custom patterns, such as stripes or checks, as shown in Section 5.8.2, "Painting with Patterns."

5.8.1 Painting with Colors

To paint an enclosed shape with a solid color, use this form of the **PAINT** statement:

PAINT [[**STEP**]](*x*, *y*) [[, [[*interior*]], [[*border*]]]]

Here, *x*, *y* are the coordinates of a point in the interior of the figure you want to paint, *interior* is the number for the color you want to paint with, and *border* is the color number for the outline of the figure.

For example, the following program lines draw a circle in cyan, then paint the inside of the circle magenta:

```
SCREEN 1
CIRCLE (160, 100), 50, 1
PAINT (160, 100), 2, 1
```

The following three rules apply when painting figures:

1. The coordinates given in the **PAINT** statement must refer to a point inside the figure.

   For example, any one of the following statements would have the same effect as the **PAINT** statements shown in the two preceding examples, since all of the coordinates identify points in the interior of the circle:

   ```
   PAINT (150, 90), 2, 1
   PAINT (170, 110), 2, 1
   PAINT (180, 80), 2, 1
   ```

   In contrast, since `(5, 5)` identifies a point outside the circle, the next statement would paint all of the screen except the inside of the circle, leaving it colored with the current background color:

   ```
   PAINT (5, 5), 2, 1
   ```

   If the coordinates in a **PAINT** statement specify a point right on the border of the figure, then no painting occurs:

   ```
   LINE (50, 50)-(150, 150), , B  ' Draw a box.
   PAINT (50, 100)  ' The point with coordinates
                    ' (50, 100) is on the top edge of the
                    ' box, so no painting occurs.
   ```

2. The figure must be completely enclosed; otherwise, the paint color will "leak out," filling the entire screen or viewport (or any larger figure completely enclosing the first one).

   For example, in the following program, the **CIRCLE** statement draws an ellipse that is not quite complete (there is a small gap on the right side); the **LINE** statement then encloses the partial ellipse inside a box. Even though painting starts in the interior of the ellipse, the paint color flows through the gap and fills the entire box.

```
SCREEN 2
CONST PI = 3.141592653589#
CIRCLE (300, 100), 80, , 0, 1.9 * PI, 3
LINE (200, 10)-(400, 190), , B
PAINT (300, 100)
```

3. If you are painting an object a different color from the one used to outline the object, you must use the *border* option to tell **PAINT** where to stop painting.

   For example, the following program draws a triangle outlined in green (attribute 1 in palette 0) and then tries to paint the interior of the triangle red (attribute 2). However, since the **PAINT** statement doesn't indicate where to stop painting, it paints the entire screen red.

```
SCREEN 1
COLOR , 0

LINE (10, 25)-(310, 25), 1
LINE -(160, 175), 1
LINE -(10, 25), 1

PAINT (160, 100), 2
```

   Making the following change to the **PAINT** statement (choose red for the interior and stop when a border colored green is reached) produces the desired effect:

```
PAINT (160, 100), 2, 1
```

   Note that you don't have to specify a border color in the **PAINT** statement if the paint color is the same as the border color.

```
LINE (10, 25)-(310, 25), 1
LINE -(160, 175), 1
LINE -(10, 25), 1

PAINT (160, 100), 1
```

5.8.2 Painting with Patterns: Tiling

You can use the **PAINT** statement to fill any enclosed figure with a pattern; this process is known as "tiling." A "tile" is the pattern's basic building block. The process is identical to laying down tiles on a floor. When you use tiling, the argument *interior* in the syntax for **PAINT** is a string expression, rather than a number. While *interior* can be any string expression, a convenient way to define tile patterns uses the following form for *interior*:

CHR$**(*arg1*) + **CHR$(*arg2*) + **CHR$**(*arg3*) + ... + **CHR$**(*argn*)

Here, *arg1*, *arg2*, and so forth are eight-bit integers. See Sections 5.8.2.2–5.8.2.4 below for an explanation of how these eight-bit integers are derived.

5.8.2.1 Pattern-Tile Size in Different Screen Modes

Each tile for a pattern is composed of a rectangular grid of pixels. This tile grid can have up to 64 rows in all screen modes. However, the number of pixels in each row depends on the screen mode.

Here is why the length of each tile row varies according to the screen mode: although the number of bits in each row is fixed at eight (the length of an integer), the number of pixels these eight bits can represent decreases as the number of color attributes in a given screen mode increases. For example, in screen mode 2, which has only 1 color attribute, the number of bits per pixel is 1; in screen mode 1, which has 4 different attributes, the number of bits per pixel is 2; and in the EGA screen mode 7, which has 16 attributes, the number of bits per pixel is 4. The following formula allows you to compute the bits per pixel in any given screen mode:

$bits\text{-}per\text{-}pixel = \log_2(numattributes)$

Here, *numattributes* is the number of color attributes in that screen mode. (The on-line QB Advisor has this information.)

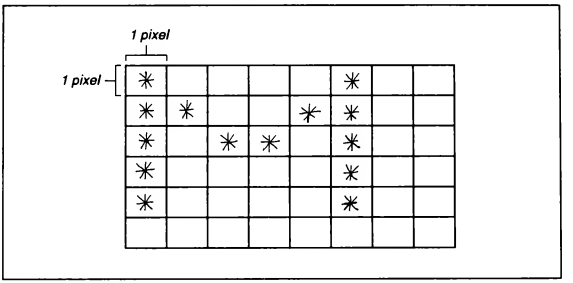
Thus, the length of a tile row is eight pixels in screen mode 2 (eight bits divided by one bit per pixel), but only four pixels in screen mode 1 (eight bits divided by two bits per pixel).

The next three sections show the step-by-step process involved in creating a pattern tile. Section 5.8.2.2 shows how to make a monochrome pattern in screen mode 2. Next, Section 5.8.2.3 shows how to make a multicolored pattern in screen mode 1. Finally, if you have an EGA, read Section 5.8.2.4 to see how to make a multicolored pattern in screen mode 8.

5.8.2.2 Creating a Single-Color Pattern in Screen Mode 2

The following steps show how to define and use a pattern tile that resembles the letter "M":

1. Draw the pattern for a tile in a grid with 8 columns and however many rows you need (up to 64). In this example, the tile has 6 rows; an asterisk (*) in a box means the pixel is on:



2. Next, translate each row of pixels to an eight-bit number, with a 1 meaning the pixel is on, and a 0 meaning the pixel is off:

1	0	0	0	0	1	0	0
1	1	0	0	1	1	0	0
1	0	1	1	0	1	0	0
1	0	0	0	0	1	0	0
1	0	0	0	0	1	0	0
0	0	0	0	0	0	0	0

3. Convert the binary numbers given in step 2 to hexadecimal integers:

```
10000100 = &H84
11001100 = &HCC
10110100 = &HB4
10000100 = &H84
10000100 = &H84
00000000 = &H00
```

These integers do not have to be hexadecimal; they could be decimal or octal. However, binary to hexadecimal conversion easier. To convert from binary to hexadecimal, read the binary number from right to left. Each group of four digits is then converted to its hexadecimal equivalent, as shown here:

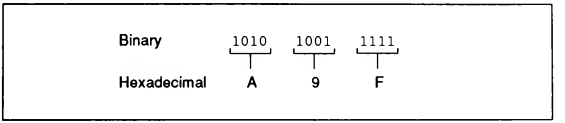


Table 5.3 lists four-bit binary sequences and their hexadecimal equivalents.

4. Create a string by concatenating the characters with the ASCII values from step 3 (use the **CHR$** function to get these characters):

```
Tile$ = CHR$(&H84) + CHR$(&HCC) + CHR$(&HB4)
Tile$ = Tile$ + CHR$(&H84) + CHR$(&H84) + CHR$(&H00)
```

5. Draw a figure and paint its interior, using **PAINT** and the string argument from step 4:

```
PAINT (X, Y), Tile$
```

Table 5.3 Binary to Hexadecimal Conversion

Binary Number	Hexadecimal Number
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	8

Table 5.3 (*continued*)

Binary Number	Hexadecimal Number
1001	9
1010	A
1011	B
1100	C
1101	D
1110	E
1111	F

Example

The following example draws a circle and then paints the circle's interior with the pattern created in the preceding steps:

```
SCREEN 2
CLS
Tile$ = CHR$(&H84) + CHR$(&HCC) + CHR$(&HB4)
Tile$ = Tile$ + CHR$(&H84) + CHR$(&H84) + CHR$(&H00)
CIRCLE STEP(0, 0), 150
PAINT STEP(0, 0), Tile$
```

Output

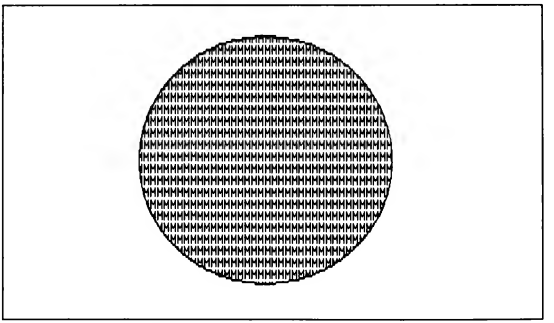
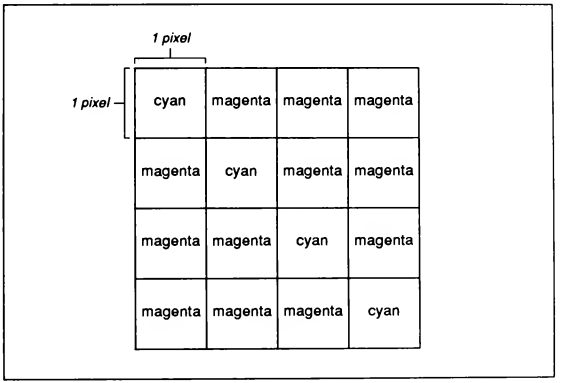


Figure 5.5 Patterned Circle

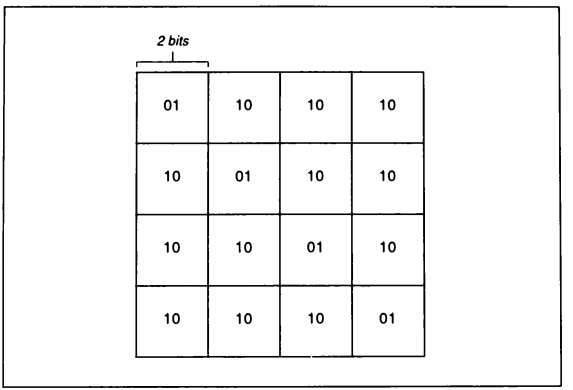
5.8.2.3 Creating a Multicolor Pattern in Screen Mode 1

The following steps show how to create a multicolor pattern consisting of alternating diagonal stripes of cyan and magenta (or green and red in palette 0):

1. Draw the pattern for a tile in a grid with four columns (four columns because each row of pixels is stored in an eight-bit integer and each pixel in screen mode 1 requires two bits) and however many rows you need (up to 64). In this example, the tile has four rows, as shown in the next diagram:



2. Convert the colors to their respective color numbers in binary notation, as shown below (be sure to use two-bit values, so 1 that = binary 01 and 2 = binary 10):



3. Convert the binary numbers from step 2 to hexadecimal integers:

```
01101010 = &H6A
10011010 = &H9A
10100110 = &HA6
10101001 = &HA9
```

4. Create a string by concatenating the characters with the ASCII values from step 3 (use the **CHR$** function to get these characters):

```
Tile$ = CHR$(&H6A) + CHR$(&H9A) + CHR$(&HA6) + CHR$(&HA9)
```

5. Draw a figure and paint its interior using **PAINT** and the string argument from step 4:

```
PAINT (X, Y), Tile$
```

The following program draws a triangle and then paints its interior with the pattern created in the preceding steps:

```
SCREEN 1

' Define a pattern:
Tile$ = CHR$(&H6A) + CHR$(&H9A) + CHR$(&HA6) + CHR$(&HA9)

' Draw a triangle in white (color 3):
LINE (10, 25)-(310, 25)
LINE -(160, 175)
LINE -(10, 25)

' Paint the interior of the triangle with the pattern:
PAINT (160, 100), Tile$
```

Note that if the figure you want to paint is outlined in a color that is also contained in the pattern, then you must give the *border* argument with **PAINT** as shown below; otherwise, the pattern spills over the edges of the figure:

```
SCREEN 1

' Define a pattern:
Tile$ = CHR$(&H6A) + CHR$(&H9A) + CHR$(&HA6) + CHR$(&HA9)

' Draw a triangle in magenta (color 2):
LINE (10, 25)-(310, 25), 2
LINE -(160, 175), 2
LINE -(10, 25), 2

' Paint the interior of the triangle with the pattern,
' adding the border argument (, 2) to tell PAINT
' where to stop:
PAINT (160, 100), Tile$, 2
```

Sometimes, after painting a figure with a solid color or pattern, you may want to repaint that figure, or some part of it, with a new pattern. If the new pattern contains two or more adjacent rows that are the same as the figure's current background, you will find that tiling does not work. Instead, the pattern starts to spread, finds itself surrounded by pixels that are the same as two or more of its rows, then stops.

You can alleviate this problem by using the *background* argument with **PAINT** if there are at most two adjacent rows in your new pattern that are the same as the old background. **PAINT** with *background* has the following syntax:

PAINT [**STEP**] (*x*, *y*) [,[*interior*] [,[*border*] [, *background*]]]

The *background* argument is a string character of the form **CHR$(*n*)** that specifies the rows in the pattern tile that are the same as the figure's current background. In essence, *background* tells **PAINT** to skip over these rows when repainting the figure. The next example clarifies how this works:

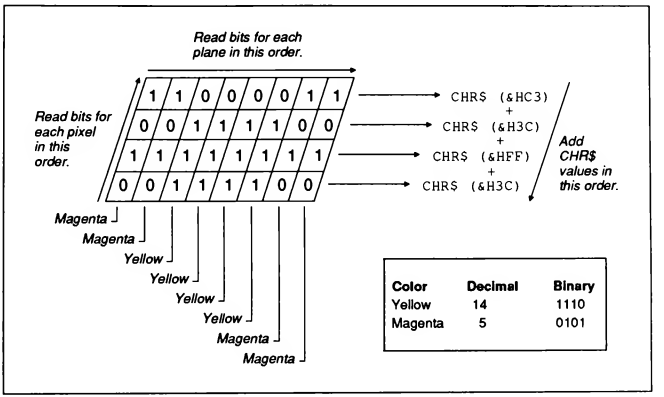
```
SCREEN 1

' Define a pattern (two rows each of cyan, magenta, white):
Tile$ = CHR$(&H55) + CHR$(&H55) + CHR$(&HAA)
Tile$ = Tile$ + CHR$(&HAA) + CHR$(&HFF) + CHR$(&HFF)

' Draw a triangle in white (color number 3):
LINE (10, 25)-(310, 25)
LINE -(160, 175)
LINE -(10, 25)

' Paint the interior magenta:
PAINT (160, 100), 2, 3

' Wait for a keystroke:
Pause$ = INPUT$(1)

' Since the background is already magenta, CHR$(&HAA) tells
' PAINT to skip over the magenta rows in the pattern tile:
PAINT (160, 100), Tile$, , CHR$(&HAA)
```

5.8.2.4 Creating a Multicolor Pattern in Screen Mode 8

In the EGA and VGA screen modes, it takes more than one eight-bit integer to define one row in a pattern tile. In these screen modes, a row is composed of several layers of eight-bit integers. This is because a pixel is represented three dimensionally in a stack of "bit planes" rather than sequentially in a single plane, as is the case with screen modes 1 and 2. For example, screen mode 8 has four of these bit planes. Each of the four bits per pixel in this screen mode is on a different plane.

The following steps diagram the process for creating a multicolor pattern consisting of rows of alternating yellow and magenta. Note how each row in the pattern tile is represented by four parallel bytes:

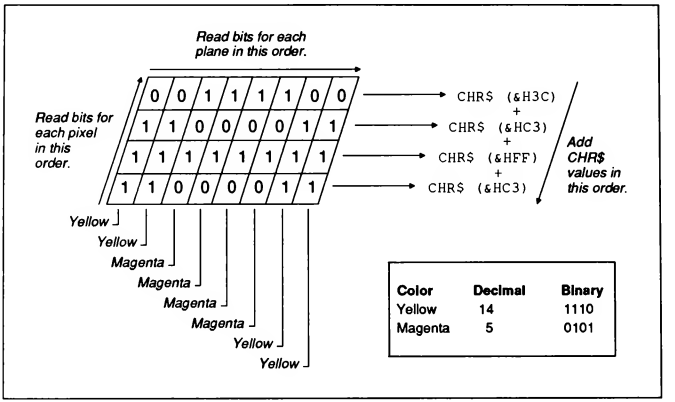
1. Define one row of pixels in the pattern tile. Each pixel in the row takes four bits, and each bit is in a different plane, as shown below:



Add the **CHR$** values for all four bit planes to get one tile byte. This row is repeated in the pattern tile, so

Row$(1) = Row$(2) =
CHR$**(&HC3) + **CHR$(&H3C) + **CHR$**(&HFF) + **CHR$**(&H3C)

2. Define another row of pixels in the pattern tile, as follows:



This row is also repeated in the pattern tile, so

Row$(3) = Row$(4) =
CHR$**(&H3C) + **CHR$(&HC3) + **CHR$**(&HFF) + **CHR$**(&HC3)

Example

The following example draws a box, then paints its interior with the pattern created in the preceding steps:

```
SCREEN 8
DIM Row$(1 TO 4)

' Two rows of alternating magenta and yellow:
Row$(1) = CHR$(&HC3) + CHR$(&H3C) + CHR$(&HFF) + CHR$(&H3C)
Row$(2) = Row$(1)

' Invert the pattern (two rows of alternating yellow
' and magenta):
Row$(3) = CHR$(&H3C) + CHR$(&HC3) + CHR$(&HFF) + CHR$(&HC3)
Row$(4) = Row$(3)
```

```
' Create a pattern tile from the rows defined above:
FOR I% = 1 TO 4
   Tile$ = Tile$ + Row$(I%)
NEXT I%

' Draw box and fill it with the pattern:
LINE (50, 50)-(570, 150), , B
PAINT (320, 100), Tile$
```

5.9 DRAW: a Graphics Macro Language

The **DRAW** statement is a miniature language by itself. It draws and paints images on the screen using a set of one- or two-letter commands, known as "macros," embedded in a string expression.

DRAW offers the following advantages over the other graphics statements discussed so far:

- The macro string argument to **DRAW** is compact: a single, short string can produce the same output as several **LINE** statements.
- Images created with **DRAW** can easily be scaled—that is, enlarged or reduced in size—by using the **S** macro in the macro string.
- Images created with **DRAW** can be rotated any number of degrees by using the **TA** macro in the macro string.

Consult the QB Advisor for more information.

Example

The following program gives a brief introduction to the movement macros **U**, **D**, **L**, **R**, **E**, **F**, **G**, and **H**; the "plot/don't plot" macro **B**; and the color macro **C**. This program draws horizontal, vertical, and diagonal lines in different colors, depending on which DIRECTION key on the numeric keypad (UP, DOWN, LEFT, PGUP, PGDN, and so on) is pressed.

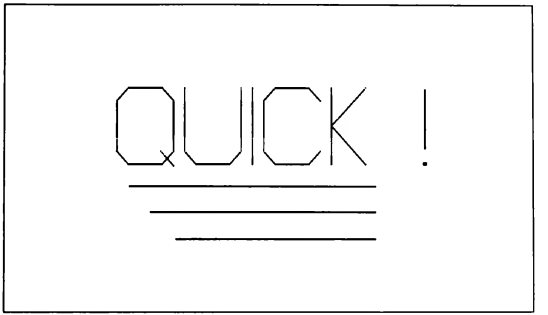
This program is in the file named PLOTTER.BAS on the QuickBASIC distribution disks.

```
' Values for keys on the numeric keypad and the spacebar:
CONST UP = 72, DOWN = 80, LFT = 75, RGHT = 77
CONST UPLFT = 71, UPRGHT = 73, DOWNLFT = 79, DOWNRGHT = 81
CONST SPACEBAR = " "

' Null$ is the first character of the two-character INKEY$
' value returned for direction keys such as UP and DOWN:
Null$ = CHR$(0)
```

```
' Plot$ = "" means draw lines; Plot$ = "B" means
' move graphics cursor, but don't draw lines:
Plot$ = ""

PRINT "Use the cursor movement keys to draw lines."
PRINT "Press spacebar to toggle line drawing on and off."
PRINT "Press <ENTER> to begin. Press q to end the program."
DO : LOOP WHILE INKEY$ = ""

SCREEN 1

DO
   SELECT CASE KeyVal$
      CASE Null$ + CHR$(UP)
         DRAW Plot$ + "C1 U2"
      CASE Null$ + CHR$(DOWN)
         DRAW Plot$ + "C1 D2"
      CASE Null$ + CHR$(LFT)
         DRAW Plot$ + "C2 L2"
      CASE Null$ + CHR$(RGHT)
         DRAW Plot$ + "C2 R2"
      CASE Null$ + CHR$(UPLFT)
         DRAW Plot$ + "C3 H2"
      CASE Null$ + CHR$(UPRGHT)
         DRAW Plot$ + "C3 E2"
      CASE Null$ + CHR$(DOWNLFT)
         DRAW Plot$ + "C3 G2"
      CASE Null$ + CHR$(DOWNRGHT)
         DRAW Plot$ + "C3 F2"
      CASE SPACEBAR
         IF Plot$ = "" THEN Plot$ = "B " ELSE Plot$ = ""
      CASE ELSE
         ' The user pressed some key other than one of the
         ' direction keys, the spacebar, or "q," so
         ' don't do anything.
   END SELECT

   KeyVal$ = INKEY$

LOOP UNTIL KeyVal$ = "q"

SCREEN 0, 0             ' Restore the screen to 80-column
WIDTH 80                ' text mode and end.
END
```

Output

Here's a sample sketch created with this program.



5.10 Basic Animation Techniques

Using only the graphics statements covered in earlier sections, you can do simple animation of objects on the screen. For instance, you can first draw a circle with **CIRCLE**, then redraw it with the background color to erase it, and finally recalculate the circle's center point and draw it in a new location.

This technique works well enough for simple figures, but its disadvantages become apparent when animating more complex images. Even though the graphics statements are very fast, you can still notice the lag. Moreover, it is not possible to preserve the background with this method: when you erase the object, you also erase whatever was already on the screen.

Two statements allow you to do high-speed animation: **GET** and **PUT**. You can create an image using output from statements such as **PSET**, **LINE**, **CIRCLE**, or **PAINT**, then take a "snapshot" of that image with **GET**, copying the image to memory. With **PUT**, you can then reproduce the image stored with **GET** anywhere on the screen or viewport.

5.10.1 Saving Images with GET

After you have created the original image on the screen, you need to calculate the *x*- and *y*-coordinates of a rectangle large enough to hold the entire image. You then use **GET** to copy the entire rectangle to memory. The syntax for the graphics **GET** statement is

GET [[**STEP**]](*x1*, *y1*) – [[**STEP**]](*x2*, *y2*), *array-name*

where (*x1*, *y1*) and (*x2*, *y2*) give the coordinates of a rectangle's upper-left and lower-right corners. The argument *array-name* refers to any numeric array. The size specified in a **DIM** statement for *array-name* depends on the following three factors:

1. The height and width of the rectangle enclosing the image
2. The screen mode chosen for graphics output
3. The type of the array (integer, long integer, single precision, or double precision)

NOTE *Although the array used to store images can have any numeric type, it is strongly recommended that you use only integer arrays. All possible graphics patterns on the screen can be represented by integers. This is not the case, however, with single-precision or double-precision real numbers: some graphics patterns are not valid real numbers, and it could lead to unforeseen results if these patterns were stored in a real-number array.*

The formula for calculating the size in bytes of *arrayname* is

size-in-bytes = 4 + *height* * *planes* * **INT**((*width* * *bits-per-pixel*/*planes* + 7)/8)

where *height* and *width* are the dimensions, in number of pixels, of the rectangle to get, and the value for *bits-per-pixel* depends on the number of colors available in the given screen mode. More colors mean more bits are required to define each pixel. In screen mode 1, two bits define a pixel, while in screen mode 2, one bit is enough. (See Section 5.8.2.1 above, "Pattern-Tile Size in Different Screen Modes," for the general formula for *bits-per-pixel*.) The following list shows the value for *planes* for each of the screen modes:

Screen Mode	Number of Bit Planes
1, 2, 11, and 13	1
9 (64K of video memory) and 10	2
7, 8, 9 (more than 64K of video memory), and 12	4

To get the number of elements that should be in the array, divide the *size-in-bytes* by the number of bytes for one element in the array. This is where the type of the array comes into play. If it is an integer array, each element takes two bytes of memory (the size of an integer), so *size-in-bytes* should be divided by two to get the actual size of the array. Similarly, if it is a long integer array, *size-in-bytes* should be divided by four (since one long integer requires four bytes of memory), and so on. If it is single precision, divide by four; if it is double precision, divide by eight.

The following steps show how to calculate the size of an integer array large enough to hold a rectangle in screen mode 1 with coordinates (10, 40) for the upper-left corner and (90, 80) for the lower-right corner:

1. Calculate the height and width of the rectangle:

   RectHeight = **ABS**(*y2* – *y1*) + 1 = 80 – 40 + 1 = 41
   RectWidth = **ABS**(*x2* – *x1*) + 1 = 90 – 10 + 1 = 81

   Remember to add one after subtracting *y1* from *y2* or *x1* from *x2*. For example, if *x1* = 10 and *x2* = 20, then the width of the rectangle is 20 – 10 + 1, or 11.

2. Calculate the size in bytes of the integer array:

   ByteSize = 4 + RectHeight * **INT**((RectWidth * BitsPerPixel + 7) / 8)
   = 4 + 41 * **INT**((81 * 2 + 7) / 8)
   = 4 + 41 * **INT**(169 / 8)
   = 4 + 41 * 21
   = 865

3. Divide the size in bytes by the bytes per element (two for integers) and round the result up to the nearest whole number:

   865 / 2 = 433

Therefore, if the name of the array is `Image`, the following **DIM** statement ensures that `Image` is big enough to copy the pixel information in the rectangle:

```
DIM Image (1 TO 433) AS INTEGER
```

NOTE *Although the* ***GET*** *statement uses view coordinates after a* ***WINDOW*** *statement, you must use physical coordinates to calculate the size of the array used in* ***GET****. (See Section 5.6 above, "Redefining Viewport Coordinates with WINDOW," for more information on* ***WINDOW*** *and how to convert view coordinates to physical coordinates.)*

Note that the steps outlined above give the minimum size required for the array; however, any larger size will do. For example, the following statement also works:

```
DIM Image (1 TO 500) AS INTEGER
```

Example

The following program draws an ellipse and paints its interior. A **GET** statement copies the rectangular area containing the ellipse into memory. (Section 5.10.2 below, "Moving Images with PUT," shows how to use the **PUT** statement to reproduce the ellipse in a different location.)

```
SCREEN 1

' Dimension an integer array large enough
' to hold the rectangle:
DIM Image (1 TO 433) AS INTEGER

' Draw an ellipse inside the rectangle, using magenta for
' the outline and painting the interior white:
CIRCLE (50, 60), 40, 2, , , .5
PAINT (50, 60), 3, 2

' Store the image of the rectangle in the array:
GET (10, 40)-(90, 80), Image
```

5.10.2 Moving Images with PUT

While the **GET** statement allows you to take a snapshot of an image, **PUT** allows you to paste that image anywhere you want on the screen. A statement of the form

PUT (*x*, *y*), *array-name* [[, *actionverb*]]

copies the rectangular image stored in *array-name* back to the screen and places its upper-left corner at the point with coordinates (*x*, *y*). Note that only one coordinate pair appears in **PUT**.

If a **WINDOW** statement appears in the program before **PUT**, the coordinates *x* and *y* refer to the lower-left corner of the rectangle. **WINDOW SCREEN**, however, does not have this effect; that is, after **WINDOW SCREEN**, *x* and *y* still refer to the upper-left corner of the rectangle.

For example, adding the next line to the last example in Section 5.10.1 above causes an exact duplicate of the painted ellipse to appear on the right side of the screen much more quickly than redrawing and repainting the same figure with **CIRCLE** and **PAINT**:

```
PUT (200, 40), Image
```

Take care not to specify coordinates that would put any part of the image outside the screen or active viewport, as in the following statements:

```
SCREEN 2
.
.
.
' Rectangle measures 141 pixels x 91 pixels:
GET (10, 10)-(150, 100), Image
PUT (510, 120), Image
```

Unlike other graphics statements such as **LINE** or **CIRCLE**, **PUT** does not clip images lying outside the viewport. Instead, it produces an error message reading `Illegal function call`.

One of the other advantages of the **PUT** statement is that you can control how the stored image interacts with what is already on the screen by using the argument *actionverb*. The *actionverb* argument can be one of the following options: **PSET, PRESET, AND, OR,** or **XOR**.

If you do not care what happens to the existing screen background, use the **PSET** option, since it transfers an exact duplicate of the stored image to the screen and overwrites anything that was already there.

Table 5.4 shows how other options affect the way the **PUT** statement causes pixels in a stored image to interact with pixels on the screen. In this table, a 1 means a pixel is on and a 0 means a pixel is off.

Table 5.4 The Effect of Different Action Options in Screen Mode 2

Action Option	Pixel in Stored Image	Pixel on Screen before PUT Statement	Pixel on Screen after PUT Statement
PSET	0	0	0
	0	1	0
	1	0	1
	1	1	1
PRESET	0	0	1
	0	1	1
	1	0	0
	1	1	0
AND	0	0	0
	0	1	0
	1	0	0
	1	1	1

Table 5.4 (*continued*)

Action Option	Pixel in Stored Image	Pixel on Screen before PUT Statement	Pixel on Screen after PUT Statement
OR	0	0	0
	0	1	1
	1	0	1
	1	1	1
XOR	0	0	0
	0	1	1
	1	0	1
	1	1	0

As you can see, these options cause a **PUT** statement to treat pixels the same way logical operators treat numbers. The **PRESET** option is like the logical operator **NOT** in that it inverts the pixels in the stored image, regardless of what was on the screen. The options **AND**, **OR**, and **XOR** are identical to the logical operators with the same names; for example, the logical operation

```
1 XOR 1
```

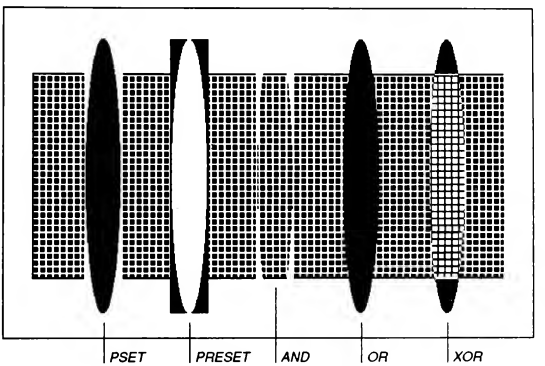
gives 0 as its result, just as using the **XOR** option turns a pixel off when both the pixel in the image and the pixel in the background are on.

The output from the following program shows the same image superimposed over a filled rectangle using each of the five options discussed above:

```
SCREEN 2

DIM CircImage (1 TO 485) AS INTEGER

' Draw and paint an ellipse then store its image with GET:
CIRCLE (22, 100), 80, , , , 4
PAINT (22, 100)
GET (0, 20)-(44, 180), CircImage
CLS

' Draw a box and fill it with a pattern:
LINE (40, 40)-(600, 160), , B
Pattern$ = CHR$(126) + CHR$(0) + CHR$(126) + CHR$(126)
PAINT (50, 50), Pattern$
```

```
' Put the images of the ellipse over the box
' using the different action options:
PUT (100, 20), CircImage, PSET
PUT (200, 20), CircImage, PRESET
PUT (300, 20), CircImage, AND
PUT (400, 20), CircImage, OR
PUT (500, 20), CircImage, XOR
```

Output



In screen modes supporting color, the options **PRESET**, **AND**, **OR**, and **XOR** produce a more complicated interaction, since color involves more than simply turning a pixel on or off. However, the analogy made above between these options and logical operators still holds in these modes. For example, if the current pixel on the screen is color 1 (cyan in palette 1) and the pixel in the overlaid image is color 2 (magenta in palette 1), then the color of the resulting pixel after a **PUT** statement depends on the option, as shown for just 2 of the 16 different combinations of image color and background color in Table 5.5.

Table 5.5 The Effect of Different Action Options on Color in Screen Mode 1 (Palette 1)

Action Option	Pixel Color in Stored Image	Pixel Color on Screen before PUT Statement	Pixel Color on Screen after PUT Statement
PSET	10 (magenta)	01 (cyan)	10 (magenta)
PRESET	10 (magenta)	01 (cyan)	01 (cyan)
AND	10 (magenta)	01 (cyan)	00 (black)
OR	10 (magenta)	01 (cyan)	11 (white)
XOR	10 (magenta)	01 (cyan)	11 (white)

In palette 1, cyan is assigned to attribute 1 (01 binary), magenta is assigned to attribute 2 (10 binary), and white is assigned to attribute 3 (11 binary). If you have an EGA, you can use the **PALETTE** statement to assign different colors to the attributes 1, 2, and 3.

5.10.3 Animation with GET and PUT

One of the most useful things that can be done with the **GET** and **PUT** statements is animation. The two options best suited for animation are **XOR** and **PSET**. Animation done with **PSET** is faster; but as shown by the output from the last program, **PSET** erases the screen background. In contrast, **XOR** is slower but restores the screen background after the image is moved. Animation with **XOR** is done with the following four steps:

1. Put the object on the screen with **XOR**.
2. Recalculate the new position of the object.
3. Put the object on the screen a second time at the old location, using **XOR** again, this time to remove the old image.
4. Go to step 1, but this time put the object at the new location.

Movement done with these four steps leaves the background unchanged after step 3. Flicker can be reduced by minimizing the time between steps 4 and 1 and by making sure that there is enough time delay between steps 1 and 3. If more than one object is being animated, every object should be processed at once, one step at a time.

If it is not important to preserve the background, use the **PSET** option for animation. The idea is to leave a border around the image when you copy it with the **GET** statement. If this border is as large or larger than the maximum distance the object will move, then each time the image is put in a new location, the

border erases all traces of the image in the old location. This method can be somewhat faster than the method using **XOR** described above, since only one **PUT** statement is required to move an object (although you must move a larger image).

Examples

The following example shows how to use **PUT** with the **PSET** option to produce the effect of a ball bouncing off the bottom and sides of a box. Note in the output that follows how the rectangle containing the ball, specified in the **GET** statement, erases the filled box and the printed message.

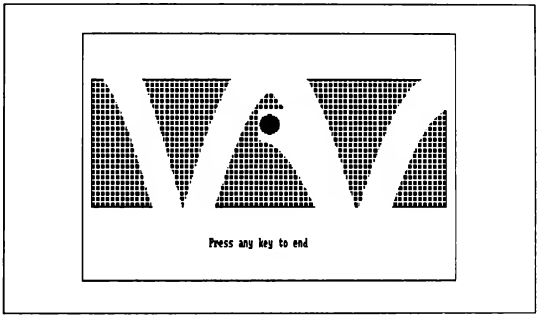
This program is in the file named BALLPSET.BAS on the QuickBASIC distribution disks.

```
DECLARE FUNCTION GetArraySize (WLeft, WRight, WTop, WBottom)

SCREEN 2

' Define a viewport and draw a border around it:
VIEW (20, 10)-(620, 190),,1

CONST PI = 3.141592653589#

' Redefine the coordinates of the viewport with view
' coordinates:
WINDOW (-3.15, -.14)-(3.56, 1.01)

' Arrays in program are now dynamic:
' $DYNAMIC

' Calculate the view coordinates for the top and bottom of a
' rectangle large enough to hold the image that will be
' drawn with CIRCLE and PAINT:
WLeft = -.21
WRight = .21
WTop = .07
WBottom = -.07

' Call the GetArraySize function,
' passing it the rectangle's view coordinates:
ArraySize% = GetArraySize(WLeft, WRight, WTop, WBottom)

DIM Array (1 TO ArraySize%) AS INTEGER

' Draw and paint the circle:
CIRCLE (0, 0), .18
PAINT (0, 0)

' Store the rectangle in Array:
GET (WLeft, WTop)-(WRight, WBottom), Array
CLS
```

```
' Draw a box and fill it with a pattern:
LINE (-3, .8)-(3.4, .2), , B
Pattern$ = CHR$(126) + CHR$(0) + CHR$(126) + CHR$(126)
PAINT (0, .5), Pattern$

LOCATE 21, 29
PRINT "Press any key to end."

' Initialize loop variables:
StepSize = .02
StartLoop = -PI
Decay = 1

DO
   EndLoop = -StartLoop
   FOR X = StartLoop TO EndLoop STEP StepSize

      ' Each time the ball "bounces" (hits the bottom of the
      ' viewport), the Decay variable gets smaller, making
      ' the height of the next bounce smaller:
      Y = ABS(COS(X)) * Decay - .14
      IF Y < -.13 THEN Decay = Decay * .9

      ' Stop if key pressed or Decay less than .01:
      Esc$ = INKEY$
      IF Esc$ <> "" OR Decay < .01 THEN EXIT FOR

      ' Put the image on the screen. The StepSize offset is
      ' smaller than the border around the circle. Thus,
      ' each time the image moves, it erases any traces
      ' left from the previous PUT (and also erases anything
      ' else on the screen):
      PUT (X, Y), Array, PSET
   NEXT X

   ' Reverse direction:
   StepSize = -StepSize
   StartLoop = -StartLoop
LOOP UNTIL Esc$ <> "" OR Decay < .01

END

FUNCTION GetArraySize (WLeft, WRight, WTop, WBottom) STATIC

   ' Map the view coordinates passed to this function to
   ' their physical-coordinate equivalents:
   VLeft = PMAP(WLeft, 0)
   VRight = PMAP(WRight, 0)
   VTop = PMAP(WTop, 1)
   VBottom = PMAP(WBottom, 1)
```

```
' Calculate the height and width in pixels
   ' of the enclosing rectangle:
   RectHeight = ABS(VBottom - VTop) + 1
   RectWidth = ABS(VRight - VLeft) + 1

   ' Calculate size in bytes of array:
   ByteSize = 4 + RectHeight * INT((RectWidth + 7) / 8)

   ' Array is integer, so divide bytes by two:
   GetArraySize = ByteSize \ 2 + 1
END FUNCTION
```

Output

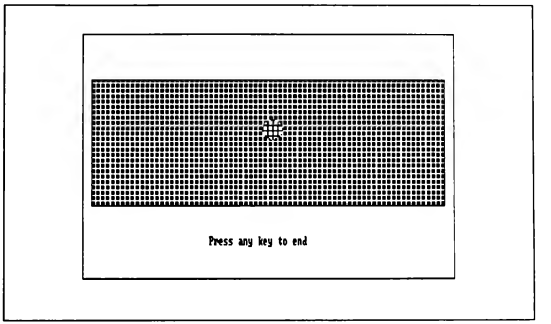


Contrast the preceding program with the next program, which uses **PUT** with **XOR** to preserve the screen's background, according to the four steps outlined above. Note how the rectangle containing the ball is smaller than in the preceding program, since it is not necessary to leave a border. Also note that two **PUT** statements are required, one to make the image visible and another to make it disappear. Finally, observe the empty **FOR...NEXT** delay loop between the **PUT** statements; this loop reduces the flicker that results from the image appearing and disappearing too rapidly.

This program is in the file named BALLXOR.BAS on the QuickBASIC distribution disks.

```
.
.
.
' The rectangle is smaller than the one in the previous
' program, which means Array is also smaller:
WLeft = -.18
WRight = .18
WTop = .05
WBottom = -.05
.
.
.
DO
   EndLoop = -StartLoop
   FOR X = StartLoop TO EndLoop STEP StepSize
      Y = ABS(COS(X)) * Decay - .14

      ' The first PUT statement places the image
      ' on the screen:
      PUT (X,Y), Array, XOR

      ' Use an empty FOR...NEXT loop to delay
      ' the program and reduce image flicker:
      FOR I = 1 TO 5: NEXT I

      IF Y < -.13 THEN Decay = Decay * .9
      Esc$ = INKEY$
      IF Esc$ <> "" OR Decay < .01 THEN EXIT FOR

      ' The second PUT statement erases the image and
      ' restores the background:
      PUT (X, Y), Array, XOR
   NEXT X

   StepSize = -StepSize
   StartLoop = -StartLoop
LOOP UNTIL Esc$ <> "" OR Decay < .01

END
.
.
.
```

Output



5.10.4 Animating with Screen Pages

This section describes an animation technique that utilizes multiple pages of your computer's video memory.

Pages in video memory are analogous to pages in a book. Depending on the graphics capability of your computer, what you see displayed on the screen may only be part of the video memory available—just as what you see when you open a book is only part of the book. However, unlike a book, the unseen pages of your computer's video memory can be active; that is, while you are looking at one page on the screen, graphics output can be taking place on the others. It's as if the author of a book were still writing new pages even as you were reading the book.

The area of video memory visible on the screen is called the "visual page," while the area of video memory where graphics statements put their output is called the "active page." The **SCREEN** statement allows you to select visual and active screen pages with the following syntax:

SCREEN [[*mode*]], [[, [[*apage*]][[, *vpage*]]]]

In this syntax, *apage* is the number of the active page, and *vpage* is the number of the visual page. The active page and the visual page can be one and the same (and are by default when the *apage* or *vpage* arguments are not used with **SCREEN**, in which case the value of both arguments is 0).

You can animate objects on the screen by selecting a screen mode with more than one video memory page, then alternating the pages, sending output to one or more active pages while displaying finished output on the visual page. This technique makes an active page visible only when output to that page is complete. Since the viewer sees only a finished image, the display is instantaneous.

Example

The following program demonstrates the technique discussed above. It selects screen mode 7, which has two pages, then draws a cube with the **DRAW** statement. This cube is then rotated through successive 15° angles by changing the value of the **TA** macro in the string used by **DRAW**. By swapping the active and visual pages back and forth, this program always shows a completed cube while a new one is being drawn.

This program is in the file named CUBE.BAS on the QuickBASIC distribution disks.

```
' Define the macro string used to draw the cube
' and paint its sides:
One$ =  "BR30 BU25 C1 R54 U45 L54 D45 BE20 P1,1G20 C2 G20"
Two$ =  "R54 E20 L54 BD5 P2,2 U5 C4 G20 U45 E20 D45 BL5 P4,4"
Plot$ = One$ + Two$

APage% = 1      ' Initialize values for the active and visual
VPage% = 0      ' pages as well as the angle of rotation.
Angle% = 0

DO
   SCREEN 7, , APage%, VPage% ' Draw to the active page
                              ' while showing the visual page.

   CLS 1                      ' Clear the active page.

   ' Rotate the cube "Angle%" degrees:
   DRAW "TA" + STR$(Angle%) + Plot$

   ' Angle% is some multiple of 15 degrees:
   Angle% = (Angle% + 15) MOD 360

   ' Drawing is complete, so make the cube visible in its
   ' new position by switching the active and visual pages:
   SWAP APage%, VPage%

LOOP WHILE INKEY$ = ""         ' A keystroke ends the program.

END
```

5.11 Sample Applications

The sample applications in this chapter are a bar-graph generator, a program that plots points in the Mandelbrot Set using different colors, and a pattern editor.

5.11.1 Bar-Graph Generator (BAR.BAS)

This program uses all the forms of the **LINE** statement presented above in Sections 5.3.2.1–5.3.2.3 to draw a filled bar chart. Each bar is filled with a pattern specified in a **PAINT** statement. The input for the program consists of titles for the graph, labels for the *x*- and *y*-axes, and a set of up to five labels (with associated values) for the bars.

Statements and Functions Used

This program demonstrates the use of the following graphics statements:

- **LINE**
- **PAINT** (with a pattern)
- **SCREEN**

Program Listing

The bar-graph generator program BAR.BAS is listed below.

```
' Define type for the titles:
TYPE TitleType
   MainTitle AS STRING * 40
   XTitle AS STRING * 40
   YTitle AS STRING * 18
END TYPE

DECLARE SUB InputTitles (T AS TitleType)
DECLARE FUNCTION DrawGraph$ (T AS TitleType, Label$(), Value!(), N%)
DECLARE FUNCTION InputData% (Label$(), Value!())

' Variable declarations for titles and bar data:
DIM Titles AS TitleType, Label$(1 TO 5), Value(1 TO 5)

CONST FALSE = 0, TRUE = NOT FALSE

DO
   InputTitles Titles
   N% = InputData%(Label$(), Value())
   IF N% <> FALSE THEN
      NewGraph$ = DrawGraph$(Titles, Label$(), Value(), N%)
   END IF
LOOP WHILE NewGraph$ = "Y"

END
```

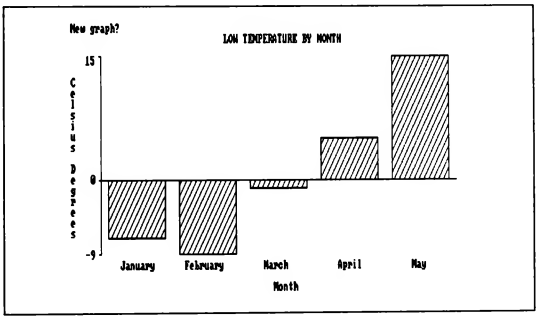
```
' ======================== DRAWGRAPH ======================
'   Draws a bar graph from the data entered in the
'   INPUTTITLES and INPUTDATA procedures.
' =========================================================

FUNCTION DrawGraph$ (T AS TitleType, Label$(), Value(), N%) STATIC

   ' Set size of graph:
   CONST GRAPHTOP = 24, GRAPHBOTTOM = 171
   CONST GRAPHLEFT = 48, GRAPHRIGHT = 624
   CONST YLENGTH = GRAPHBOTTOM - GRAPHTOP

   ' Calculate maximum and minimum values:
   YMax = 0
   YMin = 0
   FOR I% = 1 TO N%
      IF Value(I%) < YMin THEN YMin = Value(I%)
      IF Value(I%) > YMax THEN YMax = Value(I%)
   NEXT I%

   ' Calculate width of bars and space between them:
   BarWidth = (GRAPHRIGHT - GRAPHLEFT) / N%
   BarSpace = .2 * BarWidth
   BarWidth = BarWidth - BarSpace

   SCREEN 2
   CLS

   ' Draw y-axis:
   LINE (GRAPHLEFT, GRAPHTOP)-(GRAPHLEFT, GRAPHBOTTOM), 1

   ' Draw main graph title:
   Start% = 44 - (LEN(RTRIM$(T.MainTitle)) / 2)
   LOCATE 2, Start%
   PRINT RTRIM$(T.MainTitle);

   ' Annotate y-axis:
   Start% = CINT(13 - LEN(RTRIM$(T.YTitle)) / 2)
   FOR I% = 1 TO LEN(RTRIM$(T.YTitle))
      LOCATE Start% + I% - 1, 1
      PRINT MID$(T.YTitle, I%, 1);
   NEXT I%

   ' Calculate scale factor so labels aren't bigger than four digits:
   IF ABS(YMax) > ABS(YMin) THEN
      Power = YMax
   ELSE
      Power = YMin
   END IF
   Power = CINT(LOG(ABS(Power) / 100) / LOG(10))
   IF Power < 0 THEN Power = 0

   ' Scale minimum and maximum values down:
   ScaleFactor = 10 ^ Power
   YMax = CINT(YMax / ScaleFactor)
   YMin = CINT(YMin / ScaleFactor)
```

```
' If power isn't zero then put scale factor on chart:
IF Power <> 0 THEN
   LOCATE 3, 2
   PRINT "x 10^"; LTRIM$(STR$(Power))
END IF

' Put tic mark and number for Max point on y-axis:
LINE (GRAPHLEFT - 3, GRAPHTOP) -STEP(3, 0)
LOCATE 4, 2
PRINT USING "####"; YMax

' Put tic mark and number for Min point on y-axis:
LINE (GRAPHLEFT - 3, GRAPHBOTTOM) -STEP(3, 0)
LOCATE 22, 2
PRINT USING "####"; YMin

YMax = YMax * ScaleFactor ' Scale minimum and maximum back
YMin = YMin * ScaleFactor ' up for charting calculations.

' Annotate x-axis:
Start% = 44 - (LEN(RTRIM$(T.XTitle)) / 2)
LOCATE 25, Start%
PRINT RTRIM$(T.XTitle);

' Calculate the pixel range for the y-axis:
YRange = YMax - YMin

' Define a diagonally striped pattern:
Tile$ = CHR$(1)+CHR$(2)+CHR$(4)+CHR$(8)+CHR$(16)+CHR$(32)+CHR$(64)+CHR$(128)

' Draw a zero line if appropriate:
IF YMin < 0 THEN
   Bottom = GRAPHBOTTOM - ((-YMin) / YRange * YLENGTH)
   LOCATE INT((Bottom - 1) / 8) + 1, 5
   PRINT "0";
ELSE
   Bottom = GRAPHBOTTOM
END IF

' Draw x-axis:
LINE (GRAPHLEFT - 3, Bottom)-(GRAPHRIGHT, Bottom)
' Draw bars and labels:
Start% = GRAPHLEFT + (BarSpace / 2)
FOR I% = 1 TO N%

   ' Draw a bar label:
   BarMid = Start% + (BarWidth / 2)
   CharMid = INT((BarMid - 1) / 8) + 1
   LOCATE 23, CharMid - INT(LEN(RTRIM$(Label$(I%))) / 2)
   PRINT Label$(I%);

   ' Draw the bar and fill it with the striped pattern:
   BarHeight = (Value(I%) / YRange) * YLENGTH
   LINE (Start%, Bottom) -STEP(BarWidth, -BarHeight), , B
   PAINT (BarMid, Bottom - (BarHeight / 2)), Tile$, 1

   Start% = Start% + BarWidth + BarSpace
NEXT I%
```

```
    LOCATE 1, 1
    PRINT "New graph? ";
    DrawGraph$ = UCASE$(INPUT$(1))

END FUNCTION

' ======================== INPUTDATA =======================
'     Gets input for the bar labels and their values
' ==========================================================

FUNCTION InputData% (Label$(), Value()) STATIC

    ' Initialize the number of data values:
    NumData% = 0

    ' Print data-entry instructions:
    CLS
    PRINT "Enter data for up to 5 bars:"
    PRINT "   * Enter the label and value for each bar."
    PRINT "   * Values can be negative."
    PRINT "   * Enter a blank label to stop."
    PRINT
    PRINT "After viewing the graph, press any key ";
    PRINT "to end the program."

    ' Accept data until blank label or 5 entries:
    Done% = FALSE
    DO
       NumData% = NumData% + 1
       PRINT
       PRINT "Bar("; LTRIM$(STR$(NumData%)); "):"
       INPUT ; "        Label? ", Label$(NumData%)

       ' Only input value if label isn't blank:
       IF Label$(NumData%) <> "" THEN
          LOCATE , 35
          INPUT "Value? ", Value(NumData%)

       ' If label is blank, decrement data counter
       ' and set Done flag equal to TRUE:
       ELSE
          NumData% = NumData% - 1
          Done% = TRUE
       END IF
    LOOP UNTIL (NumData% = 5) OR Done%

    ' Return the number of data values input:
    InputData% = NumData%

END FUNCTION
```

```
' ======================= INPUTTITLES ======================
'     Accepts input for the three different graph titles
' ==========================================================

SUB InputTitles (T AS TitleType) STATIC
   SCREEN 0, 0             ' Set text screen.
   DO                      ' Input titles.
      CLS
      INPUT "Enter main graph title: ", T.MainTitle
      INPUT "Enter x-axis title    : ", T.XTitle
      INPUT "Enter y-axis title    : ", T.YTitle

      ' Check to see if titles are OK:
      LOCATE 7, 1
      PRINT "OK (Y to continue, N to change)? ";
      LOCATE , , 1
      OK$ = UCASE$(INPUT$(1))
   LOOP UNTIL OK$ = "Y"
END SUB
```

Output



5.11.2 Color in a Figure Generated Mathematically (MANDEL.BAS)

This program uses BASIC graphics statements to generate a figure known as a "fractal." A fractal is a graphic representation of what happens to numbers when they are subjected to a repeated sequence of mathematical operations. The fractal generated by this program shows a subset of the class of numbers known as complex numbers; this subset is called the "Mandelbrot Set," named after Benoit B. Mandelbrot of the IBM Thomas J. Watson Research Center.

Briefly, complex numbers have two parts, a real part and a so-called imaginary part, some multiple of the $\sqrt{-1}$. Squaring a complex number, then plugging the real and imaginary parts back into a second complex number, squaring the new complex number, and repeating the process causes some complex numbers to get very large fairly fast. However, others hover around some stable value. The stable values are in the Mandelbrot Set and are represented in this program by the color black. The unstable values—that is, the ones that are moving away from the Mandelbrot Set—are represented by the other colors in the palette. The smaller the color attribute, the more unstable the point.

See A.K. Dewdney's column, "Computer Recreations," in *Scientific American*, August 1985, for more background on the Mandelbrot Set.

This program also tests for the presence of an EGA card, and if one is present, it draws the Mandelbrot Set in screen mode 8. After drawing each line, the program rotates the 16 colors in the palette with a **PALETTE USING** statement. If there is no EGA card, the program draws a four-color (white, magenta, cyan, and black) Mandelbrot Set in screen mode 1.

Statements and Functions Used

This program demonstrates the use of the following graphics statements:

- **LINE**
- **PALETTE USING**
- **PMAP**
- **PSET**
- **SCREEN**
- **VIEW**
- **WINDOW**

Program Listing

```
DEFINT A-Z          ' Default variable type is integer.

DECLARE SUB ShiftPalette ()
DECLARE SUB WindowVals (WL%, WR%, WT%, WB%)
DECLARE SUB ScreenTest (EM%, CR%, VL%, VR%, VT%, VB%)

CONST FALSE = 0, TRUE = NOT FALSE ' Boolean constants

' Set maximum number of iterations per point:
CONST MAXLOOP = 30, MAXSIZE = 1000000

DIM PaletteArray(15)
FOR I = 0 TO 15: PaletteArray(I) = I: NEXT I

' Call WindowVals to get coordinates of window corners:
WindowVals WLeft, WRight, WTop, WBottom

' Call ScreenTest to find out if this is an EGA machine
' and get coordinates of viewport corners:
ScreenTest EgaMode, ColorRange, VLeft, VRight, VTop, VBottom

' Define viewport and corresponding window:
VIEW (VLeft, VTop)-(VRight, VBottom), 0, ColorRange
WINDOW (WLeft, WTop)-(WRight, WBottom)

LOCATE 24, 10 : PRINT "Press any key to quit.";

XLength = VRight - VLeft
YLength = VBottom - VTop
ColorWidth = MAXLOOP \ ColorRange

' Loop through each pixel in viewport and calculate
' whether or not it is in the Mandelbrot Set:
FOR Y = 0 TO YLength          ' Loop through every line
                              ' in the viewport.
   LogicY = PMAP(Y, 3)        ' Get the pixel's view
                              ' y-coordinate.
   PSET (WLeft, LogicY)       ' Plot leftmost pixel in the line.
   OldColor = 0               ' Start with background color.

   FOR X = 0 TO XLength       ' Loop through every pixel
                              ' in the line.
      LogicX = PMAP(X, 2)     ' Get the pixel's view
                              ' x-coordinate.
      MandelX& = LogicX
      MandelY& = LogicY
```

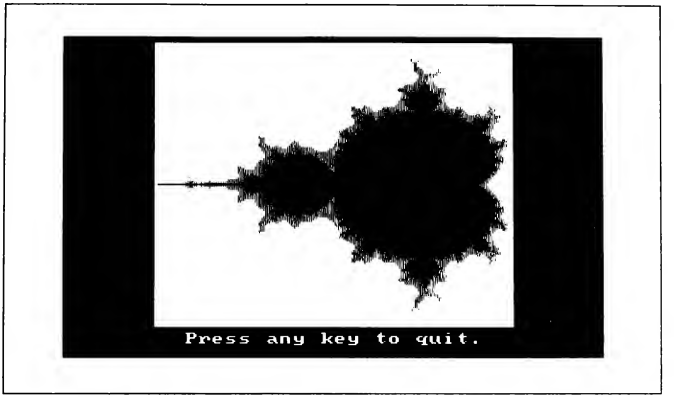
```
        ' Do the calculations to see if this point
        ' is in the Mandelbrot Set:
        FOR I = 1 TO MAXLOOP
           RealNum& = MandelX& * MandelX&
           ImagNum& = MandelY& * MandelY&
           IF (RealNum& + ImagNum&) >= MAXSIZE THEN EXIT FOR
           MandelY& = (MandelX& * MandelY&) \ 250 + LogicY
           MandelX& = (RealNum& - ImagNum&) \ 500 + LogicX
        NEXT I

        ' Assign a color to the point:
        PColor = I \ ColorWidth

        ' If color has changed, draw a line from
        ' the last point referenced to the new point,
        ' using the old color:
        IF PColor <> OldColor THEN
           LINE -(LogicX, LogicY), (ColorRange - OldColor)
           OldColor = PColor
        END IF

        IF INKEY$ <> "" THEN END
     NEXT X

     ' Draw the last line segment to the right edge
     ' of the viewport:
     LINE -(LogicX, LogicY), (ColorRange - OldColor)

     ' If this is an EGA machine, shift the palette after
     ' drawing each line:
     IF EgaMode THEN ShiftPalette
NEXT Y

DO
     ' Continue shifting the palette
     ' until the user presses a key:
     IF EgaMode THEN ShiftPalette
LOOP WHILE INKEY$ = ""

SCREEN 0, 0              ' Restore the screen to text mode,
WIDTH 80                 ' 80 columns.
END

BadScreen:               ' Error handler that is invoked if
     EgaMode = FALSE     ' there is no EGA graphics card
     RESUME NEXT
```

```
' ======================= ShiftPalette ====================
'    Rotates the palette by one each time it is called
' =========================================================

SUB ShiftPalette STATIC
   SHARED PaletteArray(), ColorRange

   FOR I = 1 TO ColorRange
      PaletteArray(I) = (PaletteArray(I) MOD ColorRange) + 1
   NEXT I
   PALETTE USING PaletteArray(0)

END SUB

' ======================= ScreenTest ======================
'    Uses a SCREEN 8 statement as a test to see if user has
'    EGA hardware. If this causes an error, the EM flag is
'    set to FALSE, and the screen is set with SCREEN 1.

'    Also sets values for corners of viewport (VL = left,
'    VR = right, VT = top, VB = bottom), scaled with the
'    correct aspect ratio so viewport is a perfect square.
' =========================================================

SUB ScreenTest (EM, CR, VL, VR, VT, VB) STATIC
   EM = TRUE
   ON ERROR GOTO BadScreen
   SCREEN 8, 1
   ON ERROR GOTO 0

   IF EM THEN                    ' No error, SCREEN 8 is OK.
      VL = 110: VR = 529
      VT = 5: VB = 179
      CR = 15                    ' 16 colors (0 - 15)

   ELSE                          ' Error, so use SCREEN 1.
      SCREEN 1, 1
      VL = 55: VR = 264
      VT = 5: VB = 179
      CR = 3                     ' 4 colors (0 - 3)
   END IF

END SUB
```

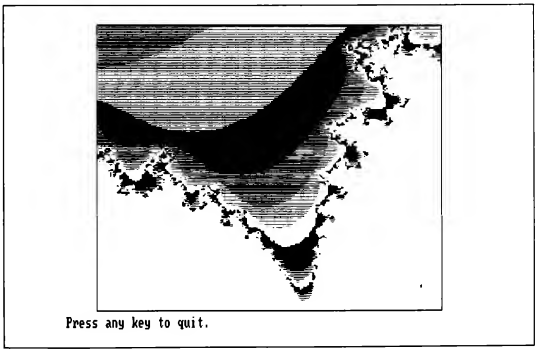
```
' ======================= WindowVals =====================
'     Gets window corners as input from the user, or sets
'     values for the corners if there is no input
' ========================================================

SUB WindowVals (WL, WR, WT, WB) STATIC
   CLS
   PRINT "This program prints the graphic representation of"
   PRINT "the complete Mandelbrot Set. The default window"
   PRINT "is from (-1000,625) to (250,-625). To zoom in on"
   PRINT "part of the figure, input coordinates inside"
   PRINT "this window."
   PRINT "Press <ENTER> to see the default window or"
   PRINT "any other key to input window coordinates: ";
   LOCATE , , 1
   Resp$ = INPUT$(1)

   ' User didn't press ENTER, so input window corners:
   IF Resp$ <> CHR$(13) THEN
      PRINT
      INPUT "x-coordinate of upper-left corner: ", WL
      DO
         INPUT "x-coordinate of lower-right corner: ", WR
         IF WR <= WL THEN
         PRINT "Right corner must be greater than left corner."
         END IF
      LOOP WHILE WR <= WL
      INPUT "y-coordinate of upper-left corner: ", WT
      DO
         INPUT "y-coordinate of lower-right corner: ", WB
         IF WB >= WT THEN
         PRINT "Bottom corner must be less than top corner."
         END IF
      LOOP WHILE WB >= WT

   ' User pressed ENTER, so set default values:
   ELSE
      WL = -1000
      WR = 250
      WT = 625
      WB = -625
   END IF
END SUB
```

Output

The following figure shows the Mandelbrot Set in screen mode 1. This is the output you see if you have a CGA and you choose the default window coordinates.



The next figure shows the Mandelbrot Set with (x, y) coordinates of (–500, 250) for the upper-left corner and (–300, 50) for the lower-right corner. This figure is drawn in screen mode 8, the default for an EGA or VGA.



5.11.3 Pattern Editor (EDPAT.BAS)

This program allows you to edit a pattern tile for use with **PAINT**. While you are editing the tile on the left side of the screen, you can check the appearance of the finished pattern on the right side of the screen. When you have finished editing the pattern tile, the program prints the integer arguments used by the **CHR$** statement to draw each row of the tile.

Statements and Functions Used

This program demonstrates the use of the following graphics statements:

- **LINE**
- **PAINT** (with pattern)
- **VIEW**

Program Listing

```
DECLARE SUB DrawPattern ()
DECLARE SUB EditPattern ()
DECLARE SUB Initialize ()
DECLARE SUB ShowPattern (OK$)

DIM Bit%(0 TO 7), Pattern$, Esc$, PatternSize%

DO
   Initialize
   EditPattern
   ShowPattern OK$
LOOP WHILE OK$ = "Y"

END

' ======================= DRAWPATTERN ====================
'  Draws a patterned rectangle on the right side of screen
' ========================================================

SUB DrawPattern STATIC
SHARED Pattern$

   VIEW (320, 24)-(622, 160), 0, 1  ' Set view to rectangle.
   PAINT (1, 1), Pattern$           ' Use PAINT to fill it.
   VIEW                             ' Set view to full screen.

END SUB
```

```
' ======================== EDITPATTERN =====================
'                  Edits a tile-byte pattern
' ==========================================================

SUB EditPattern STATIC
SHARED Pattern$, Esc$, Bit%(), PatternSize%

   ByteNum% = 1          ' Starting position.
   BitNum% = 7
   Null$ = CHR$(0)       ' CHR$(0) is the first byte of the
                         ' two-byte string returned when a
                         ' direction key such as UP or DOWN is
                         ' pressed.
   DO

      ' Calculate starting location on screen of this bit:
      X% = ((7 - BitNum%) * 16) + 80
      Y% = (ByteNum% + 2) * 8

      ' Wait for a key press (flash cursor each 3/10 second):
      State% = 0
      RefTime = 0
      DO

         ' Check timer and switch cursor state if 3/10 second:
         IF ABS(TIMER - RefTime) > .3 THEN
            RefTime = TIMER
            State% = 1 - State%

            ' Turn the  border of bit on and off:
            LINE (X%-1, Y%-1) -STEP(15, 8), State%, B
         END IF

         Check$ = INKEY$         ' Check for keystroke.

      LOOP WHILE Check$ = ""     ' Loop until a key is pressed.

      ' Erase cursor:
      LINE (X%-1, Y%-1) -STEP(15, 8), 0, B

      SELECT CASE Check$         ' Respond to keystroke.

      CASE CHR$(27)              ' ESC key pressed:
         EXIT SUB                ' exit this subprogram.
```

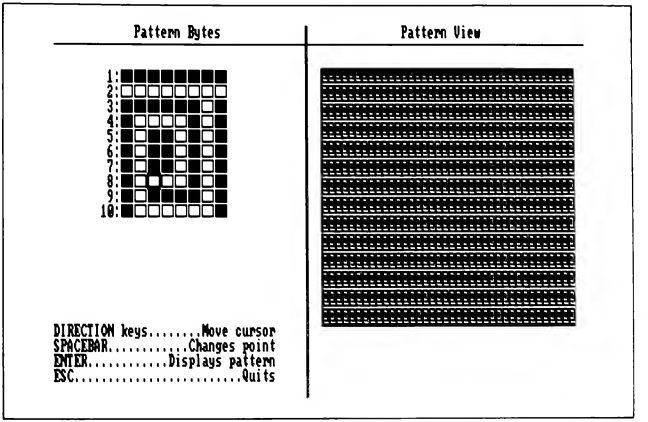
```
CASE CHR$(32)              ' SPACEBAR pressed:
                                ' reset state of bit.

        ' Invert bit in pattern string:
        CurrentByte% = ASC(MID$(Pattern$, ByteNum%, 1))
        CurrentByte% = CurrentByte% XOR Bit%(BitNum%)
        MID$ (Pattern$, ByteNum%) = CHR$(CurrentByte%)

        ' Redraw bit on screen:
        IF (CurrentByte% AND Bit%(BitNum%)) <> 0 THEN
            CurrentColor% = 1
        ELSE
            CurrentColor% = 0
        END IF
        LINE (X%+1, Y%+1) -STEP(11, 4), CurrentColor%, BF

     CASE CHR$(13)              ' ENTER key pressed: draw
        DrawPattern             ' pattern in box on right.

     CASE Null$ + CHR$(75)      ' LEFT key: move cursor left

        BitNum% = BitNum% + 1
        IF     BitNum% > 7 THEN BitNum% = 0

     CASE Null$ + CHR$(77)      ' RIGHT key: move cursor rig

        BitNum% = BitNum% - 1
        IF     BitNum% < 0 THEN BitNum% = 7

     CASE Null$ + CHR$(72)      ' UP key: movecursor up.

        ByteNum% =     ByteNum% - 1
        IF     ByteNum% < 1 THEN ByteNum% = PatternSize%

     CASE Null$ + CHR$(80)      ' DOWNkey: move cursor down.

        ByteNum% =     ByteNum% + 1
        IF     ByteNum% > PatternSize%THEN ByteNum% =1

     CASE ELSE
        ' User pressed a key other than ESC, SPACEBAR,
        ' ENTER, UP, DOWN, LEFT, or RIGHT, so don't
        ' do anything.
     END SELECT
  LOOP
END SUB
```

```
' ======================= INITIALIZE ======================
'             Sets up starting pattern and screen
' =========================================================

SUB Initialize STATIC
SHARED Pattern$, Esc$, Bit%(), PatternSize%

   Esc$ = CHR$(27)              ' ESC character is ASCII 27.

   ' Set up an array holding bits in positions 0 to 7:
   FOR I% = 0 TO 7
      Bit%(I%) = 2 ^ I%
   NEXT I%

   CLS

   ' Input the pattern size (in number of bytes):
   LOCATE 5, 5
   PRINT "Enter pattern size (1-16 rows):";
   DO
      LOCATE 5, 38
      PRINT "    ";
      LOCATE 5, 38
      INPUT "", PatternSize%
   LOOP WHILE PatternSize% < 1 OR PatternSize% > 16

   ' Set initial pattern to all bits set:
   Pattern$ = STRING$(PatternSize%, 255)

   SCREEN 2              ' 640 x 200 monochrome graphics mode

   ' Draw dividing lines:
   LINE (0, 10)-(635, 10), 1
   LINE (300, 0)-(300, 199)
   LINE (302, 0)-(302, 199)

   ' Print titles:
   LOCATE 1, 13: PRINT "Pattern Bytes"
   LOCATE 1, 53: PRINT "Pattern View"
```

```
' Draw editing screen for pattern:
   FOR I% = 1 TO PatternSize%

      ' Print label on left of each line:
      LOCATE I% + 3, 8
      PRINT USING "##:"; I%

      ' Draw "bit" boxes:
      X% = 80
      Y% = (I% + 2) * 8
      FOR J% = 1 TO 8
         LINE (X%, Y%) -STEP(13, 6), 1,BF
         X% = X% + 16
      NEXT J%
   NEXT I%

   DrawPattern          ' Draw  "Pattern View" box.

   LOCATE 21, 1
   PRINT "DIRECTION keys........Move cursor"
   PRINT "SPACEBAR............Changes point"
   PRINT "ENTER............Displays pattern"
   PRINT "ESC.........................Quits";

END SUB

' ======================== SHOWPATTERN ====================
'   Prints the CHR$ values used by PAINT to make pattern
' =========================================================

SUB ShowPattern (OK$) STATIC
SHARED Pattern$, PatternSize%

   ' Return screen to 80-column text mode:
   SCREEN 0, 0
   WIDTH 80

   PRINT "The following characters make up your pattern:"
   PRINT

   ' Print out the value for each pattern byte:
   FOR I% = 1 TO PatternSize%
      PatternByte% = ASC(MID$(Pattern$,I%, 1))
      PRINT "CHR$("; LTRIM$(STR$(PatternByte%)); ")"
   NEXT I%
   PRINT
   LOCATE , , 1
   PRINT "New pattern? ";
   OK$ = UCASE$(INPUT$(1))
END SUB
```

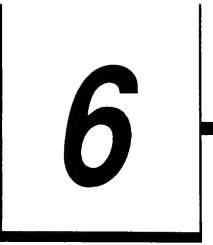
Output

This is a sample pattern generated by the pattern editor.



CHAPTER 6

Error and Event Trapping



This chapter shows how to trap errors and events that occur while a program is running. With error trapping, you can protect your program from such user errors as entering a string when a numeric value is expected or trying to open a file in a nonexistent directory. With event trapping, your program can detect and respond to real-time events such as keystrokes or data arriving at a COM port.

When you have completed this chapter, you will know how to perform these tasks:

- Activate error trapping or event trapping
- Write a routine to process trapped errors or events
- Return control from an error-handling routine or an event-handling routine
- Write a program that traps any keystroke or combination of keystrokes
- Trap errors or events within **SUB** or **FUNCTION** procedures
- Trap errors or events in programs composed of more than one module

6.1 Error Trapping

Error trapping lets your program intercept run-time errors before they force the program to halt. Without error trapping, errors during program execution (such as trying to open a nonexistent data file) cause BASIC to display the appropriate error message, then stop the program. If someone is running the stand-alone version of your program, they will have to restart the program, losing data entered or calculations performed before the error occurred.

When error trapping is active, the trapped error makes the program branch to a user-written "error-handling routine," which corrects the error. If you can anticipate errors that might occur when someone else uses your program, error trapping will let you handle those errors in a "user-friendly" way.

Sections 6.1.1 and 6.1.2 below show how to activate error trapping, how to write a routine to handle errors when they are trapped, and how to return control from the error-handling routine after it has dealt with the error.

6.1.1 Activating Error Trapping

You activate error trapping with the statement **ON ERROR GOTO** *line*, where *line* is a line number or line label identifying the first line of an error-handling routine. Once BASIC has encountered an **ON ERROR GOTO** *line* statement, any run-time error within the module containing that statement causes a branch to the specified *line*. (If the number or label does not exist within the module, a run-time error message reading `Label not defined` appears.)

An **ON ERROR GOTO** statement must be executed before error trapping takes effect. Therefore, you must position the **ON ERROR GOTO** statement so that program execution reaches it before it is needed. This statement would usually be one of the first executable statements in the main module or a procedure.

You cannot use an **ON ERROR GOTO** 0 statement to branch to an error handler, because this statement has a special meaning in error trapping. The **ON ERROR GOTO** 0 statement has two effects, depending on where it appears. If it appears outside an error-handling routine, it turns off error trapping. If it appears inside an error-handling routine (as it does in the `Handler` routine in the next example), it causes BASIC to print its standard message for the given error and stops the program.

Therefore, even if your program has a line with line-number 0, an **ON ERROR GOTO** 0 statement tells BASIC either to turn off error detection or to terminate execution.

6.1.2 Writing an Error-Handling Routine

An error-handling routine consists of the following three parts:

1. The line label or line number that is specified in the statement **ON ERROR GOTO** *line*, which is the first statement the program branches to after an error
2. The body of the routine, which determines the error that caused the branch and takes appropriate action for each anticipated error
3. At least one **RESUME** or **RESUME NEXT** statement to return control from the routine

An error handler must be placed where it cannot be executed during the normal flow of program execution. For example, an error-handling routine in the program's main module might be placed after an **END** statement. Otherwise, a **GOTO** statement might be needed to skip over it during normal execution.

6.1.2.1 Using ERR to Identify Errors

Once the program has branched to an error-handling routine, it must determine which error caused the branch. To identify the culprit, use the **ERR** function. This function returns a numeric code for the program's most recent run-time error. (See Table I.1, "Run-Time Error Codes," for a complete list of the error codes associated with run-time errors.)

NOTE *Errors cannot be trapped within an error-handling routine. If an error occurs while the error-handling routine is processing another error, the program displays the message for the new error and then terminates. Event handling is also suspended, but resumes when QuickBASIC returns from the error handler. The first event that occurred after event handling was suspended is saved.*

Example

The following program includes the error-handling routine `Handler`, designed to deal specifically with three different error situations. The **ERR** function, used in a **SELECT CASE** statement, allows this routine to take actions appropriate for each error.

```
DATA BASIC, Pascal, FORTRAN, C, Modula, Forth
DATA LISP, Ada, COBOL

CONST FALSE = 0, TRUE = NOT FALSE

EndOfData = FALSE                ' Set end of data flag.

ON ERROR GOTO Handler            ' Activate error trapping.

OPEN "LPT1:" FOR OUTPUT AS #1    ' Open the printer for output

DO
   ' Continue reading items from the DATA statements above,
   ' then printing them on the line printer, until there is
   ' an "Out of DATA" error message signifying no more data:
   READ Buffer$
   IF NOT EndOfData THEN
      PRINT #1, Buffer$
   ELSE
      EXIT DO
   END IF
LOOP

CLOSE #1
END
```

```
Handler:                          ' Error-handling routine

   ' Use ERR to determine which error
   ' caused the branch to "Handler":
   SELECT CASE ERR
      CASE 4

         ' 4 is the code for "Out of DATA" in DATA
         ' statements:
         EndOfData = TRUE
         RESUME NEXT
      CASE 25

         ' 25 is the code for "Device fault" error,
         ' which can be caused by trying to output
         ' to the printer when it's not on:
         PRINT "Turn printer on, then";
         PRINT "press any key to continue"
         Pause$ = INPUT$(1)
         RESUME
      CASE 27

         ' 27 is the code for "Out of paper":
         PRINT "Printer is out of paper. Insert"
         PRINT "paper, then press any key to continue."
         Pause$ = INPUT$(1)

         ' Start reading data from the beginning of first
         ' DATA statement after the paper is inserted:
         RESTORE
         RESUME
      CASE ELSE

         ' An unanticipated error has occurred; display the
         ' message for that error and stop the program:
         ON ERROR GOTO 0
   END SELECT
```

6.1.2.2 Returning from an Error-Handling Routine

The **RESUME** statement returns control from an error-handling routine. The preceding example used the following two variations of **RESUME** to return from `Handler`:

Statement	Action
RESUME	Causes the program to branch back to the exact statement that caused the error. If the program had to go to another module to find an active error handler, control returns to the last statement executed in that module. The preceding program used **RESUME** to return to the **PRINT** statement that attempted to send output to the printer, giving it another chance to print the value in `Buffer$` after the printer has (presumably) been turned on.
RESUME NEXT	Causes the program to branch back to the statement following the one that caused the error. If the program had to go to another module to find an active error handler, control returns to the statement following the last statement executed in that module. The preceding program used the statement **RESUME NEXT** when recovering from the `Out of DATA` error. In this case, a simple **RESUME** would have sent the program into an endless loop, since each time control returned to the **READ** statement in the main program, another `Out of DATA` error would result, invoking the error routine over and over again.

Figure 6.1 outlines the flow of a program's control during error handling with **RESUME**.

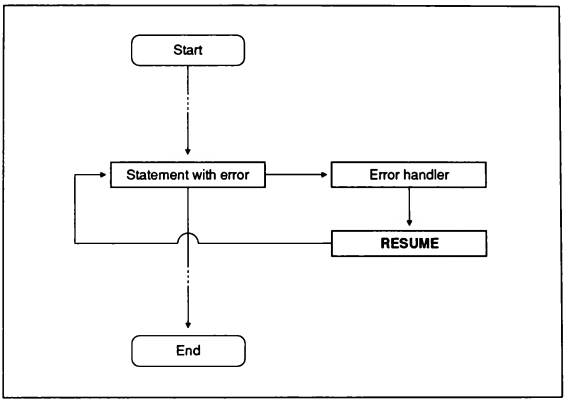


Figure 6.1 Program Flow of Control with RESUME

Contrast the preceding figure with Figure 6.2, which shows what happens in a program using **RESUME NEXT**.

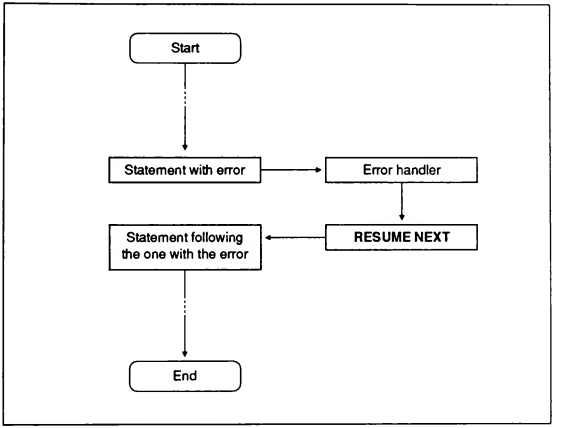


Figure 6.2 Program Flow of Control with RESUME NEXT

Another variant of **RESUME** is **RESUME** *line*, where *line* is a line number or line label outside any **SUB...END SUB**, **FUNCTION...END FUNCTION**, or **DEF FN...END DEF** block. Because *line* must be outside these blocks, a **RESUME** *line* statement can produce unexpected effects if it appears in an error-handling routine inside a **SUB** or **FUNCTION** procedure or a **DEF FN** function. It should generally be avoided in favor of **RESUME** or **RESUME NEXT**.

6.2 Event Trapping

Section 6.2.1 below compares two methods your BASIC programs can use to detect events: polling and trapping. Sections 6.2.2–6.2.4 then discuss the different events BASIC can trap, how to set and activate a trap, and how to suspend or deactivate the trap. Sections 6.2.5–6.2.7 provide more detailed information on trapping keystrokes and music events.

6.2.1 Detecting Events by Polling

One way to detect an event and reroute program control to the appropriate routine is to use "event polling." In polling, your program stops whatever it is currently doing and explicitly checks for an event. For example, the following loop keeps checking the keyboard until the user presses either the Q or N keys:

```
DO
   Test$ = UCASE$(INKEY$)
LOOP UNTIL Test$ = "N" OR Test$ = "Q"
```

Polling works well when you know ahead of time exactly where in the flow of the program you want to check for an event.

6.2.2 Detecting Events by Trapping

But suppose you don't want your program to pause, or you would like it to check between every statement (that is, check continuously). In these cases, polling is unwieldy or impossible, and trapping becomes the better alternative.

Example

The following example shows how to use event trapping:

```
' Alert the program that it should branch to "KeySub"
' whenever the user presses the F1 function key:
ON KEY(1) GOSUB KeySub

' Turn on trapping for the F1 key. BASIC now checks in the
' background for this event until the end of the program,
' or until trapping is disabled with KEY(1) OFF
' or suspended with KEY(1) STOP:
KEY(1) ON
OPEN "Data" FOR INPUT AS #1
OPEN "Data.Out" FOR OUTPUT AS #2

DO UNTIL EOF(1)
   LINE INPUT #1, LineBuffer$
   PRINT #2, LineBuffer$
LOOP

KeySub:          ' When the user presses F1, the program
   .             ' branches to this procedure.
   .
   .
RETURN
```

6.2.3 Specifying the Event to Trap and Activating Event Trapping

As you can see from the preceding example, three steps are necessary for event trapping:

1. Define a target subroutine that is known as an "event-handling subroutine."
2. Tell the program to branch to the event-handling subroutine when the event takes place with an **ON** *event* **GOSUB** statement.
3. Activate trapping of that event with an **ON** *event* **GOSUB** statement.

Your program cannot trap a given *event* until it encounters both an *event* **ON** and an **ON** *event* **GOSUB** statement.

An event-handling routine is an ordinary BASIC **GOSUB** routine: control passes to the first line of the routine whenever the event is detected and returns to the main level of the module containing the routine with a **RETURN** statement.

Note the following important differences between the syntax of error trapping and the syntax of event trapping:

Error Trapping	Event Trapping
Activated with the statement **ON ERROR GOTO**, which also specifies the first line of the error-handling routine	Activated with an *event* **ON** statement; a separate statement, **ON** *event* **GOSUB**, specifies the first line of the event-handling subroutine
Returns control from the error routine with one or more **RESUME**, **RESUME NEXT**, or **RESUME** *line* statements	Returns control from the event subroutine with one or more **RETURN** statements

6.2.4 Events That BASIC Can Trap

You can trap the following events within a BASIC program:

BASIC Statement	Event Trapped
COM(*portnumber*)	Data from one of the serial ports (1 or 2) appearing in the communications buffer (the intermediate storage area for data sent to or received from the serial port)
KEY(*keynumber*)	The user pressing the given key
PEN	The user activating the light pen

PLAY(*queuelimit*)	The number of notes remaining to be played in the background dropping below *queuelimit*
STRIG(*triggernumber*)	The user squeezing the trigger of the joystick
TIMER(*interval*)	The passage of *interval* seconds

6.2.5 Suspending or Disabling Event Trapping

You can turn off detection and trapping of any event with the *event* **OFF** statement. Events occurring after an *event* **OFF** statement has been executed are ignored. If you want to suspend trapping a given event but continue detecting it, use the *event* **STOP** statement.

After *event* **STOP**, if the *event* occurs, there is no branch to the event-handling routine. However, your program remembers that the event occurred, and as soon as trapping is turned back on with *event* **ON**, branching to the event routine occurs immediately.

Event-handling routines execute an implicit *event* **STOP** statement for a given event whenever program control is in the routine; this is followed by an implicit *event* **ON** for that event when program control returns from the routine. For example, if a key-handling routine is processing a keystroke, trapping the same key is suspended until the previous keystroke is completely processed by the routine. If the user presses the same key during this time, this new keystroke is remembered and trapped after control returns from the key-handling routine.

Example

An event trap interrupts whatever is happening in the program at the moment the event occurs. Therefore, if you have a block of statements that you do not want interrupted, precede them with *event* **STOP** and follow them with *event* **ON**, as in this example:

```
' Once every minute (60 seconds),
' branch to the ShowTime subroutine:
ON TIMER(60) GOSUB ShowTime

' Activate trapping of the 60-second event:
TIMER ON
.
.
.
TIMER STOP      ' Suspend trapping.
.       ' A sequence of lines you don't want interrupted,
.       ' even if 60 or more seconds elapse
.
```

```
TIMER ON          ' Reactivate trapping.
.
.
.
END

ShowTime:

   ' Get the current row and column position of the cursor,
   ' and store them in the variables Row and Column:
   Row    = CSRLIN
   Column = POS(0)

   ' Go to the 24th row, 20th column, and print the time:
   LOCATE 24, 20
   PRINT TIME$

   ' Restore the cursor to its former position
   ' and return to the main program:
   LOCATE Row, Column
RETURN
```

6.2.6 Trapping Keystrokes

To detect a keystroke and route program control to a key-press routine, you need both of the following statements in your program:

ON KEY(*keynumber***) GOSUB** *line*
KEY(*keynumber***) ON**

Here, the *keynumber* value corresponds to the following keys:

Value	Key
1–10	Function keys F1-F10 (sometimes called "soft keys")
11	The UP direction key
12	The LEFT direction key
13	The RIGHT direction key
14	The DOWN direction key
15–25	User-defined keys (see Sections 6.2.6.1–6.2.6.2)
30	The F11 function key (101-key keyboard only)
31	The F12 function key (101-key keyboard only)

Examples

The following two lines cause the program to branch to the KeySub routine each time the F2 function key is pressed:

```
ON KEY(2) GOSUB KeySub
KEY(2) ON
.
.
.
```

The following four lines cause the program to branch to the DownKey routine when the DOWN direction key is pressed and to the UpKey routine when the UP direction key is pressed:

```
ON KEY(11) GOSUB UpKey
ON KEY(14) GOSUB DownKey
KEY(11) ON
KEY(14) ON
.
.
.
```

6.2.6.1 Trapping User-Defined Keys

In addition to providing the preassigned key numbers 1–14 (plus 30 and 31 with the 101-key keyboard), BASIC allows you to assign the numbers 15–25 to any of the remaining keys on the keyboard. To define your own key to trap, use these three statements:

KEY *keynumber*, **CHR$(0) + CHR$(***scancode***)**
ON KEY(*keynumber***) GOSUB** *line*
KEY(*keynumber***) ON**

Here, *keynumber* is a value from 15 to 25, and *scancode* is the scan code for that key. (See the first column of the "Keyboard Scan Codes" table in Appendix D for these codes.) For example, the following lines cause the program to branch to the TKey routine each time the user presses T:

```
' Define key 15 (the scan code for "t" is decimal 20):
KEY 15, CHR$(0) + CHR$(20)

' Define the trap (where to go when "t" is pressed):
ON KEY(15) GOSUB TKey

KEY(15) ON                  ' Turn on detection of key 15.

PRINT "Press q to end."
```

```
DO                              ' Idle loop: wait for user to
LOOP UNTIL INKEY$ = "q"         ' press "q".

END

TKey:                           ' Key-handling subroutine
   PRINT "Pressed t."
RETURN
```

6.2.6.2 Trapping User-Defined Shifted Keys

You can also set a trap for "shifted" keys. A key is said to be shifted when you press it simultaneously with one or more of the special keys SHIFT, CTRL, or ALT or if you press it after toggling on the keys NUM LOCK or CAPS LOCK.

This is how to trap the following key combinations:

KEY *keynumber*, **CHR$**(*keyboardflag*) + **CHR$**(*scancode*)
ON KEY(*keynumber*) **GOSUB** *line*
KEY(*keynumber*) **ON**

Here, *keynumber* is a value from 15 to 25; *scancode* is the scan code for the primary key; and *keyboardflag* is the sum of the individual codes for the special keys pressed, as shown in the following list:

Key	Code for *keyboardflag*
SHIFT	1, 2, or 3 Key trapping assumes the left and right shift keys are the same, so you can trap the SHIFT key with 1 (left), 2 (right), or 3 (left + right).
CTRL	4
ALT	8
NUM LOCK	32
CAPS LOCK	64
Any extended key on the 101-key keyboard (in other words, a key such as LEFT or DELETE that is not on the numeric keypad)	128

For example, the following statements turn on trapping of CTRL+S. Note these statements are designed to trap both the CTRL+S (lowercase) and CTRL+SHIFT+S (uppercase) key combinations. To trap the uppercase S, your program must recognize capital letters produced by holding down the SHIFT key, as well as those produced when the CAPS LOCK key is active, as shown here:

```
' 31 = scan code for S key
'  4 = code for CTRL key
KEY 15, CHR$(4) + CHR$(31)      ' Trap CTRL+S.

'  5 = code for CTRL key + code for SHIFT key
KEY 16, CHR$(5) + CHR$(31)      ' Trap CTRL+SHIFT+S.

' 68 = code for CTRL key + code for CAPSLOCK
KEY 17, CHR$(68) + CHR$(31)     ' Trap CTRL+CAPSLOCK+S.

ON KEY (15) GOSUB CtrlSTrap     ' Tell program where to
ON KEY (16) GOSUB CtrlSTrap     ' branch (note: same
ON KEY (17) GOSUB CtrlSTrap     ' subroutine for each key).

KEY (15) ON                     ' Activate key detection for
KEY (16) ON                     ' all three combinations.
KEY (17) ON
.
.
.
```

The following statements turn on trapping of CTRL+ALT+DEL:

```
' 12 = 4 + 8 = (code for CTRL key) + (code for ALT key)
' 83 = scan code for DEL key
KEY 20, CHR$(12) + CHR$(83)
ON KEY(20) GOSUB KeyHandler
KEY(20) ON
.
.
.
```

Note in the preceding example that the BASIC event trap overrides the normal effect of CTRL+ALT+DEL (system reset). Using this trap in your program is a handy way to prevent the user from accidentally rebooting while a program is running.

If you use a 101-key keyboard, you can trap any of the keys on the dedicated keypad by assigning the string

CHR$(128) + CHR$(*scancode*)

to any of the *keynumber* values from 15 to 25.

The next example shows how to trap the LEFT direction keys on both the dedicated cursor keypad and the numeric keypad.

```
' 128 = keyboard flag for keys on the
'       dedicated cursor keypad
'  75 = scan code for LEFT arrow key

KEY 15, CHR$(128) + CHR$(75)     ' Trap LEFT key on
ON KEY(15) GOSUB CursorPad       ' the dedicated
KEY(15) ON                       ' cursor keypad.

ON KEY(12) GOSUB NumericPad      ' Trap LEFT key on
KEY(12) ON                       ' the numeric keypad.

DO: LOOP UNTIL INKEY$ = "q"      ' Start idle loop.
END

CursorPad:
   PRINT "Pressed LEFT key on cursor keypad."
RETURN

NumericPad:
   PRINT "Pressed LEFT key on numeric keypad."
RETURN
```

IMPORTANT *For compatibility, QuickBASIC adopts many conventions of BASICA. One of these has to do with the way FUNCTION keys are trapped.*

If you initiate FUNCTION key trapping with an ***ON KEY (n) GOSUB*** *statement, both BASICA and QuickBASIC trap Fn regardless of whether a shift key (CTRL, ALT, SHIFT) is also pressed. This means that user-defined traps for shifted versions of that FUNCTION key are ignored.*

Therefore, if you wish to trap a FUNCTION key in both its shifted and unshifted states, you should create a user definition for each state. Keys that are only used unshifted can continue to use the ***ON KEY (n) GOSUB*** *statement.*

6.2.7 Trapping Music Events

When you use the **PLAY** statement to play music, you can choose whether the music plays in the foreground or in the background. If you choose foreground music (which is the default) nothing else can happen until the music finishes playing. However, if you use the **MB** (Music Background) option in a **PLAY** music string, the tune plays in the background while subsequent statements in your program continue executing.

The **PLAY** statement plays music in the background by feeding up to 32 notes at a time into a buffer, then playing the notes in the buffer while the program does other things. A "music trap" works by checking the number of notes currently left to be played in the buffer. As soon as this number drops below the limit you set in the trap, the program branches to the first line of the specified routine.

To set a music trap in your program, you need the following statements:

ON PLAY(*limit*) **GOSUB** *line*
PLAY ON
PLAY "MB"
.
.
.
PLAY *musicstring*
[[**PLAY** *musicstring*]]
.
.
.

Here, *limit* is a number between 1 and 32. For example, this fragment causes the program to branch to the `MusicTrap` subroutine whenever the number of notes remaining to be played in the music buffer goes from eight to seven:

```
ON PLAY(8) GOSUB MusicTrap
PLAY ON
.
.
.
PLAY "MB"        ' Play subsequent notes in the background.
PLAY "o1 A# B# C-"
.
.
.
MusicTrap:
   ' Music-trap subroutine
   .
   .
   .
RETURN
```

IMPORTANT *A music trap is triggered only when the number of notes goes from limit to limit–1. For example, if the music buffer in the preceding example never contained more than seven notes, the trap would never occur. In the example, the trap happens only when the number of notes drops from eight to seven.*

You can use a music-trap subroutine to play the same piece of music repeatedly while your program executes, as shown in the next example:

```
' Turn on trapping of background music events:
PLAY ON

' Branch to the Refresh subroutine when there are fewer than
' two notes in the background music buffer:
ON PLAY(2) GOSUB Refresh
```

```
PRINT "Press any key to start, q to end."
Pause$ = INPUT$(1)

' Select the background music option for PLAY:
PLAY "MB"

' Start playing the music, so notes will be put in the
' background music buffer:
GOSUB Refresh

I = 0

DO

   ' Print the numbers from 0 to 10,000 over and over until
   ' the user presses the "q" key. While this is happening,
   ' the music will repeat in the background:
   PRINT I
   I = (I + 1) MOD 10001
LOOP UNTIL INKEY$ = "q"

END

Refresh:

   ' Plays the opening motive of
   ' Beethoven's Fifth Symphony:
   Listen$ = "t180 o2 p2 p8 L8 GGG L2 E-"
   Fate$   = "p24 p8 L8 FFF L2 D"
   PLAY Listen$ + Fate$
RETURN
```

6.3 Error and Event Trapping in SUB or FUNCTION Procedures

The most important thing to remember when using error or event trapping with BASIC procedures is that either of the following statements can appear within a **SUB...END SUB** or **FUNCTION...END FUNCTION** block:

ON ERROR GOTO *line*
ON *event* **GOSUB** *line*

However, the *line* referred to in each case must identify a line in the module-level code, not another line within the **SUB** or **FUNCTION**.

Example

The following example shows where to put an error-handling routine that processes errors trapped within a SUB procedure:

```
CALL ShortSub
END

CatchError:
   ' Put the CatchError routine at the module level,
   ' outside the subprogram:
   PRINT "Error" ERR "caught by error handler."
RESUME NEXT

SUB ShortSub STATIC
   ON ERROR GOTO CatchError
   ERROR 62
END SUB
```

Output

```
Error 62 caught by error handler.
```

6.4 Trapping across Multiple Modules

Prior to QuickBASIC 4.5, only events could be trapped across modules. Once an event-handling routine was defined and an event trap was activated in one module, an occurrence of that event during program execution in any other module triggered a branch to the routine.

Errors could not be trapped across modules. If an error occurred in a module lacking an active error handler, program execution would halt, even if there were a handler in another module that could have taken care of the problem.

QuickBASIC 4.5 expands the scope of error trapping. Errors, as well as events, can be trapped across modules. The next two sections explain how this works and the slight remaining differences between event and error trapping.

6.4.1 Event Trapping across Modules

The output from the following program shows that a trap set for the F1 function key in the main module is triggered even when program control is in another module:

```
' ========================================================
'                        MODULE
' ========================================================
ON KEY (1) GOSUB GotF1Key
KEY (1) ON
PRINT "In main module. Press c to continue."
```

```
DO: LOOP UNTIL INKEY$ = "c"

CALL SubKey

PRINT "Back in main module. Press q to end."
DO : LOOP UNTIL INKEY$ = "q"
END

GotF1Key:
   PRINT "Handled F1 keystroke in main module."
RETURN

' ========================================================
'                        SUBKEY MODULE
' ========================================================
SUB SubKey STATIC
   PRINT "In module with SUBKEY. Press r to return."

   ' Pressing F1 here still invokes the GotF1Key
   ' subroutine in the MAIN module:
   DO: LOOP UNTIL INKEY$ = "r"
END SUB
```

Output

```
In main module. Press c to continue.
Handled F1 keystroke in main module.
In module with SUBKEY. Press r to return.
Handled F1 keystroke in main module.
Back in main module. Press q to end.
Handled F1 keystroke in main module.
```

6.4.2 Error Trapping across Modules

Errors can be trapped across multiple modules. If there is no active error handler in the module where the error occurred, BASIC looks for one in the module from which the active module was invoked, continuing back through the sequence of modules called until it finds a module with an active error handler. If it cannot find one, an error message is displayed and program execution halts.

Note that BASIC does not search through all modules—only the ones that were in the path of invocation leading to the code that produced the error.

An **ON ERROR** statement must be executed before an error occurs in order for BASIC to look for a handler. Therefore, you must put the **ON ERROR GOTO** statement where the flow of a program's control can reach it; for example, place it inside a procedure called from the main module (see the examples that follow).

Examples

The following example shows how to trap errors for a procedure in a Quick library. The `AdapterType` procedure in this library tries all possible **SCREEN** statements to see which graphics screen modes your computer's hardware supports. (See Appendix H, "Creating and Using Quick Libraries," for more information on Quick libraries.)

```
' ========== MODULE-LEVEL CODE IN QUICK LIBRARY ==========
' The error-handling routines for procedures in this
' library must be defined at this level. This level is
' executed only when there is an error in the procedures.
' ========================================================

DEFINT A-Z

CONST FALSE = 0, TRUE = NOT FALSE

.
.
.
CALL AdapterType
.
.
.
END

DitchMe:                          ' Error-handling routine
   DisplayError = TRUE
   RESUME NEXT

' ========== PROCEDURE-LEVEL CODE IN QUICK LIBRARY ========
'          Error trapping is activated at this level.
' ========================================================

SUB AdapterType STATIC

   SHARED DisplayError            ' DisplayError variable is
                                  ' shared with DitchMe error
                                  ' handler in code above.
   DIM DisplayType(1 TO 13)       ' Dimension DisplayType
                                  ' array.

   J = 1                          ' Initialize subscript
                                  ' counter for DisplayType
                                  ' array.

ON ERROR GOTO DitchMe    ' Set up the error trap.
```

```
FOR Test = 13 TO 1 STEP -1
      SCREEN Test                ' Try a SCREEN statement to
                                 ' see if it causes error.
      IF NOT DisplayError THEN   ' No error, so this is a
                                 ' valid screen mode.
          DisplayType(J) = Test  ' Store mode in array.
          J = J + 1              ' Increment the subscript
                                 ' counter.
      ELSE
          DisplayError = FALSE   ' Error; reset DisplayError.
      END IF
   NEXT Test

   SCREEN 0, 0                   ' Set 80-column text mode.
   WIDTH 80
   LOCATE 5, 10
   PRINT "Your computer supports these screen modes:"
   PRINT

   FOR I = 1 TO J                ' Print modes not causing
                                 ' errors.
      PRINT TAB(20); "SCREEN"; DisplayType(I)
   NEXT I

   LOCATE 20, 10
   PRINT "Press any key to continue..."

   DO: LOOP WHILE INKEY$ = ""

END SUB
```

If you want the error-handling routine in each module to do the exactly the same thing, you can have each routine invoke a common error-processing SUB procedure, as shown in the next example:

```
' ============================================================
'                        MAIN MODULE
' ============================================================

DECLARE SUB GlobalHandler (ModuleName$)
DECLARE SUB ShortSub ()

ON ERROR GOTO LocalHandler
ERROR 57                 ' Simulate occurrence of error 57
                         ' ("Device I/O error").
ShortSub                 ' Call the ShortSub SUB.
END
```

```
LocalHandler:
   ModuleName$ = "MAIN"
   CALL GlobalHandler(ModuleName$)   ' Call the
                                     ' GlobalHandler SUB.
RESUME NEXT

' ========================================================
'                      SHORTSUB MODULE
' ========================================================

DECLARE SUB GlobalHandler (ModuleName$)

LocalHandler:
   ModuleName$ = "SHORTSUB"
   GlobalHandler ModuleName$     ' Call GlobalHandler.
RESUME NEXT

SUB ShortSub STATIC
   ON ERROR GOTO LocalHandler
   ERROR 13              ' Simulate a "Type mismatch" error.
END SUB


' ========================================================
'                     GLOBALHANDLER MODULE
' ========================================================

SUB GlobalHandler (ModuleName$) STATIC
   PRINT "Trapped error";ERR;"in";ModuleName$;"module"
END SUB
```

Output

```
Trapped error 57 in MAIN module.
Trapped error 13 in SHORTSUB module.
```

6.5 Trapping Errors and Events in Programs Compiled with BC

If both of the following statements apply to your program, then you must use the appropriate BC command-line option listed below when compiling your program:

- You are developing the program outside the QuickBASIC environment—that is, you are using another text editor to enter your BASIC source code, then creating a stand-alone executable program from this source code with the commands BC and LINK.
- Your program contains any of the statements listed in the second column below.

Table 6.1 BC Command-Line Options for Error and Event Trapping

Command-Line Option	Statements	Explanation
/E	**ON ERROR GOTO** *line* **RESUME** *line*	The /E option tells the compiler your program does its own error trapping with **ON ERROR GOTO** and **RESUME** statements.
/V	**ON** *event* **GOSUB** *event* **ON**	The /V option tells the program to check between each statement to see if the given *event* has taken place (contrast this with the effect of /W).
/W	**ON** *event* **GOSUB** *event* **ON**	The /W option tells the program to check between each line to see if the given *event* has taken place (contrast this with the effect of /V). Since BASIC allows multiple statements on a single line, programs compiled with /W may check less frequently than those compiled with /V.
/X	**RESUME** **RESUME NEXT** **RESUME 0**	The /X option tells the compiler you have used one of the preceding forms of **RESUME** in your program to return control from an error-handling routine.

Example

The following DOS command lines compile and link a BASIC module named RMTAB.BAS, creating a stand-alone program RMTAB.EXE. (As this program is compiled without the /O option, it requires the BRUN45.LIB run-time library in order to run. See Appendix G, "Compiling and Linking from DOS," for more information on compiling and linking.) Since this module contains **ON ERROR GOTO** and **RESUME NEXT** statements, the /E and /X options are required when compiling.

```
BC RMTAB , , /E /X;
LINK RMTAB;
```

6.6 Sample Application: Trapping File-Access Errors (FILERR.BAS)

The following program gets as input both a file name and a string to search for in the file. It then lists all lines in the given file containing the specified string. If a file-access error occurs, it is trapped and dealt with in the `ErrorProc` routine.

Statements and Functions Used

This program demonstrates the use of the following error-handling statements and functions:

- **ERR**
- **ON ERROR GOTO**
- **RESUME**
- **RESUME NEXT**

Program Listing

```
' Declare symbolic constants:
CONST FALSE = 0, TRUE = NOT FALSE

DECLARE FUNCTION GetFileName$ ()

' Set up the ERROR trap and specify
' the name of the error-handling routine:
ON ERROR GOTO ErrorProc

DO
   Restart = FALSE
   CLS

   FileName$ = GetFileName$      ' Input file name.

   IF FileName$ = "" THEN
      END                        ' End if <ENTER> pressed.
   ELSE

      ' Otherwise, open the file, assigning it
      ' the next available file number:
      FileNum = FREEFILE
      OPEN FileName$ FOR INPUT AS FileNum
   END IF
```

```
IF NOT Restart THEN

      ' Input search string:
      LINE INPUT "Enter string to locate:", LocString$
      LocString$ = UCASE$(LocString$)

      ' Loop through the lines in the file,
      ' printing them if they contain the search string:
      LineNum = 1
      DO WHILE NOT EOF(FileNum)

         ' Input line from file:
         LINE INPUT #FileNum, LineBuffer$

         ' Check for string, printing the line
         ' and its number if found:
         IF INSTR(UCASE$(LineBuffer$), LocString$) <> 0 THEN
            PRINT USING "#### &"; LineNum, LineBuffer$
         END IF

         LineNum = LineNum + 1
      LOOP

      CLOSE FileNum              ' Close the file.

   END IF
LOOP WHILE Restart = TRUE

END

ErrorProc:

   SELECT CASE ERR

      CASE 64:                   ' Bad file name
         PRINT "** ERROR - Invalid file name"

         ' Get a new file name and try again:
         FileName$ = GetFileName$

         ' Resume at the statement that caused the error:
         RESUME
```

```
CASE 71:                    ' Disk not ready.
        PRINT "** ERROR - Disk drive not ready"
        PRINT "Press C to continue, R to restart, Q to quit: "
        DO
           Char$ = UCASE$(INPUT$(1))
           IF Char$ = "C" THEN
              RESUME            ' Resume where you left off.

           ELSEIF Char$ = "R" THEN
              Restart = TRUE    ' Resume at beginning.
              RESUME NEXT

           ELSEIF Char$ = "Q" THEN
              END               ' Don't resume at all.
           END IF
        LOOP

     CASE 53, 76:               ' File or path not found.
        PRINT "** ERROR - File or path not found"
        FileName$ = GetFileName$
        RESUME

     CASE ELSE:                 ' Unforeseen error.

        ' Disable error trapping and
        ' print standard system message:
        ON ERROR GOTO 0
  END SELECT

' ====================== GETFILENAME$ ======================
'           Returns a file name from user input
' ==========================================================

FUNCTION GetFileName$ STATIC
   INPUT "Enter file to search (or ENTER to quit):", FTemp$
   GetFileName$ = FTemp$
END FUNCTION
```

CHAPTER 7

7 Programming with Modules

This chapter shows how you can gain more control over your programming projects by dividing them into "modules." Modules provide a powerful organizing function by letting you divide a program into logically related parts (rather than keeping all the code in one file).

This chapter will show you how to use modules to:

- Write and test new procedures separately from the rest of the program
- Create libraries of your own **SUB** and **FUNCTION** procedures that can be added to any new program
- Combine routines from other languages (such as C or MASM) with your BASIC programs

7.1 Why Use Modules?

A module is a file that contains an executable part of your program. A complete program can be contained in a single module, or it can be divided among two or more modules.

In dividing up a program into modules, logically related sections are placed in separate files. This organization can speed and simplify the process of writing, testing, and debugging.

Dividing your program into modules has these advantages:

- Modules allow procedures to be written separately from the rest of the program, then combined with it. This arrangement is especially useful for testing the procedures, since they can then be checked outside the environment of the program.

- Two or more programmers can work on different parts of the same program without interference. This is especially helpful in managing complex programming projects.
- As you create procedures that meet your own specific programming needs, you can add these procedures to their own module. They can then be reused in new programs simply by loading that module.
- Multiple modules simplify software maintenance. A procedure used by many programs can be in one library module; if changes are needed, it only has to be modified once.

7.2 Main Modules

The module containing the first executable statement of a program is called the "main module." This statement is never part of a procedure, because execution cannot begin within a procedure.

Everything in a module except **SUB** and **FUNCTION** procedures is said to be "module-level code." In QuickBASIC, the module-level code is every statement that can be accessed without switching to a procedure-editing window. Figure 7.1 illustrates the relationship between these elements.

7.3 Modules Containing Only Procedures

A module need not contain module-level code; a module can consist of nothing but **SUB** and **FUNCTION** procedures. Indeed, this is the most important use of modules.

Modules are often used to "split off" the procedures from the body of the program. This makes it easy to divide a project among programmers, for example. Also, as you create general-purpose procedures that are useful in a variety of programs (such as procedures that evaluate matrices, send binary data to a COM port, alter strings, or handle errors), these can be stored in modules, then used in new programs simply by loading the appropriate module into QuickBASIC.

NOTE *If a procedure in a procedures-only module needs an error- or event-handling routine or a* ***COMMON SHARED*** *statement, these are inserted at module level.*

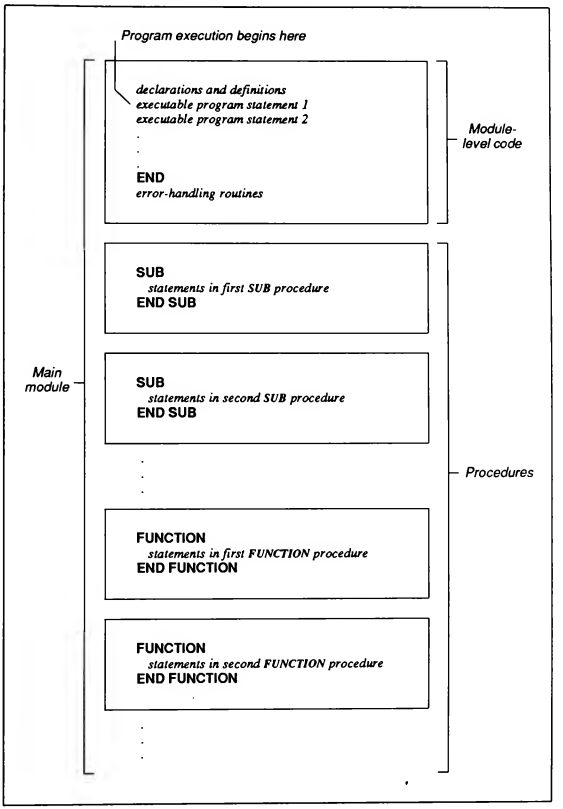


Figure 7.1 Main Module Showing Module-Level Code and Procedures

7.4 Creating a Procedures-Only Module

It's easy to create a module that contains only procedures. You can enter new procedures in a separate file, or you can move procedures from one file to another.

To create a new module:

1. Invoke QuickBASIC without opening or loading any files.
2. Write all the **SUB** and **FUNCTION** procedures you wish, but don't enter any module-level code. (Any error- or event-trapping routines and BASIC declarations needed are exceptions.)
3. Use the Save As command to name and save this module.

To move procedures from one module to another:

1. Load the files containing the procedures you want to move.
2. If the destination file already exists, use the Load File command from the File menu to load it, too. If it doesn't exist, use the Create File option from the File menu to make the new file.
3. Choose the SUBs command from the View menu and use its Move feature to transfer the procedures from the old to the new file. This transfer is made final when you quit QuickBASIC and respond Yes to the dialog box that asks whether you want to save the modified files; otherwise, the procedures remain where they were when you started.

7.5 Loading Modules

In QuickBASIC, you can load as many modules as you wish (limited only by the available memory) using the Load File command from the File menu. All the procedures in all the loaded modules can be called from any other procedure or from module-level code. If a module happens to contain a procedure that is never called, no harm occurs.

Any or all of the loaded modules can contain module-level code. QuickBASIC normally begins execution with the module-level code of the first module loaded. If you want execution to begin in a different module, use the Set Main Module command from the Run menu. Execution normally halts at the end of the designated main module; by design, QuickBASIC does not continue execution with the module-level code from other modules.

The ability to choose which module-level code gets executed is useful when comparing two versions of the same program. For example, you might want to test different user interfaces by putting each in a separate module. You can also place test code in a module containing only procedures, then use the Set Main Module command to switch between the program and the tests.

You do not have to keep track of which modules your program uses. Whenever you use the Save All command, QuickBASIC creates (or updates) a .MAK file, which lists all the modules currently loaded. The next time the main module is loaded with the Open Program command, QuickBASIC consults this .MAK file and automatically loads the modules listed in it.

7.6 Using the DECLARE Statement with Multiple Modules

The **DECLARE** statement has several important functions in QuickBASIC. Using a **DECLARE** statement will

1. Specify the sequence and data types of a procedure's parameters
2. Enable type-checking, which confirms, each time a procedure is called, that the arguments agree with the parameters in both number and data type
3. Identify a **FUNCTION** procedure's name as a procedure name, not a variable name
4. Most important of all, enable the main module to call procedures located in other modules (or Quick libraries)

QuickBASIC has its own system for automatically inserting the required **DECLARE** statements into your modules. Section 2.5.4, "Checking Arguments with the DECLARE Statement," explains the features and limitations of this system.

Despite QuickBASIC's automatic insertion of the **DECLARE** statement, you may wish to create a separate include file that contains all the **DECLARE** statements required for a program. You can update this file manually as you add and delete procedures or modify your argument lists.

If you write your programs with a text editor (rather than the QuickBASIC programming environment) and compile with BC, you must insert **DECLARE** statements manually.

7.7 Accessing Variables from Two or More Modules

You can use the **SHARED** attribute to make variables accessible both at module level and within that module's procedures. If these procedures are moved to another module, however, these variables are no longer shared.

You could pass these variables to each procedure through its argument list. This may be inconvenient, though, if you need to pass a large number of variables.

One solution is to use **COMMON** statements, which enable two or more modules to access the same group of variables. Section 2.6, "Sharing Variables with SHARED," explains how to do this.

Another solution is to use a **TYPE...END TYPE** statement to combine all the variables you wish to pass into a single structure. The argument and parameter lists then have to include only one variable name, no matter how many variables are passed.

If you are simply splitting up a program and its procedures into separate modules, either of these approaches works well. If, on the other hand, you are adding a procedure to a module (for use in other programs), you should avoid using a **COMMON** statement. Modules are supposed to make it easy to add existing procedures to new programs; **COMMON** statements complicate the process. If a procedure needs a large group of variables, it may not belong in a separate module.

7.8 Using Modules During Program Development

When you start a new programming project, you should first look through existing modules to see if there are procedures that can be reused in your new software. If any of these procedures aren't already in a separate module, you should consider moving them to one.

As your program takes shape, newly written procedures automatically become part of the program file (that is, the main module). You can move them to a separate module for testing or perhaps add them to one of your own modules along with other general-purpose procedures that are used in other programs.

Your program may need procedures written in other languages. (For example, MASM is ideal for direct interface with the hardware, FORTRAN has almost any math function you could want, Pascal allows the creation of sophisticated data structures, and C provides both structured code and direct memory access.) These procedures are compiled and linked into a Quick library for use in your program. You can also write a separate module to test Quick library procedures the same way you would test other procedures.

7.9 Compiling and Linking Modules

The end product of your programming efforts is usually a stand-alone .EXE file. You can create one in QuickBASIC by loading all of a program's modules, then selecting the Make EXE File command from the Run menu.

You can also compile modules with the BC command-line compiler, then use LINK to combine the object code. Object files from code written in other languages can be linked at the same time.

NOTE *When you use the Make EXE File command, all the module-level code and every procedure currently loaded is included in the .EXE file, whether or not the program uses this code. If you want your program to be as compact as possible, you must unload all unneeded module-level code and all unneeded procedures before compiling. The same rule applies when using BC to compile from the command line; all unused code should be removed from the files.*

7.10 Quick Libraries

Although Microsoft Quick libraries are not modules, it is important that you understand their relationship with modules.

A Quick library contains nothing but procedures. These procedures can be written not only in QuickBASIC, but also in other Microsoft languages as well (C, Pascal, FORTRAN, and MASM).

A Quick library contains only compiled code. (Modules contain QuickBASIC source code.) A Quick library is created by linking compiled object code (.OBJ files). The code in a Quick library can come from any combination of Microsoft languages. Appendix H, "Creating and Using Quick Libraries," explains how to create Quick libraries from object code and how to add new object code to existing Quick libraries.

Quick libraries have several uses:

- They provide an interface between QuickBASIC and other languages.
- They allow designers to hide proprietary software. Updates and utilities can be distributed as Quick libraries without revealing the source code.
- They load faster and are usually smaller than modules. If a large program with many modules loads slowly, converting the nonmain modules into a Quick library will improve loading performance.

Note, however, that modules are the easiest way to work on procedures during development because modules are immediately ready to run after each edit; you don't have to recreate the Quick library. If you want to put your QuickBASIC procedures in a Quick library, wait until the procedures are complete and thoroughly debugged.

When a Quick library is created, any module-level code in the file it was created from is automatically included. However, other modules cannot access this code, so it just wastes space. Before converting a module to a Quick library, be sure that all module-level statements (except any error or event handlers and declarations that are used by the procedures) have been removed.

NOTE *Quick libraries are not included in .MAK files and must be loaded with the /L option when you run QuickBASIC. A Quick library has the file extension .QLB. During the process of creating the Quick library, another library file with the extension .LIB is created. This file contains the same code as the Quick library but in a form that allows it to be linked with the rest of the program to create a stand-alone applicaation.*

If you use Quick libraries to distribute proprietary code (data-manipulation procedures, for example), be sure to include the .LIB files so that your customers can create stand-alone applications that use these procedures. Otherwise, they will be limited to running applications within the QuickBASIC environment.

7.10.1 Creating Quick Libraries

You can create a Quick library of QuickBASIC procedures with the Make Library command from the Run menu. The Quick library created contains every procedure currently loaded, whether or not your program calls it. (It also contains all the module-level code.) If you want the Quick library to be compact, be sure to remove all unused procedures and all unnecessary module-level code first.

You can create as many Quick libraries as you like, containing whatever combination of procedures you wish. However, only one Quick library can be loaded into QuickBASIC at a time. (You would generally create application-specific Quick libraries, containing only the procedures a particular program needs.) Large Quick libraries can be created by loading many modules, then using the Make Library command.

You can also compile one or more modules with the BC command, then link the object code files to create a Quick library. Quick libraries of procedures written

in other languages are created the same way. In linking, you are not limited to one language; the object-code files from any number of Microsoft languages can be combined in one Quick library. Appendix H, "Creating and Using Quick Libraries," explains how to convert object-code (.OBJ) files into Quick libraries.

7.11 Tips for Good Programming with Modules

You can use modules in any way you think will improve your program or help organize your work. The following suggestions are offered as a guide.

1. Think and organize first.

   When you start on a new project, make a list of the operations you want to be performed by procedures. Then look through your own procedure library to see if there are any you can use, either as-is or with slight modifications. Don't waste time "reinventing the wheel."

2. Write generalized procedures with broad application.

   Try to write procedures that are useful in a wide variety of programs. Don't, however, make the procedure needlessly complex. A good procedure is a simple, finely honed tool, not a Swiss army knife.

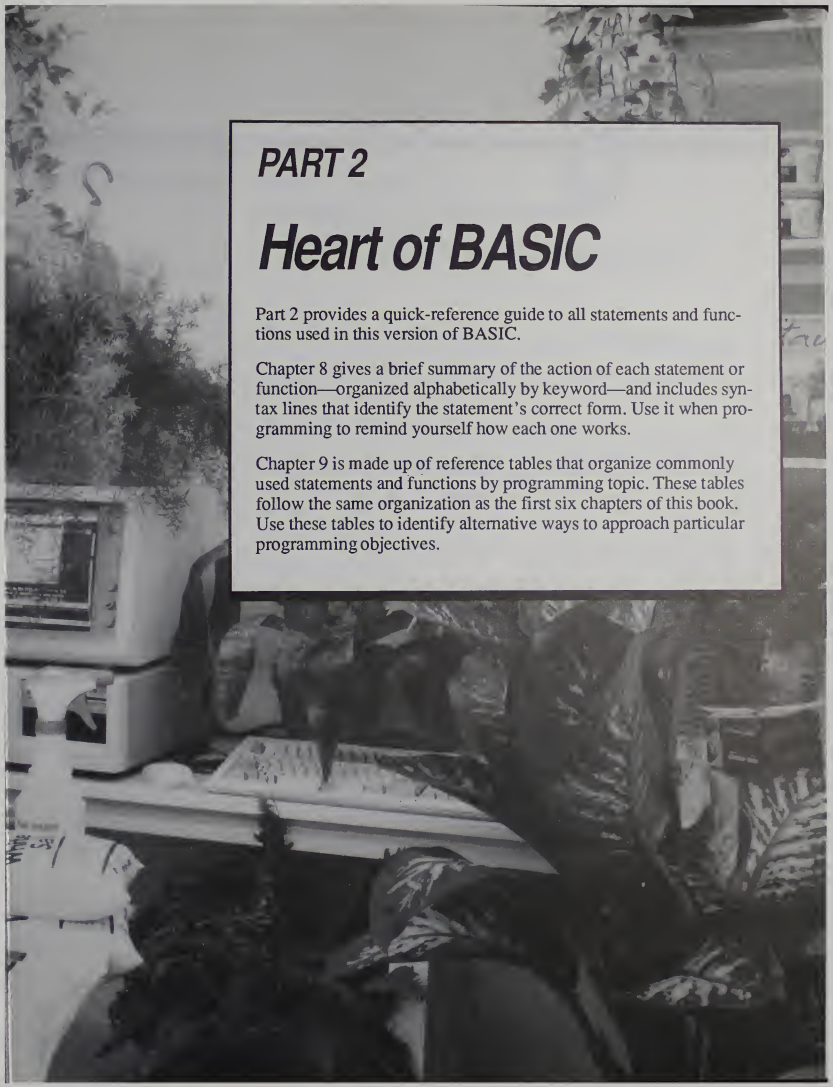
   It is sometimes useful to alter an existing procedure to work in a new program. This might require modifying programs you've already written, but it's worth the trouble if the revised procedure is more powerful or has broader application.

3. When creating your own procedure modules, keep logically separate procedures in separate modules.

   It makes sense to put string-manipulation procedures in one module, matrix-handling procedures in another, and data-communication procedures in a third. This arrangement avoids confusion and makes it easy to find the procedure you need.

PART 2

Heart of BASIC



PART 2

Heart of BASIC

Part 2 provides a quick-reference guide to all statements and functions used in this version of BASIC.

Chapter 8 gives a brief summary of the action of each statement or function—organized alphabetically by keyword—and includes syntax lines that identify the statement's correct form. Use it when programming to remind yourself how each one works.

Chapter 9 is made up of reference tables that organize commonly used statements and functions by programming topic. These tables follow the same organization as the first six chapters of this book. Use these tables to identify alternative ways to approach particular programming objectives.

CHAPTERS

8 Statement and Function Summary 265

9 Quick-Reference Tables 297

CHAPTER 8

Statement and Function Summary

8

This chapter summarizes QuickBASIC statements and functions. Each statement or function name is followed by an action description and a syntax line. The syntax line shows exactly what you must enter to use the statement.

Complete details of the typographic conventions used in syntax lines are given in the introduction to this manual. In general, items you must type exactly as shown are in boldface type; placeholders for items of information you must supply are in Italic type. Square brackets indicate optional items.

The QB Advisor (QuickBASIC's on-line language help) provides for each statement or function:

- A summary of the statement's action and syntax (the QuickSCREEN)
- Complete details of how to use the statement, including explanations of the placeholders
- One of more program examples illustrating statement use

You can access this information by putting the cursor on any BASIC keyword displayed in the QuickBASIC View window and pressing F1.

The statements and functions in this chapter are grouped by topic in Chapter 9, "Quick-Reference Tables." Use Chapter 9 to find out what statements are available for a particular programming task.

ABS Function

Returns the absolute value of a numeric expression

SYNTAX **ABS**(*numeric-expression*)

ASC Function

Returns a numeric value that is the ASCII code for the first character in a string expression

SYNTAX **ASC**(*stringexpression*)

ATN Function

Returns the arctangent of a numeric expression (the angle whose tangent is equal to the numeric expression)

SYNTAX **ATN**(*numeric-expression*)

BEEP Statement

Sounds the speaker

SYNTAX **BEEP**

BLOAD Statement

Loads a memory-image file, created by **BSAVE**, into memory from an input file or device

SYNTAX **BLOAD** *filespec* [[, *offset*]]

BSAVE Statement

Transfers the contents of an area of memory to an output file or device

SYNTAX **BSAVE** *filespec*, *offset*, *length*

CALL Statement (BASIC Procedures)

Transfers control to a BASIC **SUB**

SYNTAX 1 **CALL** *name*[[(*argumentlist*)]]

SYNTAX 2 *name*[[*argumentlist*]]

CALL, CALLS Statement (Non-BASIC Procedures)

Transfers control to a procedure written in another language

SYNTAX 1 **CALL** *name* [[(*call-argumentlist*)]]

SYNTAX 2 *name* [[*call-argumentlist*]]

SYNTAX 3 **CALLS** *name* [[(*calls-argumentlist*)]]

CALL INT86OLD Statements

Allows programs to perform DOS system calls

SYNTAX **CALL INT86OLD** (*int_no, in_array*( ), *out_array*( ))
CALL INT86XOLD (*int_no, in_array*( ), *out_array*( ))

CALL ABSOLUTE Statement

Transfers control to a machine-language procedure

SYNTAX **CALL ABSOLUTE** ([*argumentlist,*]*integervariable*)

CALL INTERRUPT Statements

Allows BASIC programs to perform DOS system calls

SYNTAX **CALL INTERRUPT** (*interruptnum, inregs, outregs*)
CALL INTERRUPTX (*interruptnum, inregs, outregs*)

CDBL Function

Converts a numeric expression to a double-precision number

SYNTAX **CDBL**(*numeric-expression*)

CHAIN Statement

Transfers control from the current program to another program

SYNTAX **CHAIN** *filespec*

CHDIR Statement

Changes the current default directory for the specified drive

SYNTAX **CHDIR** *pathspec*

CHR$ Function

Returns a one-character string whose ASCII code is the argument

SYNTAX **CHR$(***code***)**

CINT Function

Converts a numeric expression to an integer by rounding the fractional part of the expression

SYNTAX **CINT(***numeric-expression***)**

CIRCLE Statement

Draws an ellipse or circle with a specified center and radius

SYNTAX **CIRCLE [[STEP]] (*x,y*), *radius*[[,[*color*]][[,[*start*]][,[*end*]][, *aspect*]]]]]**

CLEAR Statement

Reinitializes all program variables, closes files, and sets the stack size

SYNTAX **CLEAR [[,,*stack*]]**

CLNG Function

Converts a numeric expression to a long (4-byte) integer by rounding the fractional part of the expression

SYNTAX **CLNG(***numeric-expression***)**

CLOSE Statement

Concludes I/O to a file or device

SYNTAX **CLOSE [[[#]] *filenumber* [[,[#]] *filenumber*]...]]**

CLS Statement

Clears the screen

SYNTAX **CLS** [[(**0** | **1** | **2**)]]

COLOR Statement

Selects display colors

SYNTAX	**COLOR** [[*foreground*]][,[*background*]][, *border*]]	Screen mode 0
	COLOR [[*background*]][, *palette*]]	Screen mode 1
	COLOR [[*foreground*]][, *background*]	Screen modes 7–10
	COLOR [[*foreground*]]	Screen modes 12–13

COM Statements

Enables, disables, or inhibits event trapping of communications activity on a specified port

SYNTAX **COM**(*n*) **ON**
COM(*n*) **OFF**
COM(*n*) **STOP**

COMMAND$ Function

Returns the command line used to invoke the program

SYNTAX **COMMAND$**

COMMON Statement

Defines global variables for sharing between modules or for chaining to another program

SYNTAX **COMMON** [[**SHARED**]] [[/*blockname*/]] *variablelist*

CONST Statement

Declares symbolic constants to use in place of numeric or string values

SYNTAX **CONST** *constantname* = *expression* [[, *constantname* = *expression*]]...

COS Function

Returns the cosine of an angle given in radians

SYNTAX **COS**(*numeric-expression*)

CSNG Function

Converts a numeric expression to a single-precision value

SYNTAX **CSNG**(*numeric-expression*)

CSRLIN Function

Returns the current line (row) position of the cursor

SYNTAX **CSRLIN**

CVI, CVS, CVL, CVD Functions

Convert strings containing numeric values to numbers

SYNTAX **CVI** (*2-byte-string*)
CVS (*4-byte-string*)
CVL (*4-byte-string*)
CVD (*8-byte-string*)

CVSMBF, CVDMBF Functions

Convert strings containing Microsoft Binary format numbers to IEEE-format numbers

SYNTAX **CVSMBF** (*4-byte-string*)
CVDMBF (*8-byte-string*)

DATA Statement

Stores the numeric and string constants used by a program's **READ** statements

SYNTAX **DATA** *constant1* [[, *constant2*]]...

DATE$ Function

Returns a string containing the current date

SYNTAX **DATE$**

DATE$ Statement

Sets the current date

SYNTAX **DATE$** = *stringexpression*

DECLARE Statement (BASIC Procedures)

Declares references to BASIC procedures and invokes argument type checking

SYNTAX **DECLARE** { **FUNCTION** | **SUB** } *name* [[([*parameterlist*])]]

DECLARE Statement (Non-BASIC Procedures)

Declares calling sequences for external procedures written in other languages

SYNTAX 1 **DECLARE FUNCTION** *name* [[**CDECL**]]
[[**ALIAS** "*aliasname*"]][[([*parameterlist*])]]

SYNTAX 2 **DECLARE SUB** *name* [[**CDECL**]] [[**ALIAS** "*aliasname*"]][[([*parameterlist*])]]

DEF FN Statement

Defines and names a function

SYNTAX 1 **DEF FN***name*[[(*parameterlist*)]] = *expression*

SYNTAX 2
```
DEF FNname[[( parameterlist)]]
.
.
.
FNname = expression
.
.
.
END DEF
```

DEF SEG Statement

Sets the current segment address for a subsequent **PEEK** function or a **BLOAD**, **BSAVE**, **CALL ABSOLUTE**, or **POKE** statement

SYNTAX **DEF SEG** [[=*address*]]

DEF*type* Statements

Set the default data type for variables, **DEF FN** functions, and **FUNCTION** procedures

SYNTAX **DEFINT** *letterrange* [[, *letterrange*]]...
DEFSNG *letterrange* [[, *letterrange*]]...
DEFDBL *letterrange* [[, *letterrange*]]...
DEFLNG *letterrange* [[, *letterrange*]]...
DEFSTR *letterrange* [[, *letterrange*]]...

DIM Statement

Declares a variable and allocates storage space

SYNTAX **DIM** [[**SHARED**]] *variable*[[(*subscripts*)]][[**AS** *type*]]
[[,*variable*[[(*subscripts*)]][[**AS***type*]]]]...

DO...LOOP Statements

Repeats a block of statements while a condition is true or until a condition becomes true

SYNTAX 1
```
DO
    [[statementblock]]
LOOP [[{WHILE | UNTIL} booleanexpression]]
```

SYNTAX 2
```
DO [[{WHILE | UNTIL} booleanexpression]]
    [[statementblock]]
LOOP
```

DRAW Statement

Draws an object defined by *stringexpression*

SYNTAX **DRAW** *stringexpression*

END Statement

Ends a BASIC program, procedure, or block

SYNTAX **END** [[{**DEF** | **FUNCTION** | **IF** | **SELECT** | **SUB** | **TYPE**}]]

ENVIRON$ Function

Retrieves an environment string from the DOS environment-string table

SYNTAX **ENVIRON$** (*environmentstring*)
ENVIRON$ (*n*)

ENVIRON Statement

Modifies a parameter in the DOS environment-string table

SYNTAX **ENVIRON** *stringexpression*

EOF Function

Tests for the end-of-file condition

SYNTAX **EOF**(*filenumber*)

ERASE Statement

Reinitializes the elements of static arrays; deallocates dynamic arrays

SYNTAX **ERASE** *arrayname* [[, *arrayname*...]]

ERDEV, ERDEV$ Functions

Provides device-specific status information after an error

SYNTAX **ERDEV**
ERDEV$

ERR, ERL Functions

Return error status

SYNTAX **ERR**
ERL

ERROR Statement

Simulates the occurrence of a BASIC error or allows the user to define error codes

SYNTAX **ERROR** *integerexpression*

EXIT Statement

Exits a **DEF FN** function, **DO...LOOP** or **FOR...NEXT** loop, **FUNCTION**, or **SUB**

SYNTAX **EXIT** {**DEF** | **DO** | **FOR** | **FUNCTION** | **SUB** }

EXP Function

Calculates the exponential function

SYNTAX **EXP**(*x*)

FIELD Statement

Allocates space for variables in a random-access file buffer

SYNTAX **FIELD** [[#]] *filenumber*, *fieldwidth* **AS** *stringvariable*...

FILEATTR Function

Returns information about an open file

SYNTAX **FILEATTR**(*filenumber*, *attribute*)

FILES Statement

Prints the names of files residing on the specified disk

SYNTAX **FILES** [[*filespec*]]

FIX Function

Returns the truncated integer part of *x*

SYNTAX **FIX**(*x*)

FOR...NEXT Statement

Repeats a group of instructions a specified number of times

SYNTAX **FOR** *counter* = *start* **TO** *end* [[**STEP** *increment*]]

.
.
.

NEXT [[*counter* [[, *counter*...]]]]

FRE Function

Returns the amount of available memory

SYNTAX 1 **FRE**(*numeric-expression*)

SYNTAX 2 **FRE**(*stringexpression*)

FREEFILE Function

Returns the next free BASIC file number

SYNTAX **FREEFILE**

FUNCTION Statement

Declares the name, the parameters, and the code that form the body of a **FUNCTION** procedure

SYNTAX **FUNCTION** *name* [[(*parameter-list*)]][[**STATIC**]]
.
.
.
name = *expression*
.
.
.
END FUNCTION

GET Statement—File I/O

Reads from a disk file into a random-access buffer or variable

SYNTAX **GET** [[#]] *filenumber*[[,[*recordnumber*]][[,*variable*]]]

GET Statement—Graphics

Stores graphic images from the screen

SYNTAX **GET** [[**STEP**]](*x1*,*y1*) – [[**STEP**]](*x2*,*y2*), *arrayname*[[(*indices*)]]

GOSUB...RETURN Statements

Branches to, and returns from, a subroutine

SYNTAX **GOSUB** {*linelabel1* | *linenumber1* }
.
.
.
RETURN [[*linelabel2* | *linenumber2*]]

GOTO Statement

Branches unconditionally to the specified line

SYNTAX **GOTO** {*linelabel* | *linenumber*}

HEX$ Function

Returns a string that represents the hexadecimal value of the decimal argument *expression*

SYNTAX **HEX$** (*expression*)

IF...THEN...ELSE Statements

Allows conditional execution, based on the evaluation of a Boolean expression

SYNTAX 1 (SINGLE LINE) **IF** *booleanexpression* **THEN** *thenpart* [[**ELSE** *elsepart*]]

SYNTAX 2 (BLOCK)
IF *booleanexpression1* **THEN**
    [[*statementblock-1*]]
[[**ELSEIF** *booleanexpression2* **THEN**
    [[*statementblock-2*]]]
.
.
.
[[**ELSE**
    [[*statement-blockn*]]]
END IF

INKEY$ Function

Reads a character from the keyboard

SYNTAX **INKEY$**

INP Function

Returns the byte read from an I/O port

SYNTAX **INP(** *port*)

INPUT$ Function

Returns a string of characters read from the specified file

SYNTAX **INPUT$** (*n*[[,[[#]] *filenumber*]])

INPUT Statement

Allows input from the keyboard during program execution

SYNTAX **INPUT**[[;]][["*promptstring*" {; | ,}]] *variablelist*

INPUT # Statement

Reads data items from a sequential device or file and assigns them to variables

SYNTAX **INPUT #** *filenumber, variablelist*

INSTR Function

Returns the character position of the first occurrence of a string in another string

SYNTAX **INSTR(**[[*start*,]]*stringexpression1, stringexpression2*)

INT Function

Returns the largest integer less than or equal to *numeric-expression*

SYNTAX **INT(***numeric-expression*)

IOCTL$ Function

Receives a control data string from a device driver

SYNTAX **IOCTL$** ([[#]] *filenumber*)

IOCTL Statement

Transmits a control data string to a device driver

SYNTAX **IOCTL** [[#]] *filenumber, string*

KEY Statements

Assign soft-key string values to function keys, then display the values and enable or disable the FUNCTION key display line

SYNTAX **KEY** *n, stringexpression*
KEY LIST
KEY ON
KEY OFF

KEY(n) Statements

Start or stop trapping of specified keys

SYNTAX **KEY(***n***) ON**
KEY(*n***) OFF**
KEY(*n***) STOP**

KILL Statement

Deletes a file from disk

SYNTAX **KILL** *filespec*

LBOUND Function

Returns the lower bound (smallest available subscript) for the indicated dimension of an array

SYNTAX **LBOUND(***array*[[, *dimension*]])

LCASE$ Function

Returns a string expression with all letters in lower-case

SYNTAX **LCASE$ (***stringexpression***)**

LEFT$ Function

Returns a string consisting of the leftmost *n* characters of a string

SYNTAX LEFT$ (*stringexpression,n*)

LEN Function

Returns the number of characters in a string or the number of bytes required by a variable

SYNTAX LEN(*stringexpression*)
LEN(*variable*)

LET Statement

Assigns the value of an expression to a variable

SYNTAX [[LET]]*variable=expression*

LINE Statement

Draws a line or box on the screen

SYNTAX LINE [[[STEP]] (*x1,y1*)]] – [[STEP]] (*x2,y2*) [[,[[*color*]][[,[[B[[F]]]][[, *style*]]]]]

LINE INPUT Statement

Inputs an entire line (up to 255 characters) to a string variable, without the use of delimiters

SYNTAX LINE INPUT[[;]] [["*promptstring*";]] *stringvariable*

LINE INPUT # Statement

Reads an entire line without delimiters from a sequential file to a string variable

SYNTAX LINE INPUT #*filenumber, stringvariable*

LOC Function

Returns the current position within the file

SYNTAX LOC(*filenumber*)

LOCATE Statement

Moves the cursor to the specified position

SYNTAX LOCATE[[*row*]][,[[*column*]][,[[*cursor*]][,[*start*, *stop*]]]]

LOCK...UNLOCK Statement

Controls access by other processes to all or part of an opened file

SYNTAX LOCK [[#]] *filenumber* [[,{*record* | [[*start*]] TO *end*}]]

.
.
.

UNLOCK [[#]] *filenumber* [[,{*record* | [[*start*]] TO *end*}]]

LOF Function

Returns the length of the named file in bytes

SYNTAX LOF(*filenumber*)

LOG Function

Returns the natural logarithm of a numeric expression

SYNTAX LOG(*n*)

LPOS Function

Returns the current position of the line printer's print head within the printer buffer

SYNTAX LPOS(*n*)

LPRINT, LPRINT USING Statements

Prints data on the printer LPT1:

SYNTAX 1 LPRINT [[*expressionlist*]] [[{; | ,}]]

SYNTAX 2 LPRINT USING *formatstring*; *expression-list* [[{; | ,}]]

LSET Statement

Moves data from memory to a random-access file buffer (in preparation for a PUT statement), copies one record variable to another, or left-justifies the value of a string in a string variable

SYNTAX **LSET** {*stringvariable=stringexpression* | *stringexpression1=stringexpression2*}

LTRIM$ Function

Returns a copy of a string with leading spaces removed

SYNTAX **LTRIM$**(*stringexpression*)

MID$ Function

Returns a substring of a string

SYNTAX **MID$** (*stringexpression, start*[, *length*])

MID$ Statement

Replaces a portion of a string variable with another string

SYNTAX **MID$** (*stringvariable, start*[, *length*])=*stringexpression*

MKD$, MKI$, MKL$, MKS$ Functions

Converts numeric values to string values

SYNTAX **MKI$** (*integerexpression*)
MKS$ (*single-precision-expression*)
MKL$ (*long-integer-expression*)
MKD$ (*double-precision-expression*)

MKDIR Statement

Creates a new directory

SYNTAX **MKDIR** *pathname*

MKSMBF$, MKDMBF$ Functions

Converts an IEEE-format number to a string containing a Microsoft Binary format number

SYNTAX MKSMBF$ (*single-precision-expression*)
MKDMBF$ (*double-precision-expression*)

NAME Statement

Changes the name of a disk file or directory

SYNTAX NAME *oldfilename* AS *newfilename*

OCT$ Function

Returns a string representing the octal value of the numeric argument

SYNTAX OCT$ (*numeric-expression*)

ON ERROR Statement

Enables error handling and specifies the first line of the error-handling routine

SYNTAX ON ERROR GOTO *line*

ON *event* Statements

Indicates the first line of an event-trapping subroutine

SYNTAX ON *event* GOSUB { *linenumber* | *linelabel* }

ON UEVENT GOSUB Statement

Defines the event-handler for a user-defined event

SYNTAX ON UEVENT GOSUB { *linenumber* | *linelabel* }

ON...GOSUB, ON...GOTO Statements

Branches to one of several specified lines, depending on the value of an expression

SYNTAX 1 ON *expression* GOSUB { *line-number-list* | *line-label-list* }

SYNTAX 2 ON *expression* GOTO { *line-number-list* | *line-label-list* }

OPEN Statement

Enables I/O to a file or device

SYNTAX 1 **OPEN** *file* [[**FOR** *mode1*]] [[**ACCESS** *access*]] [[*lock*]] **AS** [[#]] *filenum* [[**LEN**=*reclen*]]

SYNTAX 2 **OPEN** *mode2*,[[#]] *filenum*,*file* [[, *reclen*]]

OPEN COM Statement

Opens and initializes a communications channel for I/O

SYNTAX **OPEN "COM***n*: *optlist1 optlist2*" [[**FOR** *mode*]] **AS** [[#]] *filenum* [[**LEN**=*reclen*]]

OPTION BASE Statement

Declares the default lower bound for array subscripts

SYNTAX **OPTION BASE** *n*

OUT Statement

Sends a byte to a machine I/O port

SYNTAX **OUT** *port*, *data*

PAINT Statement

Fills a graphics area with the color or pattern specified

SYNTAX **PAINT** [[**STEP**]] (*x*,*y*)[[,[[*paint*]] [[,[[*bordercolor*]] [[, *background*]]]]]

PALETTE, PALETTE USING Statements

Changes one or more of the colors in the palette

SYNTAX **PALETTE** [[*attribute*, *color*]]
PALETTE USING *array-name* [[(*array-index*)]]

PCOPY Statement

Copies one screen page to another

SYNTAX **PCOPY** *sourcepage*, *destinationpage*

PEEK Function

Returns the byte stored at a specified memory location

SYNTAX **PEEK(***address***)**

PEN Function

Reads the lightpen coordinates

SYNTAX **PEN(***n***)**

PEN ON, OFF, and STOP Statements

Enables, disables, or suspends lightpen event trapping

SYNTAX **PEN ON**
PEN OFF
PEN STOP

PLAY Function

Returns the number of notes currently in the background-music queue

SYNTAX **PLAY (***n***)**

PLAY Statement

Plays music as specified by a string

SYNTAX **PLAY** *commandstring*

PLAY ON, OFF, and STOP Statements

PLAY ON enables play event trapping, **PLAY OFF** disables play event trapping, and **PLAY STOP** suspends play event trapping.

SYNTAX **PLAY ON**
PLAY OFF
PLAY STOP

PMAP Function

Maps view-coordinate expressions to physical locations, or maps physical expressions to a view-coordinate location

SYNTAX **PMAP** (*expression,function*)

POINT Function

Reads the color number of a pixel from the screen or returns the pixel's coordinates

SYNTAX **POINT** (*x,y*)
POINT (*number*)

POKE Statement

Writes a byte into a memory location

SYNTAX **POKE** *address, byte*

POS Function

Returns the current horizontal position of the cursor

SYNTAX **POS(0)**

PRESET Statement

Draws a specified point on the screen

SYNTAX **PRESET** [[**STEP**]](*xcoordinate,ycoordinate*) [[, *color*]]

PRINT Statement

Outputs data on the screen

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

PRINT #, PRINT # USING Statements

Writes data to a sequential file

SYNTAX **PRINT** #*filenumber*,[[**USING** *stringexpression*;]] *expressionlist* [[{, | ;}]]

PRINT USING Statement

Prints strings or numbers using a specified format

SYNTAX **PRINT USING** *formatstring*; *expressionlist* [[{, | ;}]]

PSET Statement

Draws a point on the screen

SYNTAX **PSET** [[**STEP**]](*xcoordinate*,*ycoordinate*) [[, *color*]]

PUT Statement—File I/O

Writes from a variable or a random-access buffer to a file

SYNTAX **PUT** [[#]] *filenumber*[[,[[*recordnumber*]][,*variable*]]]
PUT [[#]] *filenumber*[[,{*recordnumber* | *recordnumber*, *variable* | ,*variable*}]]

PUT Statement—Graphics

Places a graphic image obtained by a **GET** statement onto the screen

SYNTAX **PUT** [[**STEP**]] (*x*,*y*), *arrayname*[[(*indices*)]] [[, *actionverb*]]

RANDOMIZE Statement

Initializes (reseeds) the random-number generator

SYNTAX **RANDOMIZE**[[*expression*]]

READ Statement

Reads values from a **DATA** statement and assigns the values to variables

SYNTAX **READ** *variablelist*

REDIM Statement

Changes the space allocated to an array that has been declared **$DYNAMIC**

SYNTAX **REDIM** [[**SHARED**]] *variable*(*subscripts*)[[**AS** *type*]] [[,*variable*(*subscripts*)[[**AS** *type*]]]]...

REM Statement

Allows explanatory remarks to be inserted in a program

SYNTAX 1 **REM** *remark*

SYNTAX 2 **'** *remark*

RESET Statement

Closes all disk files

SYNTAX **RESET**

RESTORE Statement

Allows **DATA** statements to be reread from a specified line

SYNTAX **RESTORE** [[{*linenumber* | *linelabel* }]]

RESUME Statement

Continues program execution after an error-trapping routine has been invoked

SYNTAX **RESUME** [[**0**]]
RESUME NEXT
RESUME { *linenumber* | *linelabel* }

RETURN Statement

Returns control from a subroutine

SYNTAX **RETURN** [[{*linenumber* | *linelabel* }]]

RIGHT$ Function

Returns the rightmost *n* characters of a string

SYNTAX **RIGHT$** (*stringexpression*, *n*)

RMDIR Statement

Removes an existing directory

SYNTAX **RMDIR** *pathname*

RND Function

Returns a single-precision random number between 0 and 1

SYNTAX **RND**[[(*n*)]]

RSET Statement

Moves data from memory to a random-access file buffer (in preparation for a **PUT** statement) or right-justifies the value of a string in a string variable

SYNTAX **RSET** *stringvariable=stringexpression*

RTRIM$ Function

Returns a string with trailing (right-hand) spaces removed

SYNTAX **RTRIM$**(*stringexpression*)

RUN Statement

Restarts the program currently in memory or executes a specified program

SYNTAX **RUN** [[{ *linenumber* | *commandline* }]]

SADD Function

Returns the address of the specified string expression

SYNTAX **SADD**(*stringvariable*)

SCREEN Function

Reads a character's ASCII value or its color from a specified screen location

SYNTAX **SCREEN**(*row*, *column*[[, *colorflag*]])

SCREEN Statement

Sets the specifications for the display screen

SYNTAX **SCREEN** [[*mode*]] [[,[[*colorswitch*]]]][[,[[*apage*]]]][[,[[*vpage*]]]]

SEEK Function

Returns the current file position

SYNTAX **SEEK(***filenumber***)**

SEEK Statement

Sets the position in a file for the next read or write

SYNTAX **SEEK [[#]]** ***filenumber, position***

SELECT CASE Statement

Executes one of several statement blocks depending on the value of an expression

SYNTAX
```
SELECT CASE testexpression
CASE expressionlist1
    [[statementblock-1]]
[[CASE expressionlist2
    [[statementblock-2]]]]
.
.
.
[[CASE ELSE
    [[statementblock-n]]]]
END SELECT
```

SETMEM Function

Changes the amount of memory used by the far heap—the area where far objects and internal tables are stored

SYNTAX **SETMEM(***numeric-expression***)**

SGN Function

Indicates the sign of a numeric expression

SYNTAX **SGN(***numeric-expression***)**

SHARED Statement

Gives a **SUB** or **FUNCTION** procedure access to variables declared at the module level without passing them as parameters

SYNTAX **SHARED** *variable* [[**AS** *type*]] [[, *variable* [[**AS** *type*]]]]...

SHELL Statement

Exits the BASIC program, runs a .COM, .EXE, or .BAT program or a DOS command, and returns to the program at the line following the **SHELL** statement

SYNTAX **SHELL** [[*commandstring*]]

SIN Function

Returns the sine of the angle *x*, where *x* is in radians

SYNTAX **SIN**(*x*)

SLEEP Statement

Suspends execution of the calling program

SYNTAX **SLEEP** [[*seconds*]]

SOUND Statement

Generates sound through the speaker

SYNTAX **SOUND** *frequency, duration*

SPACE$ Function

Returns a string of spaces of length *n*

SYNTAX **SPACE$**(*n*)

SPC Function

Skips *n* spaces in a **PRINT** statement

SYNTAX **SPC**(*n*)

SQR Function

Returns the square root of *n*

SYNTAX **SQR(*n*)**

STATIC Statement

Makes simple variables or arrays local to either a **DEF FN** function, a **FUNCTION**, or a **SUB** and preserves values between calls

SYNTAX **STATIC** *variablelist*

STICK Function

Returns the *x* and *y* coordinates of the two joysticks

SYNTAX **STICK(*n*)**

STOP Statement

Terminates the program

SYNTAX **STOP**

STR$ Function

Returns a string representation of the value of a numeric expression

SYNTAX **STR$ (*numeric-expression*)**

STRIG Function and Statement

Returns the status of a specified joystick trigger

SYNTAX 1 (FUNCTION) **STRIG(*n*)**

SYNTAX 2 (STATEMENT) **STRIG {ON | OFF}**

STRIG ON, OFF, and STOP Statements

Enable, disable, or inhibit trapping of joystick activity

SYNTAX **STRIG(*n*) ON**
STRIG(*n*) OFF
STRIG(*n*) STOP

STRING$ Function

Returns a string whose characters all have a given ASCII code or whose characters are all the first character of a string expression

SYNTAX **STRING$ (*m*, *n*)**
STRING$ (*m*, *stringexpression*)

SUB Statements

Marks the beginning and end of a subprogram

SYNTAX **SUB *globalname*[[(*parameterlist*)]] [[STATIC]]**
.
.
.
[[EXIT SUB]]
.
.
.
END SUB

SWAP Statement

Exchanges the values of two variables

SYNTAX **SWAP *variable1*, *variable2***

SYSTEM Statement

Closes all open files and returns control to the operating system

SYNTAX **SYSTEM**

TAB Function

Moves the print position

SYNTAX **TAB**(*column*)

TAN Function

Returns the tangent of the angle *x*, where *x* is in radians

SYNTAX **TAN**(*x*)

TIME$ Function

Returns the current time from the operating system

SYNTAX **TIME$**

TIME$ Statement

Sets the time

SYNTAX **TIME$**=*stringexpression*

TIMER Function

Returns the number of seconds elapsed since midnight

SYNTAX **TIMER**

TIMER ON, OFF, and STOP Statements

Enables, disables, or inhibits timer event trapping

SYNTAX **TIMER ON**
TIMER OFF
TIMER STOP

TRON/TROFF Statements

Traces the execution of program statements

SYNTAX **TRON**
TROFF

TYPE Statement

Defines a data type containing one or more elements

SYNTAX **TYPE** *usertype*
elementname **AS** *typename*
elementname **AS** *typename*
.
.
.
END TYPE

UBOUND Function

Returns the upper bound (largest available subscript) for the indicated dimension of an array

SYNTAX **UBOUND**(*array*[, *dimension*])

UCASE$ Function

Returns a string expression with all letters in uppercase

SYNTAX **UCASE$** (*stringexpression*)

UEVENT Statement

Enables, disables, or suspends event trapping for a user-defined event

SYNTAX **UEVENT ON**
UEVENT OFF
UEVENT STOP

UNLOCK Statement

Releases locks applied to parts of a file

SYNTAX **UNLOCK** [[#]] *filenumber* [[, {*record* | [[*start*]] **TO** *end*}]]

VAL Function

Returns the numeric value of a string of digits

SYNTAX **VAL**(*stringexpression*)

VARPTR, VARSEG Functions

Return the address of a variable

SYNTAX **VARPTR**(*variablename*)
VARSEG(*variablename*)

VARPTR$ Function

Returns a string representation of a variable's address for use in **DRAW** and **PLAY** statements

SYNTAX **VARPTR$**(*variablename*)

VIEW Statement

Defines screen limits for graphics output

SYNTAX **VIEW** [[**SCREEN**] (*x1*,*y1*) – (*x2*,*y2*) [,[*color*] [,*border*]]]

VIEW PRINT Statement

Sets the boundaries of the screen text viewport

SYNTAX **VIEW PRINT** [*topline* **TO** *bottomline*]

WAIT Statement

Suspends program execution while monitoring the status of a machine input port

SYNTAX **WAIT** *portnumber*, *and-expression*[, *xor-expression*]

WHILE...WEND Statement

Executes a series of statements in a loop, as long as a given condition is true

SYNTAX **WHILE** *condition*
.
.
.
[*statements*]
.
.
.
WEND

WIDTH Statement

Assigns an output-line width to a file or device or changes the number of columns and lines displayed on the screen

SYNTAX **WIDTH** [[*columns*]][,*lines*]]
WIDTH {#*filenumber* | *device*} , *width*
WIDTH LPRINT *width*

WINDOW Statement

Defines the logical dimensions of the current viewport

SYNTAX **WINDOW** [[[**SCREEN**]] (*x1*,*y1*) – (*x2*,*y2*)]]

WRITE Statement

Sends data to the screen

SYNTAX **WRITE** [[*expressionlist*]]

WRITE # Statement

Writes data to a sequential file

SYNTAX **WRITE** #*filenumber*, *expressionlist*

CHAPTER 9

Quick-Reference Tables

9

Each section in this chapter summarizes a group of statements or functions that you typically use together. Each group is presented in a table, which lists the type of task performed (for example, looping or searching), the statement or function name, and the statement action.

The following topics are summarized in tabular form:

- Control-flow statements
- Statements used in BASIC procedures
- Standard I/O statements
- File I/O statements
- String-processing statements and functions
- Graphics statements and functions
- Trapping statements and functions

You can use these tables both as a reference guide to what each statement or function does and as a way to identify related statements.

9.1 Summary of Control-Flow Statements

Table 9.1 lists the BASIC statements used to control the flow of a program's execution.

Table 9.1 Statements Used in Looping and Decision-Making

Task	Statement	Action
Looping	**FOR...NEXT**	Repeats statements between **FOR** and **NEXT** a specific number of times.
	EXIT FOR	Provides an alternative way to exit a **FOR...NEXT** loop.
	DO...LOOP	Repeats statements between **DO** and **LOOP**, either until a given condition is true (**DO...LOOP UNTIL** *condition*), or while a given condition is true (**DO...LOOP WHILE** *condition*).
	EXIT DO	Provides an alternative way to exit a **DO...LOOP** loop.
	WHILE...WEND	Repeats statements between **WHILE** and **WEND** while a given condition is true (similar to **DO WHILE** *condition*...**LOOP**).
Making decisions	**IF...THEN...ELSE**	Conditionally executes or branches to different statements.
	SELECT CASE	Conditionally executes different statements.

9.2 Summary of Statements Used in BASIC Procedures

Table 9.2 lists the statements used in BASIC to define, declare, call, and pass arguments to BASIC procedures. Table 9.2 also lists the statements used to share variables among procedures, modules, and separate programs in a chain.

Table 9.2 Statements Used in Procedures

Task	Statement	Action
Defining a procedure	**FUNCTION... END FUNCTION**	Mark the beginning and end, respectively, of a **FUNCTION** procedure.
	SUB...END SUB	Mark the beginning and end, respectively, of a **SUB** procedure.
Calling a procedure	**CALL**	Transfers control to a BASIC **SUB** procedure, or to a procedure written in another programming language and compiled separately. (The **CALL** keyword is optional.)
Exiting from a procedure	**EXIT FUNCTION**	Provides another way to exit a **FUNCTION** procedure.
	EXIT SUB	Provides an alternative way to exit a **SUB** procedure.
Referencing a procedure before it is defined	**DECLARE**	Declares a **FUNCTION** or **SUB** and, optionally, specifies the number and type of its parameters.
Sharing variables among modules, procedures, or programs	**COMMON**	Shares variables among separate modules. When used with the **SHARED** attribute, it shares variables among different procedures in the same module. Also, passes variable values from current program to new program when control is transferred with the **CHAIN** statement.
	SHARED	When used with the **COMMON**, **DIM**, or **REDIM** statement at the module level (for example, **DIM SHARED**), shares variables with every **SUB** or **FUNCTION** in a single module. When used by itself within a procedure, shares variables between that procedure and the module-level code.

Table 9.2 *(continued)*

Task	Statement	Action
Preserving variable values	**STATIC**	Forces variables to be local to a procedure or **DEF FN** function and preserves the value stored in the variable if the procedure or function is exited, then called again.
Defining a multiline function	**DEF FN...END DEF**	Mark the beginning and end, respectively, of a multiline **DEF FN** function. (This is the old style for functions in BASIC—**FUNCTION** procedures provide a powerful alternative.)
Exiting from a multiline function	**EXIT DEF**	Provides an alternative way to exit a multiline **DEF FN** function.
Calling a BASIC subroutine	**GOSUB**	Transfers control to a specific line in a module. Control is returned from the subroutine to the line following the **GOSUB** statement with a **RETURN** statement. (This is the old style for subroutines in BASIC—**SUB** procedures provide a powerful alternative.)
Transferring to another program	**CHAIN**	Transfers control from current program in memory to another program; use **COMMON** to pass variables to the new program.

9.3 Summary of Standard I/O Statements

Table 9.3 lists the statements and functions used in BASIC for standard I/O (typically, input from the keyboard and output to the screen).

Table 9.3 Statements and Functions Used for Standard I/O

Task	Statement or Function	Action
Printing text on the screen	**PRINT**	Outputs text to the screen. Using **PRINT** with no arguments creates a blank line.
	PRINT USING	Outputs formatted text to the screen.
Changing the width of the output line	**WIDTH**	Changes the width of the screen to either 40 columns or 80 columns and, on computers with an EGA or VGA, controls the number of lines on the screen (25 or 43).
	WIDTH "SCRN:"	Assigns a maximum length to lines output to the screen when used before an **OPEN "SCRN:"** statement.
Getting input from the keyboard	**INKEY$**	Reads a character from the keyboard (or a null string if no character is waiting).
	INPUT$	Reads a specified number of characters from the keyboard and stores them in a single string variable.
	INPUT	Reads input from the keyboard and stores it in a list of variables.
	LINE INPUT	Reads a line of input from the keyboard and stores it in a single string variable.
Positioning the cursor on the screen	**LOCATE**	Moves the cursor to a specified row and column.
	SPC	Skips spaces in printed output.
	TAB	Displays printed output in a given column.
Getting information on cursor location	**CSRLIN**	Tells which row or line position the cursor is in.
	POS(*n*)	Tells which column the cursor is in.
Creating a text viewport	**VIEW PRINT**	Sets the top and bottom rows for displaying text output.

9.4 Summary of File I/O Statements

Table 9.4 lists the statements and functions used in BASIC data-file programming.

Table 9.4 Statements and Functions Used for Data-File I/O

Task	Statement or Function	Action
Creating a new file or accessing an existing file	**OPEN**	Opens a file for retrieving or storing records (I/O).
Closing a file	**CLOSE**	Ends I/O to a file.
Storing data in a file	**PRINT #**	Stores a list of variables as record fields in a previously opened file.*
	PRINT USING #	Similar to **PRINT #**, except **PRINT USING #** formats the record fields.*
	WRITE #	Stores a list of variables as record fields in a previously opened file.*
	WIDTH	Specifies a standard length for each record in a file.*
	PUT	Stores the contents of a user-defined variable in a previously opened file.†
Retrieving data from a file	**INPUT #**	Reads fields from a record and assigns each field in the record to a program variable.*
	INPUT$	Reads a string of characters from a file.
	LINE INPUT #	Reads a record and stores it in a single string variable.*
	GET	Reads data from a file and assigns the data to elements of a user-defined variable.†
Managing files on disk	**FILES**	Prints a listing of the files in a specified directory.
	FREEFILE	Returns the next available file number.
	KILL	Deletes a file from the disk.
	NAME	Changes a file's name.

Table 9.4 (*continued*)

Task	Statement or Function	Action
Getting information about a file	**EOF**	Tests whether all of the data have been read from a file.
	FILEATTR	Returns the number assigned by the operating system to an open file and a number that indicates the mode in which the file was opened (**INPUT**, **OUTPUT**, **APPEND**, **BINARY**, or **RANDOM**).
	LOC	Gives the current position within a file. With binary access, this is the byte position. With sequential access, this is the byte position divided by 128. With random access, this is the record number of the last record read or written.
	LOF	Gives the number of bytes in an open file.
	SEEK (function)	Gives the location where the next I/O operation will take place. With random access, this is the number of the next record to be read or written. With all other kinds of file access, this is the byte position of the next byte to be read or written.
Moving around in a file	**SEEK** (statement)	Sets the byte position for the next read or write operation in an open file.

* For use with sequential files

† For use with binary or random-access files

9.5 Summary of String-Processing Statements and Functions

Table 9.5 lists the statements and functions available in BASIC for working with strings.

Table 9.5 Statements and Functions Used for Processing Strings

Task	Statement or Function	Action
Getting part of a string	**LEFT$**	Returns a given number of characters from the left side of a string.
	RIGHT$	Returns a given number of characters from the right side of a string.
	LTRIM$	Returns a copy of a string with leading blank spaces stripped away.
	RTRIM$	Returns a copy of a string with trailing blank spaces stripped away.
	MID$ (function)	Returns a given number of characters from anywhere in a string.
Searching strings	**INSTR**	Searches for a string within another string.
Converting to upper-case or lowercase letters	**LCASE$**	Returns a copy of a string with all uppercase letters (A–Z) converted to lower-case letters (a–z); leaves lowercase letters and other characters unchanged.
	UCASE$	Returns a copy of a string with all lowercase letters (a–z) converted to upper-case letters (A–Z); leaves uppercase letters and other characters unchanged.
Changing strings	**MID$** (statement)	Replaces part of a string with another string.
	LSET	Left justifies a string within a fixed-length string.
	RSET	Right justifies a string within a fixed-length string.
Converting between numbers and strings	**STR$**	Returns the string representation of the value of a numeric expression.
	VAL	Returns the numeric value of a string expression.

Table 9.5 (*continued*)

Task	Statement or Function	Action
Converting numbers to data-file strings, and data-file strings to numbers*	**CV***type*	Changes numbers stored as strings back to Microsoft Binary format numbers in programs working with random-access files created with older versions of BASIC.
	CV*type***MBF**	Changes numbers stored as Microsoft Binary format strings to IEEE-format numbers.
	MK*type***$**	Changes Microsoft Binary format numbers to strings suitable for storing in random-access files created with older versions of BASIC.
	MK*type***MBF$**	Changes IEEE-format numbers to Microsoft Binary format strings.
Creating strings of repeating characters	**SPACE$**	Returns a string of blank characters of a specified length.
	STRING$	Returns a string consisting of one repeated character.
Getting the length of a string	**LEN**	Tells how many characters are in a string.
Working with ASCII values	**ASC**	Returns the ASCII value of the given character.
	CHR$	Returns the character with the given ASCII value.

* This is no longer necessary for random-access records defined with the data structure **TYPE...END TYPE**.

9.6 Summary of Graphics Statements and Functions

Table 9.6 lists the statements and functions used in BASIC for pixel-based graphics.

Table 9.6 Statements and Functions Used for Graphics Output

Task	Statement or Function	Action
Setting screen-display characteristics	**SCREEN**	Specifies a BASIC screen mode, which determines screen characteristics such as resolution and ranges for color numbers.
Plotting or erasing a single point	**PSET**	Gives a pixel on the screen a specified color using the screen's foreground color by default.
	PRESET	Gives a pixel on the screen a specified color using the screen's background color by default, effectively erasing the pixel.
Drawing shapes	**LINE**	Draws a straight line or a box.
	CIRCLE	Draws a circle or ellipse.
	DRAW	Combines many of the features of other BASIC graphics statements (drawing lines, moving the graphics cursor, scaling images) into an all-in-one graphics macro language.
Defining screen coordinates	**VIEW**	Specifies a rectangle on the screen (or viewport) as the area for graphics output.
	WINDOW	Allows the user to choose new view coordinates for a viewport on the screen.
	PMAP	Maps physical pixel coordinates to view coordinates specified by the user in the current window, or vice versa.
	POINT(*number*)	Returns the current physical or view coordinates of the graphics cursor, depending on the value for *number*.
Using color	**COLOR**	Sets the default colors used in graphics output.
	PALETTE	Assigns different colors to color numbers. Works only on systems equipped with an EGA or VGA.
	POINT(*x*, *y*)	Returns the color number of a single pixel whose screen coordinates are *x* and *y*.

Table 9.6 (*continued*)

Task	Statement or Function	Action
Painting enclosed shapes	**PAINT**	Fills an area on the screen with a color or pattern.
Animating	**GET**	Copies a rectangular area on the screen by translating the image to numeric data and storing the data in a numeric array.
	PUT	Displays an image on the screen that was previously copied with **GET**.
	PCOPY	Copies one screen page to another.

9.7 Summary of Trapping Statements and Functions

Table 9.7 lists the statements and functions used by BASIC to trap and process errors and events.

Table 9.7 Statements and Functions Used in Error and Event Trapping

Task	Statement or Function	Action
Trapping errors while a program is running	**ON ERROR GOTO** *line*	Causes a program to branch to the given *line*, where *line* refers either to a line number or line label. Branching takes place whenever an error occurs during execution.
	RESUME	Returns control to the program after executing an error-handling routine. The program resumes at either the statement causing the error (**RESUME [0]**), the statement after the one causing the error (**RESUME NEXT**), or the line identified by *line* (**RESUME** *line*).
Getting error-status data	**ERR**	Returns the code for an error that occurs at run time.
	ERL	Returns the number of the line on which an error occurred (if program has line numbers).
	ERDEV	Returns a device-specific error code for the last device (such as a printer) for which DOS detected an error.
	ERDEV$	Returns the name of the last device for which DOS detected an error.
Defining your own error codes	**ERROR**	Simulates the occurrence of a BASIC error; can also be used to define an error not trapped by BASIC.
Trapping events while a program is running	**ON** *event* **GOSUB** *line*	Causes a branch to the subroutine starting with *line*, where *line* refers either to a line number or line label, whenever the given *event* occurs during execution.
	event **ON**	Enables trapping of the given *event*.
	event **OFF**	Disables trapping of the given *event*.
	event **STOP**	Suspends trapping of the given *event*.
	RETURN	Returns control to the program after executing an event-handling subroutine. The program resumes at either the statement immediately following the place in the program where the event occurred (**RETURN**), or the line that is identified by *line* (**RETURN** *line*).

Appendixes

A Converting BASICA Programs to QuickBASIC 311

B Differences from Previous Versions of QuickBASIC 315

C Limits in QuickBASIC 337

D Keyboard Scan Codes and ASCII Character Codes 339

E BASIC Reserved Words 345

F Metacommands 347

G Compiling and Linking from DOS 349

H Creating and Using Quick Libraries 377

I Error Messages 391

Appendix A

Converting BASICA Programs to QuickBASIC

QuickBASIC generally accepts BASIC language statements written for the BASIC interpreters used on IBM Personal Computers and compatibles: IBM Advanced Personal Computer BASIC Version A3.00 (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 information on:

- Compatible source-file format
- Statements and functions prohibited or requiring modification for QuickBASIC use
- Editor differences in handling tabs

The following sections describe only the changes required to compile and run a BASICA program with QuickBASIC Version 4.5. The chapters in this book provide information on how to use QuickBASIC features to enhance your existing BASICA programs.

A.1 Source-File Format

QuickBASIC Version 4.5 expects the source file to be in ASCII format or in QuickBASIC's own 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 Statements and Functions Prohibited in 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 a cassette device (not supported by QuickBASIC), or duplicate support provided by the QuickBASIC environment:

AUTO	**LIST**	**NEW**
CONT	**LLIST**	**RENUM**
DEF USR	**LOAD**	**SAVE**
DELETE	**MERGE**	**USR**
EDIT	**MOTOR**	

A.3 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 run your program in QuickBASIC.

Table A.1 Statements Requiring Modification

Statement	Modification
BLOAD/BSAVE	Memory locations may be different in QuickBASIC.
CALL	The argument is the name of the **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.
DRAW, 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 in the QuickBASIC 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.

A.4 Editor Differences in Handling Tabs

QuickBASIC uses individual spaces (rather than the literal tab character, ASCII 9) to represent indentation levels. The Tab Stops option in the Option menu's Display dialog box set the number of spaces per indentation level. (See Chapter 20, "The Options Menu," in *Learning to Use Microsoft QuickBASIC* for more information.)

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 as follows:

1. Literal tab characters inside quoted strings appear as the character shown for ASCII character 9 in the ASCII table in Appendix D, "Keyboard Scan Codes and ASCII Character Codes."
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 such a file and then decide you prefer a different indentation, reset the indentation using the following procedure:

1. Save your file to preserve any changes, then choose the New Program command from the File menu.
2. Choose the Display command from the Options menu and set the Tab Stops option as described above.
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 this procedure works only for old programs. Text created within QuickBASIC cannot be reformatted this way.

Appendix B

Differences from Previous Versions of QuickBASIC

QuickBASIC Version 4.5 contains many new features and enhancements over previous versions of QuickBASIC. This appendix describes the differences between Versions 2.0 and 4.5 of QuickBASIC. Unless otherwise stated, differences between Versions 2.0 and 4.5 also apply as differences between Versions 3.0 and 4.5. Differences between Versions 4.0 and 4.5 are primarily in the QuickBASIC environment.

This appendix provides information on the following improvements in QuickBASIC Version 4.5:

- Product capabilities and features
- Environment enhancements
- Improvements in compiling and debugging capabilities
- Language changes
- File compatibility

B.1 QuickBASIC Features

This section compares the features of Microsoft QuickBASIC Versions 4.5 with those of previous versions. The features are listed in Table B.1 and described below.

Table B.1 Features of Microsoft QuickBASIC Version 4.5

Feature	QuickBASIC Version 2.0	3.0	4.0	4.5
On-line QB Advisor (detailed reference)	No	No	No	Yes
Selectable right mouse button function	No	No	No	Yes
Instant Watches for variables and expressions	No	No	No	Yes
Set default search paths	No	No	No	Yes
User-defined types	No	No	Yes	Yes
IEEE format, math coprocessor support	No	Yes	Yes	Yes
On-line help	No	No	Yes	Yes
Symbol help	No	No	No	Yes
Long (32-bit) integers	No	No	Yes	Yes
Fixed-length strings	No	No	Yes	Yes
Syntax checking on entry	No	No	Yes	Yes
Binary file I/O	No	No	Yes	Yes
FUNCTION procedures	No	No	Yes	Yes
CodeView® support	No	No	Yes	Yes
Compatibility with other languages	No	No	Yes	Yes
Multiple modules in memory	No	No	Yes	Yes
ProKey™, SideKick®, and SuperKey® compatibility	No	Yes	Yes	Yes
Insert/overtype modes	No	Yes	Yes	Yes
WordStar®-style keyboard interface	No	No	Yes	Yes
Recursion	No	No	Yes	Yes
Error listings during separate compilation	No	Yes	Yes	Yes
Assembly-language listings during separate compilation	No	Yes	Yes	Yes

B.1.1 Features New to QuickBASIC 4.5

You can now access on-line help for QuickBASIC's keywords, commands, and menus, as well as general topics and your own variables. Examples shown on help screens can be copied and pasted directly into your own program, reducing development time.

For mouse users, you can now set the function of the right mouse button with the Right Mouse command from the Options menu. Use the function that best suits your needs. For more information see Chapter 19, "The Calls Menu," in *Learning to Use Microsoft QuickBASIC*.

For faster debugging, QuickBASIC now offers an Instant Watch command for immediately identifying the value of a variable or the condition (true or false) of an expression. For more information see Chapter 18, "The Debug Menu," in *Learning to Use Microsoft QuickBASIC*.

Version 4.5 also lets you set default search paths to specific types of files. This lets you organize your files by type and keep them in separate directories. QuickBASIC will search the correct directory after you set the new default search path. You can set default paths for executable, include, library, and help files.

B.1.2 Features Introduced in QuickBASIC 4.0

If you are new to QuickBASIC or unfamiliar with Version 4.0, you will want to review the following features introduced in QuickBASIC 4.0 and supported in the current version.

B.1.2.1 User-Defined Types

The **TYPE** statement allows you to create composite data types from simple data elements. Such data types are similar to C-language structures. User-defined types are discussed in Chapter 3, "File and Device I/O."

B.1.2.2 IEEE Format and Math-Coprocessor Support

Microsoft QuickBASIC provides IEEE-format numbers and math-coprocessor support. When no coprocessor is present, QuickBASIC emulates the coprocessor.

Calculations done by programs compiled with QuickBASIC are generally more accurate and may produce results different from programs run under BASICA or versions of QuickBASIC prior to 4.0. 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 4.0 and 4.5 with your old programs and existing 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.
2. Modify your data files and use the new QuickBASIC 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. More information on these options is provided below.

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 and later versions. With earlier versions, be sure to use the /R command-line option or the 8087 assembler directive.*

B.1.2.3 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 seven 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.

B.1.2.4 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.1.3 Recompiling Old Programs with /MBF

Old programs and files work with QuickBASIC Version 4.5 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 at the DOS prompt:

```
BC multrgrs /MBF;
```

Then link the program as you usually do. To recompile using the Make EXE File command, be sure to use the /MBF option 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.1.4 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 4.5 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 4.5 has the familiar **CVS**, **CVD**, **MKS$**, and **MKD$** functions for reading and writing real numbers from or to random-access files. However, these functions now handle real numbers stored in your files in IEEE format, not Microsoft Binary format. To handle numbers in Microsoft Binary format, Version 4.5 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.

B.1.5 Other QuickBASIC Features

QuickBASIC 4.0 introduced the following features and tools that made QuickBASIC a development package appreciated by the professional without overwhelming the novice. QuickBASIC 4.5 supports all these same features, along with some additional refinements.

B.1.5.1 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.1.5.2 Fixed-Length Strings

Fixed-length strings let you incorporate string data into user-defined types. See Chapter 4, "String Processing," for more information.

B.1.5.3 Syntax Checking on Entry

If syntax checking is turned on, QuickBASIC checks each line as you enter it for syntax and duplicate-definition errors. See Chapter 12, "Using the Editor," in *Learning to Use Microsoft QuickBASIC* for an explanation of syntax checking and other smart-editor features.

B.1.5.4 Binary File I/O

Versions 4.0 and 4.5 provide 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 treat 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 with 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," for more information about accessing binary files.

B.1.5.5 FUNCTION Procedures

FUNCTION procedures allow you to place a function in one module and call it from a different module. See Chapter 2, "SUB and FUNCTION Procedures," for more information about using **FUNCTION** procedures.

In versions prior to 4.0, a **SUB** procedure and a variable could have the same name. Now, **SUB** and **FUNCTION** procedure names must be unique.

B.1.5.6 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 later) and with Microsoft C (Version 5.0 or later). 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 Appendix G, "Compiling and Linking from DOS," for more information.

B.1.5.7 Other-Language Compatibility

QuickBASIC Version 4.5 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 4.5; use the **VARPTR** and **VARSEG** functions instead.

B.1.5.8 Multiple Modules in Memory

You can load multiple program modules into memory simultaneously. Versions previous to 4.0 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 2, "SUB and FUNCTION Procedures," and Chapter 7, "Programming with Modules," for more information about using multiple modules.

B.1.5.9 ProKey™, SideKick®, and SuperKey® Compatibility

You can use ProKey, SideKick, and SuperKey within the QuickBASIC 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 4.5.

B.1.5.10 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.1.5.11 WordStar®-Style Keyboard Commands

QuickBASIC supports many of the key sequences familiar to WordStar users. A complete list of WordStar-style key commands appears in Chapter 12, "Using the Editor," in *Learning to Use Microsoft QuickBASIC*.

B.1.5.12 Recursion

QuickBASIC Versions 4.0 and 4.5 support recursion, which is the ability of a procedure to call itself. Recursion is useful for solving certain problems, such as sorting. See Chapter 2, "SUB and FUNCTION Procedures," for more information about using recursion.

B.1.5.13 Error Listings during Separate Compilation

QuickBASIC displays descriptive error messages when you compile programs using the BC command. Using BC, you can redirect the error messages to a file or device to get a copy of the compilation errors. See Appendix G, "Compiling and Linking from DOS," for more information about the BC command.

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.1.5.14 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 Appendix G, "Compiling and Linking from DOS," for more information.

B.2 Differences in the Environment

The QuickBASIC programming environment now provides more flexible command selection, additional window options, and more menu commands. Sections B.2.1–B.2.5 describes the differences in the programming environment between Version 4.5 and earlier versions.

B.2.1 Choosing Commands and Options

QuickBASIC Version 4.5 gives you flexibility in how you choose commands from menus and options from dialog boxes.

Version 4.5 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 in intense video.

In Version 4.5 the ENTER key functions the same way as the SPACEBAR did in versions 3.0 and earlier. You can press ENTER to execute a command from a dialog box.

B.2.2 Windows

Version 4.5 allows up to two work-area windows (referred to as View windows), a Help window, and a separate Immediate window. Versions prior to 4.0 supported only two windows: one work area and the error-message window.

B.2.3 New Menu

QuickBASIC Version 4.5 has a new menu, the Options menu. The Options menu accesses special QuickBASIC controls that manipulate the screen display attributes, the function of the right mouse button, the default path to specific file types, syntax checking, and Full Menus control.

B.2.4 Menu Commands

The Debug and Help menus, which appeared in earlier versions of QuickBASIC, have some new commands. Table B.2 lists the new commands in these menus, as well as the commands from the new Options menu.

Table B.2 Menus with New Commands in QuickBASIC Version 4.5

Menu	Command	Description
Debug	Add Watch	Adds variable or expression to Watch window
	Instant Watch (New)	Immediately checks the value of a variable or expression
	Watchpoint	Adds a watchpoint to the Watch window
	Delete Watch	Selectively removes item from Watch window
	Delete All Watch	Removes all items from the Watch window
	Trace On	Toggles tracing on and off
	History On	Records last 20 executed statements
	Toggle Breakpoint	Toggles a breakpoint on the current line on and off
	Clear All Breakpoints	Removes all breakpoints
	Break on Errors (New)	Halts program execution if an error occurs, regardless of any error handling
	Set Next Statement	Sets the next statement to execute when a suspended program continues running
Option (New)	Display (New)	Customizes the screen elements
	Set Paths (New)	Alters default search paths for specific types of files
	Right Mouse (New)	Selects the function for the right mouse button
	Syntax Checking	Toggles automatic syntax checking on and off
	Full Menus (New)	Toggles between Full Menus and Easy Menus
Help	Index (New)	Displays an alphabetical list of QuickBASIC keywords and a brief description of each
	Contents (New)	Displays a visual outline of contents
	Topic	Provides information on variables, keywords
	Help on Help (New)	Describes how to use Help and common keyword shortcuts

The Version 3.0 Debug option is removed from the Run menu. In Version 4.5 you can debug at any time, using the debugging commands in the Debug menu.

B.2.5 Editing-Key Changes

The QuickBASIC keyboard interface has been extended to include editing key sequences similar to those in the WordStar editor. (See Chapter 12, "Using the Editor," and Chapter 13, "The Edit Menu," in *Learning to Use Microsoft QuickBASIC* for more information about editing.) The functions performed by the QuickBASIC Version 2.0 key sequences listed in Table B.3 are changed in QuickBASIC Versions 4.0 and 4.5.

Table B.3 Editing-Key Changes

Function	QuickBASIC 2.0 Key	QuickBASIC 4.5 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 4.5 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 Versions 4.0 and 4.5 and previous versions.

B.3.1 Command-Line Differences

QuickBASIC Versions 4.0 and 4.5 support 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 Appendix G, "Compiling and Linking from DOS."

Versions 4.0 and 4.5 do 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 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. Table B.4 describes the way in which QuickBASIC now supports the functionality of these options.

Table B.4 QB and BC Options Not Used in QuickBASIC Versions 4.0 or 4.5

Version 2.0	Version 4.5
On Error (/E)	Automatically set whenever 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 whenever an **ON** *event* statement is present.
Resume Next (/X)	Automatically set whenever a **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 this option, you must compile from the command line with BC.

The options listed in Table B.5 are now available for the QB and BC commands.

Table B.5 Options Introduced in Version 4.0 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 (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

Versions 4.0 and 4.5 do not allow separate compilation with the QB command. Use the BC command described in Appendix G, "Compiling and Linking from DOS" 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 Versions 4.0 and 4.5. You must rebuild the library from the original source files.

User libraries are now 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 Appendix H, "Creating and Using Quick Libraries," for more information on this topic.

B.3.4 Restrictions on Include Files

Include files can contain **SUB** or **FUNCTION** procedure declarations but not definitions. If you need to use an old include file with procedure definitions in it, 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, choose the SUBs command from the View menu, 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 2, "SUB and FUNCTION Procedures," and Chapter 7, "Programming with Modules," for additional information on modules and procedures.

B.3.5 Debugging

QuickBASIC now helps you debug your programs faster by providing the following debugging features:

1. Multiple breakpoints
2. Watch expressions, watchpoints, and instant watches
3. Improved program tracing
4. Full-screen window that displays program text during single-stepping
5. The ability to change variable values during execution, then continue
6. The ability to edit the program, then continue execution without restarting
7. History feature
8. Calls menu
9. Symbol help

The debugging function-key sequences listed in Table B.6 are changed from QuickBASIC Version 2.0:

Table B.6 Debugging-Key Changes

Function	QuickBASIC Version 2.0 Key	QuickBASIC Version 4.5 Key
Trace	ALT+F8	F8
Step	ALT+F9	F10

Note that animation is turned on when you toggle on the Trace On command from the Debug menu, then run your program.

See Chapter 9, "Debugging While You Program," in *Learning to Use Microsoft QuickBASIC* for more information.

B.4 Changes to the BASIC Language

This section describes enhancements and changes to the BASIC language in Version 4.5 and earlier versions of QuickBASIC. Table B.7 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 the table.

Table B.7 Changes to the BASIC Language

Keyword	QuickBASIC Version 2.0	3.0	4.0	4.5
AS	No	No	Yes	Yes
CALL	No	No	Yes	Yes
CASE	No	Yes	Yes	Yes
CLEAR	No	No	Yes	Yes
CLNG	No	No	Yes	Yes
CLS	No	Yes	Yes	Yes
COLOR	No	No	Yes	Yes
CONST	No	Yes	Yes	Yes
CVL	No	No	Yes	Yes
CVSMBF, CVDMBF	No	Yes	Yes	Yes
DECLARE	No	No	Yes	Yes
DEFLNG	No	No	Yes	Yes
DIM	No	No	Yes	Yes
DO...LOOP	No	Yes	Yes	Yes
EXIT	No	Yes	Yes	Yes
FILEATTR	No	No	Yes	Yes
FREEFILE	No	No	Yes	Yes
FUNCTION	No	No	Yes	Yes
GET	No	No	Yes	Yes
LCASE$	No	No	Yes	Yes
LEN	No	No	Yes	Yes
LSET	No	No	Yes	Yes
LTRIM$	No	No	Yes	Yes
MKL$	No	No	Yes	Yes
MKSMBF$, MKDMBF$	No	Yes	Yes	Yes
OPEN	No	No	Yes	Yes
ON UEVENT	No	No	No	Yes
PALETTE	No	No	Yes	Yes
PUT	No	No	Yes	Yes
RTRIM$	No	No	Yes	Yes

Table B.7 *(continued)*

Keyword	QuickBASIC Version 2.0	3.0	4.0	4.5
SCREEN	No	No	Yes	Yes
SEEK	No	No	Yes	Yes
SELECT CASE	No	Yes	Yes	Yes
SETMEM	No	No	Yes	Yes
SLEEP	No	No	No	Yes
STATIC	No	No	Yes	Yes
TYPE	No	No	Yes	Yes
UCASE$	No	No	Yes	Yes
UEVENT	No	No	No	Yes
VARPTR	No	No	Yes	Yes
VARSEG	No	No	Yes	Yes
WIDTH	No	Yes	Yes	Yes

The following section explains in more detail the differences in the keywords summarized above.

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; it 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 that is 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	The **CONST** statement lets 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$** function back to a long integer for use in your BASIC program.
CVSMBF, CVDMBF	The **CVSMBF** and **CVDMBF** functions convert strings containing Microsoft Binary format numbers to IEEE–format numbers. Although QuickBASIC Version 4.5 supports these statements, they are considered obsolete since the IEEE–format is now the QuickBASIC standard.
DECLARE	The **DECLARE** statement allows you to call procedures from different modules, check the number and type of arguments passed, and call procedures before they are defined.
DEFLNG	The **DEFLNG** statement declares all variables, **DEF FN** functions, and **FUNCTION** procedures as having the long-integer type. That is, unless a variable or function has been declared in an **AS** *type* clause, or it has an explicit type-definition suffix such as **%** or **$**, 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	Using **DO...LOOP** statements gives you more powerful loops that allow you to write programs with better structure.
EXIT	The use of **EXIT {DEF | DO | FOR | FUNCTION | SUB}** statements provides 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 4.5:

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. They are obsolete (but supported) in Version 4.5, which uses the IEEE–format.

ON UEVENT

The **ON UEVENT** statement directs the program to a specified location when a user-defined event (a **UEVENT**) occurs. Use it in the same fashion as other event-handling statements.

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," 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," for more information.

SELECT CASE

The use of **SELECT CASE** statements provides 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.

SLEEP

The **SLEEP** statement causes the program to pause for an indicated period of time or until the user presses a key or an enabled event occurs. The optional argument indicates the length of the pause (in seconds).

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.
UEVENT {ON|STOP|OFF}	Enables, suspends, or disables a user-defined event. The **UEVENT** statements are used in the same way as other event-trapping statements.
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.

NOTE *You can no longer conditionally execute* ***NEXT*** *and* ***WEND*** *statements using the single-line* ***IF...THEN...ELSE*** *statement.*

B.5 File Compatibility

All versions of QuickBASIC are source-code compatible; source code created for a previous version compiles under Version 4.5, except as noted in Section B.3.4, "Restrictions on Include Files." QuickBASIC 4.5 translates QuickBASIC 4.0 binary files. If you encounter difficulty translating other binary files, save the program as an ASCII (text) format in QuickBASIC 4.0, then load it with QuickBASIC 4.5. You must recompile object files and user libraries created with previous versions of QuickBASIC.

Appendix C

Limits in QuickBASIC

QuickBASIC and the BC compiler offer programming versatility, but both have limitations in order to keep file size and complexity manageable. As a result, you may reach these limits in some situations. This appendix lists the boundaries you may encounter.

Table C.1 QuickBASIC Limits

Names and Strings	Maximum	Minimum
Variable names	40 characters	1 character
String length	32,767 characters	0 characters
Integers	32,767	–32,768
Long integers	2,147,483,647	–2,147,483,648
Single-precision numbers (positive)	3.402823 E+38	1.401298 E–45
Single-precision numbers (negative)	–1.401298 E–45	–3.402823 E+38
Double-precision numbers (positive)	1.797693134862315 D+308	4.940656458412465 D–324
Double-precision numbers (negative)	–4.940656458412465 D–324	–1.797693134862315 D+308
Arrays		
Array size (all elements)		
Static	65,535 bytes (64K)	1
Dynamic	Available memory	
Array dimensions	8	1
Array subscripts	32,767	–32,768

Table C.1 *(continued)*

Files and Procedures	Maximum	Minimum
Number of arguments passed to a procedure	60 interpreted	0
Nesting of include files	5 levels	0
Procedure size (interpreted)	65,535 bytes (64K)	0
Module size (compiled)	65,535 bytes (64K)	0
Data files open simultaneously	255	0
Data file record number	2,147,483,647	1
Data file record size (bytes)	32,767 bytes (32K)	1 byte
Data file size	Available disk space	0
Path names	127 characters	1 character
Error message numbers	255	1
Editing in the Quick-BASIC Environment		
Text box entry	128 characters	0 characters
"Search for" string	128	1
"Change to" string	40	0
Placemarkers	4	0
Watchpoints and/or watch expressions	8	0
Number of lines in Immediate window	10	0
Characters in View window on one line	255	0
Length of COMMAND$ string	124	0

Appendix D

Keyboard Scan Codes and ASCII Character Codes

D.1 Keyboard Scan Codes

The table on the next page shows the DOS keyboard scan codes. These codes are returned by the **INKEY$** function.

Key combinations with NUL in the Char column return two bytes—a null byte (&H00) followed by the value listed in the Dec and Hex columns. For example, pressing ALT+F1 returns a null byte followed by a byte containing 104 (&H68).

Key	Scan Code		ASCII or Extended†			ASCII or Extended† with SHIFT			ASCII or Extended† with CTRL			ASCII or Extended† with ALT		
	Dec	Hex	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
ESC	1	01	27	1B		27	1B		27	1B				
1 !	2	02	49	31	1	33	21	!				120	78	NUL
2 @	3	03	50	32	2	64	40	@	3	03	NUL	121	79	NUL
3 #	4	04	51	33	3	35	23	#				122	7A	NUL
4 $	5	05	52	34	4	36	24	$				123	7B	NUL
5 %	6	06	53	35	5	37	25	%				124	7C	NUL
6 ^	7	07	54	36	6	94	5E	^	30	1E		125	7D	NUL
7 &	8	08	55	37	7	38	26	&				126	7E	NUL
8 *	9	09	56	38	8	42	2A	*				127	7F	NUL
9 (	10	0A	57	39	9	40	28	(				128	80	NUL
0)	11	0B	48	30	0	41	29	)				129	81	NUL
- _	12	0C	45	2D	-	95	5F	-	31	1F		130	82	NUL
= +	13	0D	61	3D	=	43	2B	+				131	83	NUL
BKSP	14	0E	8	08		8	08		127	7F				
TAB	15	0F	9	09		15	0F	NUL						
Q	16	10	113	71	q	81	51	Q	17	11		16	10	NUL
W	17	11	119	77	w	87	57	W	23	17		17	11	NUL
E	18	12	101	65	e	69	45	E	5	05		18	12	NUL
R	19	13	114	72	r	82	52	R	18	12		19	13	NUL
T	20	14	116	74	t	84	54	T	20	14		20	14	NUL
Y	21	15	121	79	y	89	59	Y	25	19		21	15	NUL
U	22	16	117	75	u	85	55	U	21	15		22	16	NUL
I	23	17	105	69	i	73	49	I	9	09		23	17	NUL
O	24	18	111	6F	o	79	4F	O	15	0F		24	18	NUL
P	25	19	112	70	p	80	50	P	16	10		25	19	NUL
[{	26	1A	91	5B	[	123	7B	{	27	1B				
] }	27	1B	93	5D	]	125	7D	}	29	1D				
ENTER	28	1C	13	0D	CR	13	0D	CR	10	0A	LF			
CTRL	29	1D												
A	30	1E	97	61	a	65	41	A	1	01		30	1E	NUL
S	31	1F	115	73	s	83	53	S	19	13		31	1F	NUL
D	32	20	100	64	d	68	44	D	4	04		32	20	NUL
F	33	21	102	66	f	70	46	F	6	06		33	21	NUL
G	34	22	103	67	g	71	47	G	7	07		34	22	NUL
H	35	23	104	68	h	72	48	H	8	08		35	23	NUL
J	36	24	106	6A	j	74	4A	J	10	0A		36	24	NUL
K	37	25	107	6B	k	75	4B	K	11	0B		37	25	NUL
L	38	26	108	6C	l	76	4C	L	12	0C		38	26	NUL
; :	39	27	59	3B	;	58	3A	:						
' "	40	28	39	27	'	34	22	"						
` ~	41	29	96	60	`	126	7E	~						

† Extended codes return NUL (ASCII 0) as the initial character. This is a signal that a second (extended) code is available in the keystroke buffer.

Key	Scan Code		ASCII or Extended†			ASCII or Extended† with SHIFT			ASCII or Extended† with CTRL			ASCII or Extended† with ALT		
	Dec	Hex	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
L SHIFT	42	2A												
\ |	43	2B	92	5C	\	124	7C	|	28	1C				
Z	44	2C	122	7A	z	90	5A	Z	26	1A		44	2C	NUL
X	45	2D	120	78	x	88	58	X	24	18		45	2D	NUL
C	46	2E	99	63	c	67	43	C	3	03		46	2E	NUL
V	47	2F	118	76	v	86	56	V	22	16		47	2F	NUL
B	48	30	98	62	b	66	42	B	2	02		48	30	NUL
N	49	31	110	6E	n	78	4E	N	14	0E		49	31	NUL
M	50	32	109	6D	m	77	4D	M	13	0D		50	32	NUL
, <	51	33	44	2C	,	60	3C	<						
. >	52	34	46	2E	.	62	3E	>						
/ ?	53	35	47	2F	/	63	3F	?						
R SHIFT	54	36												
* PRTSC	55	37	42	2A	*		INT 5‡		16	10				
ALT	56	38												
SPACE	57	39	32	20	SPC	32	20	SPC	32	20	SPC	32	20	SPC
CAPS	58	3A												
F1	59	3B	59	3B	NUL	84	54	NUL	94	5E	NUL	104	68	NUL
F2	60	3C	60	3C	NUL	85	55	NUL	95	5F	NUL	105	69	NUL
F3	61	3D	61	3D	NUL	86	56	NUL	96	60	NUL	106	6A	NUL
F4	62	3E	62	3E	NUL	87	57	NUL	97	61	NUL	107	6B	NUL
F5	63	3F	63	3F	NUL	88	58	NUL	98	62	NUL	108	6C	NUL
F6	64	40	64	40	NUL	89	59	NUL	99	63	NUL	109	6D	NUL
F7	65	41	65	41	NUL	90	5A	NUL	100	64	NUL	110	6E	NUL
F8	66	42	66	46	NUL	91	5B	NUL	101	65	NUL	111	6F	NUL
F9	67	43	67	43	NUL	92	5C	NUL	102	66	NUL	112	70	NUL
F10	68	44	68	44	NUL	93	5D	NUL	103	67	NUL	113	71	NUL
NUM	69	45												
SCROLL	70	46												
HOME	71	47	71	47	NUL	55	37	7	119	77	NUL			
UP	72	48	72	48	NUL	56	38	8						
PGUP	73	49	73	49	NUL	57	39	9	132	84	NUL			
GREY -	74	4A	45	2D	-	45	2D	-						
LEFT	75	4B	75	4B	NUL	52	34	4	115	73	NUL			
CENTER	76	4C				53	35	5						
RIGHT	77	4D	77	4D	NUL	54	36	6	116	74	NUL			
GREY +	78	4E	43	2B	+	43	2B	+						
END	79	4F	79	4F	NUL	49	31	1	117	75	NUL			
DOWN	80	50	80	50	NUL	50	32	2						
PGDN	81	51	81	51	NUL	51	33	3	118	76	NUL			
INS	82	52	82	52	NUL	48	30	0						
DEL	83	53	83	53	NUL	46	2E	.						

† Extended codes return NUL (ASCII 0) as the initial character. This is a signal that a second (extended) code is available in the keystroke buffer.

‡ Under DOS, SHIFT + PRTSC causes interrupt 5, which prints the screen unless an interrupt handler has been defined to replace the default interrupt 5 handler.

D.2 ASCII Character Codes

Ctrl	Dec	Hex	Char	Code
^@	0	00		NUL
^A	1	01	☺	SOH
^B	2	02	☻	STX
^C	3	03	♥	ETX
^D	4	04	♦	EOT
^E	5	05	♣	ENQ
^F	6	06	♠	ACK
^G	7	07	•	BEL
^H	8	08	◘	BS
^I	9	09	○	HT
^J	10	0A	◙	LF
^K	11	0B	♂	VT
^L	12	0C	♀	FF
^M	13	0D	♪	CR
^N	14	0E	♫	SO
^O	15	0F	☼	SI
^P	16	10	►	DLE
^Q	17	11	◄	DC1
^R	18	12	↕	DC2
^S	19	13	‼	DC3
^T	20	14	¶	DC4
^U	21	15	§	NAK
^V	22	16	▬	SYN
^W	23	17	↨	ETB
^X	24	18	↑	CAN
^Y	25	19	↓	EM
^Z	26	1A	→	SUB
^[	27	1B	←	ESC
^\	28	1C	∟	FS
^]	29	1D	↔	GS
^^	30	1E	▲	RS
^_	31	1F	▼	US

Dec	Hex	Char
32	20	
33	21	!
34	22	"
35	23	#
36	24	$
37	25	%
38	26	&
39	27	'
40	28	(
41	29	)
42	2A	*
43	2B	+
44	2C	,
45	2D	-
46	2E	.
47	2F	/
48	30	0
49	31	1
50	32	2
51	33	3
52	34	4
53	35	5
54	36	6
55	37	7
56	38	8
57	39	9
58	3A	:
59	3B	;
60	3C	<
61	3D	=
62	3E	>
63	3F	?

Dec	Hex	Char
64	40	@
65	41	A
66	42	B
67	43	C
68	44	D
69	45	E
70	46	F
71	47	G
72	48	H
73	49	I
74	4A	J
75	4B	K
76	4C	L
77	4D	M
78	4E	N
79	4F	O
80	50	P
81	51	Q
82	52	R
83	53	S
84	54	T
85	55	U
86	56	V
87	57	W
88	58	X
89	59	Y
90	5A	Z
91	5B	[
92	5C	\
93	5D	]
94	5E	^
95	5F	_

Dec	Hex	Char
96	60	`
97	61	a
98	62	b
99	63	c
100	64	d
101	65	e
102	66	f
103	67	g
104	68	h
105	69	i
106	6A	j
107	6B	k
108	6C	l
109	6D	m
110	6E	n
111	6F	o
112	70	p
113	71	q
114	72	r
115	73	s
116	74	t
117	75	u
118	76	v
119	77	w
120	78	x
121	79	y
122	7A	z
123	7B	{
124	7C	|
125	7D	}
126	7E	~
127	7F	Δ†

† ASCII code 127 has the code DEL. Under DOS, this code has the same effect as ASCII 8 (BS). The DEL code can be generated by the CTRL + BKSP key combination.

Dec	Hex	Char
128	80	Ç
129	81	ü
130	82	é
131	83	â
132	84	ä
133	85	à
134	86	å
135	87	ç
136	88	ê
137	89	ë
138	8A	è
139	8B	ï
140	8C	î
141	8D	ì
142	8E	Ä
143	8F	Å
144	90	É
145	91	æ
146	92	Æ
147	93	ô
148	94	ö
149	95	ò
150	96	û
151	97	ù
152	98	ÿ
153	99	Ö
154	9A	Ü
155	9B	¢
156	9C	£
157	9D	¥
158	9E	₧
159	9F	ƒ

Dec	Hex	Char
160	A0	á
161	A1	í
162	A2	ó
163	A3	ú
164	A4	ñ
165	A5	Ñ
166	A6	ª
167	A7	º
168	A8	¿
169	A9	⌐
170	AA	¬
171	AB	½
172	AC	¼
173	AD	¡
174	AE	«
175	AF	»
176	B0	░
177	B1	▒
178	B2	▓
179	B3	│
180	B4	┤
181	B5	╡
182	B6	╢
183	B7	╖
184	B8	╕
185	B9	╣
186	BA	║
187	BB	╗
188	BC	╝
189	BD	╜
190	BE	╛
191	BF	┐

Dec	Hex	Char
192	C0	└
193	C1	┴
194	C2	┬
195	C3	├
196	C4	─
197	C5	┼
198	C6	╞
199	C7	╟
200	C8	╚
201	C9	╔
202	CA	╩
203	CB	╦
204	CC	╠
205	CD	═
206	CE	╬
207	CF	╧
208	D0	╨
209	D1	╤
210	D2	╥
211	D3	╙
212	D4	╘
213	D5	╒
214	D6	╓
215	D7	╫
216	D8	╪
217	D9	┘
218	DA	┌
219	DB	█
220	DC	▄
221	DD	▌
222	DE	▐
223	DF	▀

Dec	Hex	Char
224	E0	α
225	E1	ß
226	E2	Γ
227	E3	π
228	E4	Σ
229	E5	σ
230	E6	µ
231	E7	τ
232	E8	Φ
233	E9	Θ
234	EA	Ω
235	EB	δ
236	EC	∞
237	ED	φ
238	EE	ε
239	EF	∩
240	F0	≡
241	F1	±
242	F2	≥
243	F3	≤
244	F4	⌠
245	F5	⌡
246	F6	÷
247	F7	≈
248	F8	°
249	F9	∙
250	FA	·
251	FB	√
252	FC	ⁿ
253	FD	²
254	FE	■
255	FF	

Appendix E

BASIC Reserved Words

The following is a list of Microsoft BASIC reserved words:

ABS	CVI	FREEFILE	MID$
ACCESS	CVL	FUNCTION	MKD$
ALIAS	CVS	GET	MKDIR
AND	CVSMBF	GOSUB	MKDMBF$
ANY	DATA	GOTO	MKI$
APPEND	DATE$	HEX$	MKL$
AS	DECLARE	IF	MKS$
ASC	DEF	IMP	MKSMBF$
ATN	DEFDBL	INKEY$	MOD
BASE	DEFINT	INP	NAME
BEEP	DEFLNG	INPUT	NEXT
BINARY	DEFSNG	INPUT$	NOT
BLOAD	DEFSTR	INSTR	OCT$
BSAVE	DIM	INT	OFF
BYVAL	DO	INTEGER	ON
CALL	DOUBLE	IOCTL	OPEN
CALLS	DRAW	IOCTL$	OPTION
CASE	ELSE	IS	OR
CDBL	ELSEIF	KEY	OUT
CDECL	END	KILL	OUTPUT
CHAIN	ENDIF	LBOUND	PAINT
CHDIR	ENVIRON	LCASE$	PALETTE
CHR$	ENVIRON$	LEFT$	PCOPY
CINT	EOF	LEN	PEEK
CIRCLE	EQV	LET	PEN
CLEAR	ERASE	LINE	PLAY
CLNG	ERDEV	LIST	PMAP
CLOSE	ERDEV$	LOC	POINT
CLS	ERL	LOCAL	POKE
COLOR	ERR	LOCATE	POS
COM	ERROR	LOCK	PRESET
COMMAND$	EXIT	LOF	PRINT
COMMON	EXP	LOG	PSET
CONST	FIELD	LONG	PUT
COS	FILEATTR	LOOP	RANDOM
CSNG	FILES	LPOS	RANDOMIZE
CSRLIN	FIX	LPRINT	READ
CVD	FOR	LSET	REDIM
CVDMBF	FRE	LTRIM$	REM

RESET	SHARED	STRING$	UNTIL
RESTORE	SHELL	SUB	USING
RESUME	SIGNAL	SWAP	VAL
RETURN	SIN	SYSTEM	VARPTR
RIGHT$	SINGLE	TAB	VARPTR$
RMDIR	SLEEP	TAN	VARSEG
RND	SOUND	THEN	VIEW
RSET	SPACE$	TIME$	WAIT
RTRIM$	SPC	TIMER	WEND
RUN	SQR	TO	WHILE
SADD	STATIC	TROFF	WIDTH
SCREEN	STEP	TRON	WINDOW
SEEK	STICK	TYPE	WRITE
SEG	STOP	UBOUND	XOR
SELECT	STR$	UCASE$	
SETMEM	STRIG	UEVENT	
SGN	STRING	UNLOCK	

Appendix F

Metacommands

This appendix describes the QuickBASIC metacommands—commands that direct QuickBASIC to handle your program in a particular way. The first section describes the format used for metacommands. The next two sections describe specific metacommands.

By using the metacommands, you can:

- Read in and compile other BASIC source files at specific points during compilation (**$INCLUDE**)
- Control the allocation of dimensioned arrays (**$STATIC** and **$DYNAMIC**)

F.1 Metacommand Syntax

Metacommands begin with a dollar sign ($) and are always enclosed in a program comment. More than one metacommand can be given in one comment. Multiple metacommands are separated by white-space characters (space or tab). Metacommands that take arguments have a colon between the metacommand and the argument:

REM $*METACOMMAND* [[**:** *argument*]]

String arguments must be enclosed in single quotation marks. White-space characters between elements of a metacommand are ignored. The following are all valid forms for metacommands:

```
REM $STATIC $INCLUDE:  'datadefs.bi'
REM     $STATIC    $INCLUDE :  'datadefs.bi'
' $STATIC $INCLUDE: 'datadefs.bi'
'     $STATIC    $INCLUDE :  'datadefs.bi'
```

Note that no spaces appear between the dollar sign and the rest of the metacommand.

If you want to refer to a metacommand in a description but do not want it to execute, place a character that is not a tab or space before the first dollar sign on the line. For example, on the following line both metacommands are ignored:

```
REM x$STATIC $INCLUDE:  'datadefs.bi'
```

F.2 Processing Additional Source Files: $INCLUDE

The **$INCLUDE** metacommand instructs the compiler to temporarily switch from processing one file and instead to read program statements from the

BASIC file named in the argument. When the end of the included file is reached, the compiler returns to processing the original file. Because compilation begins with the line immediately following the line in which **$INCLUDE** occurred, **$INCLUDE** should be the last statement on a line. The following statement is correct:

```
DEFINT I-N    ' $INCLUDE: 'COMMON.BAS'
```

There are two restrictions on using include files:

1. Included files must not contain **SUB** or **FUNCTION** statements.
2. Included files created with BASICA must be saved with the ,A option.

F.3 Dimensioned Array Allocation: $STATIC and $DYNAMIC

The **$STATIC** and **$DYNAMIC** metacommands tell the compiler how to allocate memory for arrays. Neither of these metacommands takes an argument:

```
'Make all arrays dynamic.

'$DYNAMIC
```

$STATIC sets aside storage for arrays during compilation. When the **$STATIC** metacommand is used, the **ERASE** statement reinitializes all array values to zero (numeric arrays) or the null string (string arrays) but does not remove the array. The **REDIM** statement has no effect on **$STATIC** arrays.

$DYNAMIC allocates storage for arrays while the program is running. This means that the **ERASE** statement removes the array and frees the memory it took for other uses. You can also use the **REDIM** statement to change the size of a **$DYNAMIC** array.

The **$STATIC** and **$DYNAMIC** metacommands affect all arrays except implicitly dimensioned arrays (arrays not declared in a **DIM** statement). Implicitly dimensioned arrays are always allocated as if **$STATIC** had been used.

Appendix G

Compiling and Linking from DOS

This appendix explains how to compile and link outside the QuickBASIC environment. You might want to do this for some of the following reasons:

- 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

When you finish this appendix 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

G.1 BC, LINK, and LIB

The Microsoft QuickBASIC package includes BC, LINK, and LIB. The following list describes how these special tools are used when compiling and linking outside of the QuickBASIC environment:

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. These object files will be linked together to form your program or Quick library. BC is also available any time you want to compile programs outside of the QuickBASIC environment. You may prefer to use BC if you 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 Microsoft 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. You can use LINK directly whenever you want to link object files or make Quick libraries.
LIB.EXE	The Microsoft Library Manager (LIB) creates 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.

G.2 The Compiling and Linking Process

To create a stand-alone program from a BASIC source file when you are outside of the QuickBASIC environment, follow these steps:

1. Compile each source file, creating an object file.
2. Link the object files using LINK. LINK includes one or more stand-alone libraries and creates an executable file. LINK 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 following methods of compiling and linking:

- Compile and link in separate steps by using the BC and LINK commands.
- 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 to minimize the size of the executable file and maximize program-loading speed.*

When compiling and linking from DOS, the paths you defined in the Options menu are not used. To search for include and library files the way you specified on the Options menu, you must set the DOS environment variables LIB and INCLUDE to point to the appropriate directories. Otherwise the compiler and/or linker might generate `File not found` errors.

Sections G.3 and G.4 explain how to compile and link in separate steps.

G.3 Compiling with the BC Command

You can compile with the BC command in either of the following ways:

- Type all information on a single command line, using the following syntax:

  BC *sourcefile* [[,[[*objectfile*]] [[,[[*listingfile*]]]] [[*optionslist*]][[;]]

- Type

  ```
  BC
  ```

  and respond to the following prompts:

  ```
  Source Filename [.BAS]:
  Object Filename [basename.OBJ]:
  Source Listing: [NUL.LST]:
  ```

Table G.1 shows the input you must give on the BC command line or in response to each prompt:

Table G.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 G.3.2, "Using BC Command Options"

G.3.1 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 for specifying file names to the BC command.

G.3.1.1 Uppercase and Lowercase Letters

You can use any combination of uppercase and lowercase letters for file names; the compiler accepts uppercase and lowercase letters interchangeably.

Example

The BC command considers the following three file names to be equivalent:

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

G.3.1.2 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 described in the following list:

Extension	File Description
.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

G.3.1.3 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 lets you 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.*

G.3.2 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. (The forward slash and the dash can be used interchangeably. 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 for each communications port when using an asynchronous communications adapter. (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 512 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 QuickBASIC environment.
/E	Indicates presence of **ON ERROR** with **RESUME** *linenumber* statements. (See also the discussion of the /X option in this list.)
/MBF	The intrinsic functions **MKS$**, **MKD$**, **CVS**, and **CVD** are converted to **MKSMBF$**, **MKDMBF$**, **CVSMBF**, and **CVDMBF**, respectively. This allows your QuickBASIC program to read and write floating-point values stored in Microsoft Binary format.
/O	Substitutes the BCOM45.LIB run-time library for BRUN45.LIB. See Chapter 16, "The Run Menu," in *Learning to Use Microsoft 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 at each line number or label 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 of debugging information used by the Microsoft CodeView debugger, available with Microsoft C, Version 5.0 and later and Microsoft Macro Assembler, Version 5.0 and later.

G.4 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 "response file" (a 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 G.2 shows the input you must give on the LINK command line, or in response to each prompt.

Table G.2 Input to the LINK Command

Field	Prompt	Input
objfile	"Object Modules"	One or more object files that you are linking. The object files should be separated by either 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 G.4.6.11 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 G.4.3 for rules for specifying library names to the linker.
linkopts	Gives options after any response	Any of the LINK options described in Sections G.4.6.2–G.4.6.15. You can specify LINK options anywhere on the command line.

* Another way to create a map file is to specify the /MAP option to the LINK command (see Section G.4.6.11).

If you are using a response file, each response must follow the rules outlined in Table G.2.

G.4.1 Defaults for LINK

You can choose defaults for any of the information that LINK needs in any of 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 *mapfile* entry: if you use a comma as a placeholder for this entry, LINK creates a map file.
- To choose the default for any prompt, press ENTER.
- 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 LINK 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, LINK automatically appends the .EXE extension.
Map	The special file name NUL.MAP, which tells LINK *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, LINK automatically appends the .MAP extension.
Libraries	Libraries named in the given object files. If you choose the Stand-Alone EXE File option, BCOM45.LIB is the default library; otherwise, the default is BRUN45.LIB. If you specify a library other than a default library, you must give only the base name; if you give a library name with no extension, LINK automatically appends the .LIB extension. See Section G.4.3 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 LINK to link together the four object modules FRAME, TEXT, TABLE, and LINEOUT. The executable file FRAME.EXE and the map file named FRAMESYM.MAP are produced. The /PAUSE option causes LINK to pause before producing the executable file to permit disk swapping, if necessary. The /MAP option tells LINK to include public symbols and addresses in the map file. LINK also links any needed routines from the library file GRAF.LIB. See Sections G.4.6.2 and G.4.6.11 for more information on /PAUSE and /MAP options.

```
FRAME TEXT TABLE LINEOUT
/PAUSE /MAP
FRAMESYM
GRAF.LIB
```

In the following example, LINK 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 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, which is 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]: ;
```

G.4.2 Specifying Files to LINK

The rules for specifying file names to the linker are the same as 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 LINK to look for files or create files in the given path. See Section G.3.1 for more information.

G.4.3 Specifying Libraries to LINK

Ordinarily, you do not need to give LINK 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, LINK 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 LINK. You can give the library name in either 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.

LINK 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

- Link with additional stand-alone libraries.
- Link with libraries in different paths. If you specify a complete path name for the library, LINK 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 LINK 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 LINK option /NOD in addition to specifying the library you want to use for linking. See Section G.4.6.8 for more information about the /NOD option.

G.4.4 Memory Requirements for LINK

LINK uses available memory for the linking session. If the files to be linked create an output file that exceeds available memory, LINK creates a temporary disk file to serve as memory. This temporary file is handled in one of the following ways, depending on the DOS version:

- LINK uses the directory specified by the TMP environment variable from DOS for the purpose of creating a temporary file. For example, if the TMP variable was set to C:\TEMPDIR, then LINK would put the temporary file in C:\TEMPDIR.

  If there is no TMP environment variable, or if the directory specified by TMP does not exist, then LINK puts the temporary file in the current working directory.

- If LINK 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 LINK 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 LINK creates is a working file only. LINK deletes it at the end of the session.

NOTE *Do not give any of your own files the name VM.TMP. LINK displays an error message if it finds an existing file with this name.*

G.4.5 Linking with Mixed-Language Programs

You can link mixed-language programs with LINK. However, problems can result from linking .OBJ files from within the other language. Different assumptions by different linkers can corrupt QuickBASIC files.

The following sections discuss linking with modules written in Pascal, FORTRAN, and assembly language.

G.4.5.1 Pascal and FORTRAN Modules in QuickBASIC Programs

Modules compiled with Microsoft Pascal or FORTRAN can be linked with BASIC programs, as described in the *Microsoft Mixed-Language Programming Guide*. They can also be incorporated in Quick libraries. However, QuickBASIC programs containing code compiled with Microsoft Pascal must allocate at least 2K near-heap space for Pascal. The following example does this by using the **DIM** statement to allocate a static array of 2K or greater in a named common block called NMALLOC:

```
DIM name%(2048) : COMMON SHARED /NMALLOC/ name%()
```

The Pascal run-time module assumes it always has at least 2K of near-heap space available. If the Pascal code cannot allocate the required space, QuickBASIC may crash. This applies to Pascal code in Quick libraries and to Pascal code linked into executable files. The situation is similar for FORTRAN I/O, which also requires near buffer space, and which can be provided by using an NMALLOC common block.

G.4.5.2 STATIC Array Allocation in Assembly-Language Routines

Use the **SEG** or **CALLS** keywords or far pointers to pass static array data to assembly-language routines. You cannot assume data is in a particular segment. Alternatively, you can declare all arrays dynamic (still using far pointers) since BC and the QuickBASIC environment handle dynamic arrays identically.

G.4.5.3 References to DGROUP in Extended Run-Time Modules

For mixed-language programs that use the **CHAIN** command, you should make sure that any code built into an extended run-time module does not contain any references to DGROUP. (The **CHAIN** command causes DGROUP to move, but does not update references to DGROUP.) This rule applies only to mixed-language programs; because BASIC routines never refer to DGROUP, you can ignore this caution for programs written entirely in BASIC.

To avoid this problem, you can use the value of SS, since BASIC always assumes that SS coincides with DGROUP.

G.4.6 Using LINK Options

LINK options begin with the linker's option character, which is the forward slash (/). Case is not significant in LINK options; for example, /NOI and /noi are equivalent.

You can abbreviate LINK 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 LINK 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 LINK 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 `010` is an octal number, equivalent to 8 in decimal.
- A hexadecimal number from 0 to 0xFFFF. A number is interpreted as hexadecimal if it starts with a zero followed by an x or an X. For example, `0x10` is a hexadecimal number, equivalent to 16 in decimal.

LINK options affect all files in the linking process, regardless of where the options are specified.

If you usually use the same set of LINK options, you can use the LINK environment variable in DOS to specify certain options each time you link. If you set this variable, LINK checks it for options and expects to find options listed exactly as you would type them on the command line. You cannot specify filename 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;
```

G.4.6.1 Viewing the Options List (/HE)

/HE[LP]

The /HE option tells LINK to display a list of the available LINK options on the screen.

G.4.6.2 Pausing during Linking (/PAU)

/PAU[SE]

The /PAU option tells LINK 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, LINK 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. LINK 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, 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.*

G.4.6.3 Displaying Linker Process Information (/I)

/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.

G.4.6.4 Preventing Linker Prompting (/B)

/B[ATCH]

The /B option tells LINK 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 to let you use batch or MAKE files to link many executable files with a single command if you do not want LINK 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 LINK from prompting for arguments missing from the LINK command line.

G.4.6.5 Creating Quick Libraries (/Q)

/Q[[UICKLIB]]

The /Q option tells LINK 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 BQLB45.LIB in the library list in order to include QuickBASIC Quick-library support routines.

See Appendix H, "Creating and Using Quick Libraries," for more information about creating and loading Quick libraries.

NOTE *You cannot use the /EXEPACK option with the /Q option.*

G.4.6.6 Packing Executable Files (/E)

/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.

NOTE *Executable files linked with this option may be smaller and load faster than files linked without this option.*

G.4.6.7 Disabling Segment Packing (/NOP)

/NOP[[ACKCODE]]

The /NOP option is normally not necessary because code-segment packing is normally turned off. However, if a DOS environment variable such as LINK turns on code-segment packing automatically, you can use the /NOP option to turn segment packing back off again.

G.4.6.8 Ignoring the Usual BASIC Libraries (/NOD)

/NOD[[EFAULTLIBRARYSEARCH]]

When it creates an object file, BC includes the names of the "standard" libraries—libraries that LINK searches to resolve external references. The /NOD option tells LINK *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 (BRUN45.LIB and BCOM45.LIB). Thus, if you use the /NOD option, you should explicitly give the path name of the required standard library.

G.4.6.9 Ignoring Dictionaries (/NOE)

/NOE[[XTDICTIONARY]]

If LINK suspects that a public symbol has been redefined, it prompts you to link again with the /NOE option. When you do so, it searches the individual object files, rather than "dictionaries" it has created, to resolve conflicts. For example, when linking a program with 87.LIB or NOCOM.OBJ, you must use the /NOE option.

G.4.6.10 Setting Maximum Number of Segments (/SE)

/SE[[GMENTS]]:*number*

The /SE option controls the number of segments that LINK 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, LINK must allocate space to keep track of segment information. When you set the segment limit higher than 128, LINK allocates more space for segment information. For programs with fewer than 128 segments, you can minimize the amount of storage LINK needs by setting *number* to reflect the actual number of segments in the program. LINK displays an error message if this number is too high for the amount of memory it has available.

G.4.6.11 Creating a Map File (/M)

/M[[AP]]

The /M option creates a map file. A map file lists the segments of a program and the program's public symbols. LINK always tries to allocate all of the available memory for sorting public symbols. If the number of symbols exceeds the memory limit, then LINK generates an unsorted list. The map file *mapfile* contains a list of symbols sorted by address; however, it does not contain a list sorted by name. A 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 `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 `Length` gives the length

of the segment in bytes. The column `Name` gives the name of the segment; the column `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       $TKHQQ
 0000:1D86       B$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.

G.4.6.12 Including Line Numbers in a Map File (/LI)

/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.

G.4.6.13 Packing Contiguous Segments (/PAC)

/[[NO]]PAC[[KCODE]][[:*number*]]

The /PAC option tells LINK 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. LINK stops adding segments to a particular group as soon as it cannot add a segment to the group without exceeding *number*. At that point, LINK starts forming a new group with the remaining code segments. If *number* is not given, the default is 65,530.

Although LINK 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 in DOS.

G.4.6.14 Using the CodeView Debugger (/CO)

/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.

G.4.6.15 Distinguishing Case (/NOI)

/NOI[[GNORECASE]]

The /NOI option tells LINK to distinguish between uppercase and lowercase letters; for example, LINK 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.

G.4.7 Other LINK Command-Line Options

Not all options of the LINK command are suitable for use with QuickBASIC programs. The following LINK 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:

Option	Action
/CP[[ARMAXALLOC]]:*number*	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.

/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 LINK 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.

NOTE *Do not use the /DS, /HI, /NOG, or /O options when linking object files compiled with BC. They are suitable only for object files created by the Microsoft Macro Assembler (MASM).*

G.5 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 combined to form a library. Unlike an object file, an object module does not exist independently of its library, and 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.

G.5.1 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 has a maximum length of 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 response file, a file with responses to LIB command prompts, 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 LINK prompt, or at any position in the LIB command line.

Table G.3 shows the input you must give on the LIB command line, or in response to each prompt. If you are using a response file, each response must follow the rules outlined in this table.

Table G.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.
/I	None	Tells LIB to ignore case when comparing symbols (default). Use when combining with libraries that are case sensitive.
/NOE	None	Tells LIB not to generate an extended dictionary.
/NOI	None	Tells LIB to compare case when comparing symbols (LIB remains case sensitive).
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.

G.5.2 Usual Responses for LIB

LIB has its own built-in (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 omit this entry, LIB creates a cross-reference-listing file.
- To choose the default for any prompt, 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

G.5.3 Cross-Reference-Listing Files

A cross-reference-listing file tracks which routines are contained in a stand-alone library and the original object files they came from. 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 defined.
- A list of the modules in the library. Under each module name is an alphabetical listing of the public symbols defined in that module.

G.5.4 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`. LIB deletes `HEAP` 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 the cross-reference-listing file 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 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 (without deleting) 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 the cross-reference-listing file CROSSLST.

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

G.5.5 LIB Options

LIB has four options. Specify options on the command line following the required library-file name and preceding any commands.

G.5.5.1 Ignoring Case for Symbols

/I[GNORECASE]]

The /I option tells LIB to ignore case when comparing symbols, as LIB does by default. Use this option when you are combining a library that is marked /NOI (described below) with others that are unmarked and you want the new library to be unmarked.

G.5.5.2 Ignoring Extended Dictionaries

/NOE[[XTDICTIONARY]]

The /NOE option tells LIB not to generate an extended dictionary. The extended dictionary is an extra part of the library that helps the linker process libraries faster.

Use the /NOE option if you get errors `U1171` or `U1172`, or if the extended dictionary causes problems with LINK. See Section G.4.6.9 for more information on how LINK uses the extended dictionary.

G.5.5.3 Distinguishing Case for Symbols

/NOI[[GNORECASE]]

The /NOI option tells LIB not to ignore case when comparing symbols; that is, /NOI makes LIB case sensitive. By default, LIB ignores case. Using this option allows symbols that are the same except for case, such as SPLINE and Spline, to be put in the same library.

Note that when you create a library with the /NOI option, LIB marks the library internally to indicate that /NOI is in effect. Earlier version of LIB did not mark libraries in this way. If you combine multiple libraries, and any one of them is marked /NOI, then /NOI is assumed to be in effect for the output library.

G.5.5.4 Setting Page Size

/P[AGESIZE]]:*number*

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:

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 the 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).

Appendix H

Creating and Using Quick Libraries

This appendix 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 appendix 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

H.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 of frequently used procedures to your programs. A Quick library can contain procedures written in QuickBASIC or other Microsoft languages such as Microsoft C.
.LIB	The .LIB extension characterizes a stand-alone (.LIB) library, one 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 appended to QuickBASIC when the library is loaded with QuickBASIC. Libraries with the .LIB extension are essentially independent, compiled 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 appendix discusses the use of command-line utilities for some common cases, but you should refer to Appendix G, "Compiling and Linking from DOS," for a full explanation of using those utilities.

H.2 Advantages of Quick Libraries

Quick libraries facilitate program development and maintenance. 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 a Quick library can even be invoked without a **CALL** statement. See Chapter 2, "SUB and FUNCTION Procedures," for more information on calling a **SUB** procedure with or without the **CALL** keyword.

Procedures in a Quick library 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 codevelop programs with others, Quick libraries make it easy to update a pool of common procedures. 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.*

H.3 Creating a Quick Library

A Quick library automatically contains all modules, both main and nonmain, present in the QuickBASIC environment when you create the new library. It also contains the contents of any other Quick library that 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 which procedures a loaded library contains. The QLBDUMP.BAS utility program, described in Section H.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. A procedure in a Quick library can only call other procedures within the same Quick library. Procedure names must be unique within the 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 are likely to be less cluttered with useless code.*

H.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.
BQLB45.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.

H.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 your Quick library. In these cases, 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.

H.3.3 Making a Quick Library from within the Environment

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.

H.3.3.1 Unloading Unwanted Files

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.

H.3.3.2 Loading Desired Files

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.

H.3.3.3 Creating a Quick Library

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 from the Run menu. The dialog box shown in Figure H.1 appears.

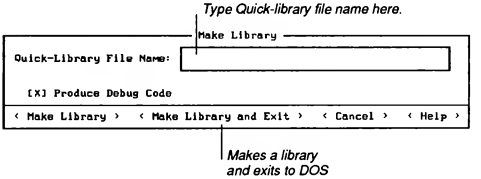


Figure H.1 Make Library Dialog Box

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 extension you like (except the name of a loaded Quick library), 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, slows program execution and gives only a small amount of error control, mostly in regard to 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.*

H.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.

H.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 [[*programname*]] /L [[*libraryname*]]

If you start QuickBASIC with the /L option and supply the name of a library (*libraryname*), 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 H.5).

You can also start QuickBASIC with the /RUN option followed by both a program name (*programname*) 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. If you specify a path, QuickBASIC looks where you indicate; otherwise, QuickBASIC searches for the Quick library in the following three locations:

1. The current directory.
2. The path specified for libraries by the Set Paths command.
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 `REPORT.BAS` using the routines in the library `FIGS.QLB`:

```
QB /RUN REPORT.BAS /L FIGS.QLB
```

H.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. For more information, see Chapter 16, "The Run Menu," in *Learning to Use Microsoft QuickBASIC*.

H.4.3 Viewing the Contents of a Quick Library

Because a Quick library is essentially a binary file, you cannot 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," for a commented listing of QLBDUMP.BAS.

H.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, included with the QuickBASIC package. This file contains three routines, INTERRUPT, INT86OLD, and ABSOLUTE, that 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 that 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.

H.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 *libraryname* option, QuickBASIC assumes that the listed *libraryname* has the .QLB extension if no other extension is specified. If your Quick library has no extension, you must put a period after the Quick-library name (*libraryname.*) or QuickBASIC searches for a file with your base name and the .QLB extension.

H.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 (.QLB) 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 LINK, the library BQLB45.LIB must always be specified after the third comma on the LINK command line or in response to the "Libraries" prompt.

H.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.

H.8.1 Building a Quick Library

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, which 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,,BQLB45.LIB;
   ```

   The linker interprets the entry that follows the names of the object files (in this case MIXED.QLB) as the file 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 .LIB library:

   ```
   LIB MIXED.LIB+FOR.OBJ+C.OBJ+ASM.OBJ;
   ```

   It is easy to overlook this step 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 Appendix G, "Compiling and Linking from DOS," for a complete description of the features of LINK and LIB.

H.8.2 Quick Libraries with Leading Zeros in the First Code Segment

A Quick library containing leading zeros in the first code segment is invalid, causing the message `Error in loading file` *filename* `- Invalid format` when you try to load it in QuickBASIC. For example, this can occur if an assembly-language routine puts data that is initialized to zero in the first code segment and it is subsequently listed first on the LINK command line when you

make a Quick library. If you have this problem, do one of the following two things:

1. Link with a BASIC module first on the LINK command line.
2. Make sure that, in whatever module comes first on the LINK command line, the first code segment starts with a nonzero byte.

H.8.3 The B_OnExit Routine

QuickBASIC provides a BASIC system-level function, B_OnExit. You can use B_OnExit when your other-language routines take special actions that need to be undone before leaving the program (intentionally or otherwise) or rerunning 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 crash. 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 any other-language (including assembly-language) routines you place in a Quick library. With programs compiled and linked completely from the command line, B_OnExit is optional.

H.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. This 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; such a program may require somewhat more memory.

H.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 library in which each module contains only closely related procedures. If you have any doubts about what a library contains, list its contents with the utility QLBDUMP.BAS, described in Section H.4.3, "Viewing the Contents of a Quick Library."

Appendix I

Error Messages

During development of a BASIC program with QuickBASIC, the following types of errors can occur:

- Invocation errors
- Compile-time errors
- Link-time errors
- Run-time errors

Each type of error is associated with a particular step in the program development process. Invocation errors occur when you invoke QuickBASIC with the QB or BC commands. Compile-time errors (and warnings) occur during compilation, and run-time errors occur when the program is executing. Link-time errors occur only when you use the LINK command to link object files created with BC or other language compilers.

Section I.2 lists alphabetically the invocation, compile-time, and run-time error messages, along with any error codes that are assigned. Table I.1 lists the run-time error messages and error codes in numerical order. Section I.3 lists the Microsoft Overlay Linker error messages, and Section I.4, the Microsoft Library Manager error messages.

I.1 Error-Message Display

When a run-time error occurs within the QuickBASIC environment (with default screen options), the error message appears in a dialog box and the cursor is placed on the line where the error occurred.

In stand-alone executable programs (that is, programs that are executed by entering the base name of the executable file at the system prompt), the run-time system prints the error messages followed by an address, unless the /D, /E, or /W option is specified on the BC command line. In those cases, the error message is followed by the number of the line in which the error occurred. The standard forms of this type of error message are as follows:

```
Error n in module module-name at address segment:offset
```

and

```
Error n in line linenumber of module module-name at
address segment:offset
```

An **ERR** code is listed for some errors. If an error occurs, the value returned by **ERR** is set to the appropriate code when an error-trapping subroutine is entered. (Error-trapping routines are entered via the **ON ERROR** statement.) The **ERR** value remains unchanged until a **RESUME** statement returns control to the main program. See Chapter 6, "Error and Event Trapping," for more information.

Table I.1 lists the error codes in numerical order. See the alphabetical listing for explanations of the errors.

Table I.1 Run-Time Error Codes

Code	Description	Code	Description
2	Syntax error	53	File not found
3	RETURN without GOSUB	54	Bad file mode
4	Out of DATA	55	File already open
5	Illegal function call	56	FIELD statement active
6	Overflow	57	Device I/O error
7	Out of memory	58	File already exists
9	Subscript out of range	59	Bad record length
10	Duplicate definition	61	Disk full
11	Division by zero	62	Input past end of file
13	Type mismatch	63	Bad record number
14	Out of string space	64	Bad file name
16	String formula too complex	67	Too many files
19	No RESUME	68	Device unavailable
20	RESUME without error	69	Communication-buffer overflow
24	Device timeout	70	Permission denied
25	Device fault	71	Disk not ready
27	Out of paper	72	Disk-media error
39	CASE ELSE expected	73	Advanced feature unavailable
40	Variable required	74	Rename across disks
50	FIELD overflow	75	Path/File access error
51	Internal error	76	Path not found
52	Bad file name or number		

I.2 Invocation, Compile-Time, and Run-Time Error Messages

Advanced feature unavailable

You are attempting to use a feature of QuickBASIC that is available with another version of BASIC, or supported only under a later version of DOS. (Compile-time or run-time error)

ERR code: 73

Argument-count mismatch

You are using an incorrect number of arguments with a BASIC subprogram or function. (Compile-time error)

Array already dimensioned

This error can be caused by any of the following:

- More than one **DIM** statement for the same static array.
- A **DIM** statement after the initial use of an array. Dynamic arrays must be deallocated with the **ERASE** statement before they can be redimensioned with **DIM**; dynamic arrays may also be redimensioned with the **REDIM** statement.
- An **OPTION BASE** statement that occurs after an array is dimensioned.

(Compile-time or run-time error)

Array not defined

An array is referenced but never defined. (Compile-time error)

Array not dimensioned

An array is referenced but not dimensioned. If you are compiling the program with BC, this error is not "fatal"; the program will execute, although program results may be incorrect. (Compile-time warning)

Array too big

There is not enough user data space to accommodate the array declaration. Reduce the size of the array or use the **$DYNAMIC** metacommand. You may also get this error if the array size exceeds 64K, the array is not dynamic, and the /AH option is not used. Reduce the size of the array, or make the array dynamic and use the /AH command-line option. (Compile-time error)

Cannot continue

While debugging, you have made a change that prevents execution from continuing. (Run-time error)

Cannot find file (*filename*). Input path:

This error occurs when QuickBASIC cannot find a Quick library or utility (BC.EXE, LINK.EXE, LIB.EXE, or QB.EXE) required by the program. Enter the correct path name, or press CTRL+C to return to the DOS prompt. (QB invocation error)

Cannot generate listing for BASIC binary source files

You are attempting to compile a binary source file with the BC command and the /A option. Recompile without the /A option. (BC invocation error)

Cannot start with 'FN'

You used "FN" as the first two letters of a subprogram or variable name. "FN" can only be used as the first two letters when calling a **DEF FN** function. (Compile-time error)

CASE ELSE expected

No matching case was found for an expression in a **SELECT CASE** statement. (Run-time error)

ERR code: 39

CASE without SELECT

The first part of a **SELECT CASE** statement is missing or misspelled. (Compile-time error)

Colon expected after /C

A colon is required between the option and the buffer size argument. (BC invocation error)

Comma missing

QuickBASIC expects a comma. (Compile-time error)

COMMON and DECLARE must precede executable statements

A **COMMON** statement or a **DECLARE** statement is misplaced. **COMMON** and **DECLARE** statements must appear before any executable statements. All BASIC statements are executable except the following:

- **COMMON**
- **DEF***type*
- **DIM** (for static arrays)
- **OPTION BASE**
- **REM**
- **TYPE**
- All metacommands

(Compile-time error)

COMMON in Quick library too small

More common variables are specified in the module than in the currently loaded Quick library. (Compile-time error)

COMMON name illegal

QuickBASIC encountered an illegal */blockname/* specification (for example, a *blockname* that is a BASIC reserved word) in a named **COMMON** block. (Compile-time error).

Communication-buffer overflow

During remote communications, the receive buffer overflowed. The size of the receive buffer is set by the /C command line option or the **RB** option in the **OPEN COM** statement. Try checking the buffer more frequently (with the **LOC** function) or emptying it more often (with the **INPUT$** function). (Run-time error)

ERR code: 69

CONST/DIM SHARED follows SUB/FUNCTION

CONST and **DIM SHARED** statements should appear before any subprogram or **FUNCTION** procedure definitions. If you are compiling your program with BC, this error is not "fatal"; the program will execute, although the results may be incorrect. (Compile-time warning)

Control structure in IF...THEN...ELSE incomplete

An unmatched **NEXT, WEND, END IF, END SELECT,** or **LOOP** statement appears in a single-line **IF...THEN...ELSE** statement. (Compile-time error)

Data-memory overflow

There is too much program data to fit in memory. This error is often caused by too many constants, or too much static array data. If you are using the BC command, or the Make EXE File or Make Library commands, try turning off any debugging options. If memory is still exhausted, break your program into parts and use the **CHAIN** statement or use the **$DYNAMIC** metacommand. (Compile-time error)

DECLARE required

An implicit **SUB** or **FUNCTION** procedure call appears before the procedure definition. (An implicit call does not use the **CALL** statement.) All procedures must be defined or declared before they are implicitly called. (Compile-time error)

DEF FN not allowed in control statements

DEF FN function definitions are not permitted inside control constructs such as **IF...THEN...ELSE** and **SELECT CASE**. (Compile-time error)

DEF without END DEF

There is no corresponding **END DEF** in a multiline function definition. (Compile-time error)

DEF*type* character specification illegal

A **DEF***type* statement is entered incorrectly. **DEF** can only be followed by **LNG, DBL, INT, SNG, STR,** or (for user-defined functions) a blank space. (Compile-time error)

Device fault

A device has returned a hardware error. If this message occurs while data are being transmitted to a communications file, it indicates that the signals being tested with the **OPEN COM** statement were not found in the specified period of time. (Run-time error)

ERR code: 25

Device I/O error

An I/O error occurred on a device I/O operation. The operating system cannot recover from the error. (Run-time error)

ERR code: 57

Device timeout

The program did not receive information from an I/O device within a predetermined amount of time. (Run-time error)

ERR code: 24

Device unavailable

The device you are attempting to access is not on line or does not exist. (Run-time error)

ERR code: 68

Disk full

There is not enough room on the disk for the completion of a **PRINT**, **WRITE**, or **CLOSE** operation. This error can also occur if there is not enough room for QuickBASIC to write out an object or executable file. (Run-time error)

ERR code: 61

Disk-media error

Disk-drive hardware has detected a physical flaw on the disk. (Run-time error)

ERR code: 72

Disk not ready

The disk-drive door is open, or no disk is in the drive. (Run-time error)

ERR code: 71

Division by zero

A division by zero is encountered in an expression, or an exponentiation operation results in zero being raised to a negative power. (Compile-time or run-time error)

ERR code: 11

DO without LOOP

The terminating **LOOP** clause is missing from a **DO...LOOP** statement. (Compile-time error)

Document too large

Your document exceeds QuickBASIC's internal limit. Divide the document into separate files.

Duplicate definition

You are using an identifier that has already been defined. For example, you are attempting to use the same name in a **CONST** statement and as a variable definition, or the same name for a procedure and a variable.

This error also occurs if you attempt to redimension an array. You must use **DIM** or **REDIM** when redimensioning dynamic arrays. (Compile-time or run-time error)

ERR code: 10

Duplicate label

Two program lines are assigned the same number or label. Each line number or label in a module must be unique. (Compile-time error)

Dynamic array element illegal

Dynamic array elements are not allowed with **VARPTR$**. (Compile-time error)

Element not defined

A user-defined type element is referenced but not defined. For example, if the user-defined type `MYTYPE` contained elements A, B, and C, then an attempt to use the variable D as an element of `MYTYPE` would cause this message to appear. (Compile-time error)

ELSE without IF

An **ELSE** clause appears without a corresponding **IF**. Sometimes this error is caused by incorrectly nested **IF** statements. (Compile-time error)

ELSEIF without IF

An **ELSEIF** statement appears without a corresponding **IF**. Sometimes this error is caused by incorrectly nested **IF** statements. (Compile-time error)

END DEF without DEF

An **END DEF** statement has no corresponding **DEF** statement. (Compile-time error)

END IF without block IF

The beginning of an **IF** block is missing. (Compile-time error)

END SELECT without SELECT

The end of a **SELECT CASE** statement appears without a beginning **SELECT CASE**. The beginning of the **SELECT CASE** statement may be missing or misspelled. (Compile-time error)

END SUB or END FUNCTION must be last line in window

You are attempting to add code after a procedure. You must either return to the main module or open another module. (Compile-time error)

END SUB/FUNCTION without SUB/FUNCTION

You deleted the **SUB** or **FUNCTION** statement. (Compile-time error)

END TYPE without TYPE

An **END TYPE** statement is used outside a **TYPE** declaration. (Compile-time error)

Equal sign missing

QuickBASIC expects an equal sign. (Compile-time error)

Error during QuickBASIC initialization

Several conditions can cause this error. It is most commonly caused when there is not enough memory in the machine to load QuickBASIC. If you are loading a Quick library, try reducing the size of the library.

This error may occur when you attempt to use QuickBASIC on unsupported hardware. (QB invocation error)

Error in loading file (*filename*)—Cannot find file

This error occurs when redirecting input to QuickBASIC from a file. The input file is not at the location specified on the command line. (QB invocation error)

Error in loading file (*filename*)—Disk I/O error

This error is caused by physical problems accessing the disk, for example, if the drive door is left open. (QB invocation error)

Error in loading file (*filename*)—DOS memory-area error

The area of memory used by DOS has been written to, either by an assembly language routine or with the **POKE** statement. (QB invocation error)

Error in loading file (*filename*)—Invalid format

You are attempting to load a Quick library that is not in the correct format. This error can occur if you are attempting to use a Quick library created with a previous version of QuickBASIC, if you are trying to use a file that has not been processed with QuickBASIC's Make Library command or the /QU option from LINK, or if you are trying to load a stand-alone (.LIB) library with QuickBASIC. (QB invocation error)

Error in loading file (*filename*)—Out of memory

More memory is required than is available. For example, there may not be enough memory to allocate a file buffer. Try reducing the size of your DOS buffers, getting rid of any terminate-and-stay resident programs, or eliminating some device drivers. If you have large arrays, try placing a **$DYNAMIC** metacommand at the top of your program. If you have documents loaded, then unloading them will free some memory. (Run-time error)

EXIT DO not within DO...LOOP

An **EXIT DO** statement is used outside of a **DO...LOOP** statement. (Compile-time error)

EXIT not within FOR...NEXT

An **EXIT FOR** statement is used outside of a **FOR...NEXT** statement. (Compile-time error)

Expected: *item*

This is a syntax error. The cursor is positioned at the unexpected item. (Compile-time error)

Expression too complex

This error is caused when certain internal limitations are exceeded. For example, during expression evaluation, strings that are not associated with variables are assigned temporary locations. A large number of such strings can cause this error to occur. Try simplifying expressions, and assigning strings to variables. (Compile-time error)

Extra file name ignored

You specified too many files on the command line; the last file name on the line is ignored. (BC invocation error)

Far heap corrupt

The far-heap memory has been corrupted by one of the following:

- The QB compiler does not support a terminate-and-stay-resident program resident in DOS.
- The POKE statement modified areas of memory used by QuickBASIC. (This may modify the descriptor for a dynamic array of numbers or fixed-length strings.)
- The program called an other-language routine that modified areas of memory used by QuickBASIC. (This may modify the descriptor for a dynamic array of numbers or fixed-length strings.

(Compile-time error)

FIELD overflow

A **FIELD** statement is attempting to allocate more bytes than were specified for the record length of a random file. (Run-time error)

ERR code: 50

FIELD statement active

A **GET** or **PUT** statement referred to a record variable used in a a file with space previously allocated by the **FIELD** statement. **GET** or **PUT** with a record variable argument may only be used on files where no **FIELD** statements have been executed. (Run-time error)

ERR code: 56

File already exists

The file name specified in a **NAME** statement is identical to a file name already in use on the disk. (Run-time error)

ERR code: 58

File already open

A sequential-output-mode **OPEN** statement is issued for a file that is already open, or a **KILL** statement is given for a file that is open. (Run-time error)

ERR code: 55

File not found

A **FILES, KILL, NAME, OPEN** or **RUN** statement references a file that does not exist. (Run-time error)

ERR code: 53

File not found in module *module-name* **at address** *segment:offset*

A **FILES, KILL, NAME, OPEN** or **RUN** statement references a file that does not exist. This error message is equivalent to the `File not found` message, but it occurs during execution of compiled programs. The *module-name* is the name of the calling module. The address is the location of the error in the code. (Run-time error)

ERR code: 53

File previously loaded

You are attempting to load a file that is already in memory. (Compile-time error)

Fixed-length string illegal

You are attempting to use a fixed-length string as a formal parameter. (Compile-time error)

FOR index variable already in use

This error occurs when an index variable is used more than once in nested **FOR** loops. (Compile-time error)

FOR index variable illegal

This error is usually caused when an incorrect variable type is used in a **FOR**-loop index. A **FOR**-loop index variable must be a simple numeric variable. (Compile-time error)

FOR without NEXT

Each **FOR** statement must have a matching **NEXT** statement. (Compile-time error)

Formal parameter specification illegal

There is an error in a function or subprogram parameter list. (Compile-time error)

Formal parameters not unique

A **FUNCTION** or **SUB** declaration contains duplicate parameters, as in this example: `SUB GetName(A,B,C,A) STATIC`. (Compile-time error)

Function already defined

This error occurs when a previously defined **FUNCTION** is redefined. (Compile-time error)

Function name illegal

A BASIC reserved word is used as a user-defined **FUNCTION** name. (Compile-time error)

Function not defined

You must declare or define a **FUNCTION** before using it. (Compile-time error)

GOSUB missing

The **GOSUB** is missing from an **ON** *event* statement. (Compile-time error)

GOTO missing

The **GOTO** is missing from an **ON ERROR** statement. (Compile-time error)

GOTO or GOSUB expected

QuickBASIC expects a **GOTO** or **GOSUB** statement. (Compile-time error)

Help not found

Help was requested but not found, and the program contains errors prohibiting QuickBASIC from building a variable table. Press F5 to view the line that caused the error.

Identifier cannot end with %, &, !, #, or $

The above suffixes are not allowed in type identifiers, subprogram names, or names appearing in **COMMON** statements. (Compile-time error)

Identifier cannot include period

User-defined type identifier and record element names cannot contain periods. The period should only be used as a record variable separator. In addition, a variable name cannot contain a period if the part of the name before the period has been used in an *identifier* **AS** *usertype* clause anywhere in the program. If you have programs that use the period in variable names, it is recommended that you change them to use mixed case instead. For example, variable `ALPHA.BETA` would become `AlphaBeta`. (Compile-time error)

Identifier expected

You are attempting to use a number or a BASIC reserved word where an identifier is expected. (Compile-time error)

Identifier too long

Identifiers must not be longer than 40 characters. (Compile-time error)

Illegal function call

A parameter that is out of range is passed to a math or string function. A function-call error can also occur for the following reasons:

- A negative or unreasonably large subscript is used.
- A negative number is raised to a power that is not an integer.
- A negative record number is given when using **GET** *file* or **PUT** *file*.
- An improper or out-of-range argument is given to a function.
- A **BLOAD** or **BSAVE** operation is directed to a nondisk device.
- An I/O function or statement (**LOC** or **LOF**, for example) is performed on a device that does not support it.
- Strings are concatenated to create a string greater than 32,767 characters in length.

(Run-time error)

ERR code: 5

Illegal in direct mode

The statement is valid only within a program and cannot be used in the Immediate window. (Compile-time error)

Illegal in procedure or DEF FN

The statement is not allowed inside a procedure. (Compile-time error)

Illegal number

The format of the number does not correspond to a valid number format. You have probably made a typographical error. For example, the number `2p3` will produce this error. (Compile-time error)

Illegal outside of SUB, FUNCTION, or DEF FN

This statement is not allowed in module-level code. (Compile-time error)

Illegal outside of SUB/FUNCTION

The statement is not allowed in module-level code or **DEF FN** functions. (Compile-time error)

Illegal outside of TYPE block

The *element* **AS** *type* clause is permitted only within a **TYPE...END TYPE** block. (Compile-time error)

Illegal type character in numeric constant

A numeric constant contains an inappropriate type-declaration character. (Compile-time error)

$INCLUDE-file access error

The include file named in the **$INCLUDE** metacommand cannot be located. (Compile-time error)

Include file too large

Your include file exceeds QuickBASIC's internal limit. Break the file into separate files. (Compile-time error)

Input file not found

The source file you gave on the command line is not in the specified location. (BC invocation error)

INPUT missing

The compiler expects the keyword **INPUT**. (Compile-time error)

Input past end of file

An **INPUT** statement reads from a null (empty) file or from a file in which all data have already been read. To avoid this error, use the **EOF** function to detect the end of file. (Run-time error)

ERR code: 62

Input runtime module path:

This prompt appears if the run-time module BRUN45.EXE is not found. Enter the correct path specification. This error is severe and cannot be trapped. (Run-time error)

Integer between 1 and 32767 required

The statement requires an integer argument. (Compile-time error)

Internal error

An internal malfunction occurred in QuickBASIC. Use the Product Assistance Request form included with your documentation to report to Microsoft the conditions under which the message appeared. (Run-time error)

ERR code: 51

Internal error near *xxxx*

An internal malfunction occurred in QuickBASIC at location *xxxx*. Use the Product Assistance Request form included with your documentation to report the conditions under which the message appeared. (Compile-time error)

Invalid character

QuickBASIC found an invalid character, such as a control character, in the source file. (Compile-time error)

Invalid constant

An invalid expression is used to assign a value to a constant. Remember that expressions assigned to constants may contain numeric constants, symbolic constants, and any of the arithmetic or logical operators except exponentiation. A string expression assigned to a constant may consist only of a single literal string. (Compile-time error)

Invalid DECLARE for BASIC procedure

You are attempting to use the **DECLARE** statement keywords **ALIAS, CDECL,** or **BYVAL** with a BASIC procedure. **ALIAS, CDECL,** and **BYVAL** can only be used with non-BASIC procedures. (Compile-time error)

Label not defined

A line label is referenced (in a **GOTO** statement, for example), but does not occur in the program. (Compile-time error)

Label not defined: *label*

A **GOTO** *linelabel* statement refers to a nonexistent line label. (Compile-time error)

Left parenthesis missing

QuickBASIC expected a left parenthesis, or a REDIM statement tried to reallocate space for a scalar. (Compile-time error)

Line invalid. Start again

An invalid file-name character was used following the path characters "\" (backslash) or ":" (colon). (BC invocation error)

Line number or label missing

A line number or label is missing from a statement that requires one, for example, **GOTO.** (Compile-time error)

Line too long

Lines are limited to 255 characters. (Compile-time error)

LOOP without DO

The **DO** starting a **DO...LOOP** statement is missing or misspelled. (Compile-time error)

Lower bound exceeds upper bound

The lower bound exceeds the upper bound defined in a **DIM** statement. (Compile-time error)

Math overflow

The result of a calculation is too large to be represented in BASIC number format. (Compile-time error)

$Metacommand error

A metacommand is incorrect. If you are compiling the program with BC this error is not "fatal"; the program will execute, although the results may be incorrect. (Compile-time warning)

Minus sign missing

QuickBASIC expects a minus sign. (Compile-time error)

Missing Event Trapping (/W) or Checking Between Statements (/V) option

The program contains an **ON** *event* statement requiring one of these options. (Compile-time error)

Missing On Error (/E) option

When using the BC command, programs containing **ON ERROR** statements must be compiled with the On Error (/E) option. (Compile-time error)

Missing Resume Next (/X) option

When using the BC command, programs containing **RESUME**, **RESUME NEXT**, and **RESUME 0** statements must be compiled with the Resume Next (/X) option. (Compile-time error)

Module level code too large

Your module-level code exceeds QuickBASIC's internal limit. Try moving some of the code into **SUB** or **FUNCTION** procedures. (Compile-time error)

Module not found. Unload module from program?

When loading the program, QuickBASIC did not find the file containing the indicated module. QuickBASIC created an empty module instead. You must delete the empty module before you can run the program.

Must be first statement on the line

In block **IF...THEN...ELSE** constructs, **IF**, **ELSE**, **ELSEIF**, and **END IF** can be preceded only by a line number or label. (Compile-time error)

Name of subprogram illegal

The error is caused when a subprogram name is a BASIC reserved word, or a subprogram name is used twice. (Compile-time error)

Nested function definition

A **FUNCTION** definition appears inside another **FUNCTION** definition, or inside an **IF...THEN...ELSE** clause. (Compile-time error)

NEXT missing for *variable*

A **FOR** statement is missing a corresponding **NEXT** statement. The *variable* is the **FOR**-loop index variable. (Compile-time error)

NEXT without FOR

Each **NEXT** statement must have a matching **FOR** statement. (Compile-time error)

No line number in *module-name* **at address** *segment:offset*

This error occurs when the error address cannot be found in the line-number table during error trapping. This happens if there are no integer line numbers between 0 and 65,527. It may also occur if the line-number table has been accidentally overwritten by the user program. This error is severe and cannot be trapped. (Run-time error)

No main module. Choose Set Main Module from the Run menu to select one

You are attempting to run the program after you have unloaded the main module. Every program must have a main module. (Compile-time error)

No RESUME

The end of the program was encountered while the program was in an error-handling routine. A **RESUME** statement is needed to remedy this situation. (Run-time error)

ERR code: 19

Not watchable

This error occurs when you are specifying a variable in a watch expression. Make sure the module or procedure in the active View window has access to the variable you want to watch. For example, module-level code cannot access variables that are local to a **SUB** or **FUNCTION**. (Run-time error)

Numeric array illegal

Numeric arrays are not allowed as arguments to **VARPTR$**. Only simple variables and string array elements are permitted. (Compile-time error)

Only simple variables allowed

User-defined types and arrays are not permitted in **READ** and **INPUT** statements. Array elements that are not of a user-defined type are permitted. (Compile-time error)

Operation requires disk

You are attempting to load from, or save to, a nondisk device such as the printer or keyboard. (Compile-time error)

Option unknown: *option*

You have given an illegal option. (BC invocation error)

Out of DATA

A **READ** statement is executed when there are no **DATA** statements with unread data remaining in the program. (Run-time error)

ERR code: 4

Out of data space

Try modifying your data space requirements as follows:

- Use a smaller file buffer in the **OPEN** statement's **LEN** clause.
- Use the **$DYNAMIC** metacommand to create dynamic arrays. Dynamic array data can usually be much larger than static array data.
- Use fixed-length string arrays instead of variable-length string arrays.
- Use the smallest data type that will accomplish your task. Use integers whenever possible.
- Try not to use many small procedures. QuickBASIC must create several bytes of control information for each procedure.
- Use **CLEAR** to modify the size of the stack. Use only enough stack space to accomplish your task.
- Do not use source lines longer than 255 characters. Such lines require allocation of additional text buffer space.

(Compile-time or run-time error)

Out of memory

More memory is required than is available. For example, there may not be enough memory to allocate a file buffer. Try reducing the size of your DOS buffers, or getting rid of any terminate-and-stay-resident programs, or eliminating some device drivers. If you have large arrays, try placing a **$DYNAMIC** metacommand at the top of your program. If you have documents loaded, unloading them will free some memory. (BC invocation, compile-time, or run-time error)

ERR code: 7

Out of paper

The printer is out of paper or is not turned on. (Run-time error)

ERR code: 27

Out of stack space

This error can occur when a recursive **FUNCTION** procedure nests too deeply, or there are too many active subroutine, **FUNCTION**, and **SUB** calls. You can use the **CLEAR** statement to increase the program's allotted stack space. This error cannot be trapped. (Run-time error)

Out of string space

String variables exceed the allocated amount of string space. (Run-time error)

ERR code: 14

Overflow

The result of a calculation is too large to be represented within the range allowed for either floating-point or integer numbers. (Run-time error)

ERR code: 6

Overflow in numeric constant

The numeric constant is too large. (Compile-time error)

Parameter type mismatch

A subprogram or **FUNCTION** parameter type does not match the **DECLARE** statement argument or the calling argument. (Compile-time error)

Path not found

During an **OPEN**, **MKDIR**, **CHDIR**, or **RMDIR** operation, DOS was unable to find the path specified. The operation is not completed. (Run-time error)

ERR code: 76

Path/File access error

During an **OPEN**, **MKDIR**, **CHDIR**, or **RMDIR** operation, DOS was unable to make a correct connection between the path and file name. The operation is not completed. (Compile-time or run-time error)

ERR code: 75

Permission denied

An attempt was made to write to a write-protected disk, or to access a locked file. (Run-time error)

ERR code: 70

Procedure already defined in Quick library

A procedure in the Quick library has the same name as a procedure in your program. (Compile-time error)

Procedure too large

The procedure has exceeded QuickBASIC's internal limit. Make the procedure smaller by dividing it into several procedures. (Compile-time error)

Program-memory overflow

You are attempting to compile a program whose code segment is larger than 64K. Try splitting the program into separate modules, or use the **CHAIN** statement. (Compile-time error)

Read error on standard input

A system error occurred while reading from the console or a redirected input file. (BC invocation error)

Record/string assignment required

The string or record variable assignment is missing from the **LSET** statement. (Compile-time error)

Redo from start

You have responded to an **INPUT** prompt with the wrong number or type of items. Retype your response in the correct form. (Run-time error)

Rename across disks

An attempt was made to rename a file with a new drive designation. This is not allowed. (Run-time prompt)

ERR code: 74

Requires DOS 2.10 or later

You are attempting to use QuickBASIC with an incorrect version of DOS. (QB invocation or run-time error)

RESUME without error

A **RESUME** statement is encountered before an error-trapping routine is entered. (Run-time error)

ERR code: 20

RETURN without GOSUB

A **RETURN** statement is encountered for which there is no previous, unmatched **GOSUB** statement. (Run-time error)

ERR code: 3

Right parenthesis missing

QuickBASIC expects a right (closing) parenthesis. (Compile-time error)

SEG or BYVAL not allowed in CALLS

BYVAL and **SEG** are permitted only in a **CALL** statement. (Compile-time error)

SELECT without END SELECT

The end of a **SELECT CASE** statement is missing or misspelled. (Compile-time error)

Semicolon missing

QuickBASIC expects a semicolon. (Compile-time error)

Separator illegal

There is an illegal delimiting character in a **PRINT USING** or **WRITE** statement. Use a semicolon or a comma as a delimiter. (Compile-time error)

Simple or array variable expected

The compiler expects a variable argument. (Compile-time error)

Skipping forward to END TYPE statement

An error in the **TYPE** statement has caused QuickBASIC to ignore everything between the **TYPE** and **END TYPE** statement. (Compile-time error)

Statement cannot occur within $INCLUDE file

SUB...END SUB and **FUNCTION...END FUNCTION** statement blocks are not permitted in include files. Use the Merge command from the File menu to insert the include file into the current module, or load the include file as a separate module. If you load the include file as a separate module, some restructuring may be necessary because shared variables are shared only within the scope of the module. (Compile-time error)

Statement cannot precede SUB/FUNCTION definition

The only statements allowed before a procedure definition are the statements **REM** and **DEF***type*. (Compile-time error)

Statement ignored

You are using the BC command to compile a program that contains **TRON** and **TROFF** statements without using the /D option. This error is not "fatal" ; the program will execute, although the results may be incorrect. (Compile-time warning)

Statement illegal in TYPE block

The only statements allowed between the **TYPE** and **END TYPE** statements are **REM** and *element* **AS** *typename*. (Compile-time error)

Statement unrecognizable

You have probably mistyped a BASIC statement. (Compile-time error)

Statements/labels illegal between SELECT CASE and CASE

Statements and line labels are not permitted between **SELECT CASE** and the first **CASE** statement. Comments and statement separators are permitted. (Compile-time error)

STOP in module *name* **at address** *segment:offset*

A **STOP** statement was encountered in the program. (Run-time error)

String assignment required

The string assignment is missing from an **RSET** statement. (Compile-time error)

String constant required for ALIAS

The **DECLARE** statement **ALIAS** keyword requires a string-constant argument. (Compile-time error)

String expression required

The statement requires a string-expression argument. (Compile-time error)

String formula too complex

Either a string formula is too long or an **INPUT** statement requests more than 15 string variables. Break the formula or **INPUT** statement into parts for correct execution. (Run-time error)

ERR code: 16

String space corrupt

This error occurs when an invalid string in string space is being deleted during heap compaction. The probable causes of this error are as follows:

- A string descriptor or string back pointer has been improperly modified. This may occur if you use an assembly-language subroutine to modify strings.
- Out-of-range array subscripts are used and string space is inadvertently modified. The Produce Debug Code option can be used at compile time to check for array subscripts that exceed the array bounds.
- Incorrect use of the **POKE** and/or **DEF SEG** statements may modify string space improperly.
- Mismatched **COMMON** declarations may occur between two chained programs.

(Run-time error)

String variable required

The statement requires a string-variable argument. (Compile-time error)

SUB or FUNCTION missing

A **DECLARE** statement has no corresponding procedure. (Compile-time error)

SUB/FUNCTION without END SUB/FUNCTION

The terminating statement is missing from a procedure. (Compile-time error)

Subprogram error

This is a **SUB** or **FUNCTION** definition error and is usually caused by one of the following:

- The **SUB** or **FUNCTION** is already defined.
- The program contains incorrectly nested **FUNCTION** or **SUB** statements.
- The **SUB** or **FUNCTION** does not terminate with an **END SUB** or **END FUNCTION** statement.

(Compile-time error)

Subprogram not defined

A subprogram is called but never defined. (Compile-time error)

Subprograms not allowed in control statements

Subprogram **FUNCTION** definitions are not permitted inside control constructs such as **IF...THEN...ELSE** and **SELECT CASE**. (Compile-time error)

Subscript out of range

An array element was referenced with a subscript that was outside the dimensions of the array, or an element of an undimensioned dynamic array was accessed. This message may be generated if the Debug (/D) option was specified at compile time. You may also get this error if the array size exceeds 64K, the array is not dynamic, and the /AH option was not used. Reduce the size of the array, or make the array dynamic and use the /AH command-line option. (Run-time error)

ERR code: 9

Subscript syntax illegal

An array subscript contains a syntax error: for example, an array subscript contains both string and integer data types. (Compile-time error)

Syntax error

Several conditions can cause this error. The most common cause at compile time is a mistyped BASIC keyword or argument. At run-time, it is often caused by an improperly formatted **DATA** statement. (Compile-time or run-time error)

ERR code: 2

Syntax error in numeric constant

A numeric constant is not properly formed. (Compile-time error)

THEN missing

QuickBASIC expects a **THEN** keyword. (Compile-time error)

TO missing

QuickBASIC expects a **TO** keyword. (Compile-time error)

Too many arguments in function call

Function calls are limited to 60 arguments. (Compile-time error)

Too many dimensions

Arrays are limited to 60 dimensions. (Compile-time error)

Too many files

At compile time, this error occurs when include files are nested more than five levels deep. It occurs at run time when the 255-file directory maximum is exceeded by an attempt to create a new file with a **SAVE** or **OPEN** statement. (Compile-time or run-time error)

ERR code: 67

Too many labels

The number of lines in the line list following an **ON...GOTO** or **ON...GOSUB** statement exceeds 255 (Compile-time error) or 59 (Run-time error in compiled applications).

Too many named COMMON blocks

The maximum number of named **COMMON** blocks permitted is 126. (Compile-time error)

Too many TYPE definitions

The maximum number of user-defined types permitted is 240. (Compile-time error)

Too many variables for INPUT

An **INPUT** statement is limited to 60 variables. (Compile-time error)

Too many variables for LINE INPUT

Only one variable is allowed in a **LINE INPUT** statement. (Compile-time error)

Type mismatch

The variable is not of the required type. For example, you are trying to use the **SWAP** statement with a string variable and a numeric variable. (Compile-time or run-time error)

ERR code: 13

TYPE missing

The **TYPE** keyword is missing from an **END TYPE** statement. (Compile-time error)

Type more than 65535 bytes

A user-defined type cannot exceed 64K. (Compile-time error)

Type not defined

The *usertype* argument to the **TYPE** statement is not defined. (Compile-time error)

TYPE statement improperly nested

User-defined type definitions are not allowed in procedures. (Compile-time error)

TYPE without END TYPE

There is no **END TYPE** statement associated with a **TYPE** statement. (Compile-time error)

Typed variable not allowed in expression

Variables that are user-defined types are not permitted in expressions such as `CALL ALPHA((X))`, where `X` is a user-defined type. (Compile-time error)

Unexpected end of file in TYPE declaration

There is an end-of-file character inside a **TYPE...END TYPE** block.

Unprintable error

An error message is not available for the error condition that exists. This may be caused by an **ERROR** statement that doesn't have a defined error code. (Run-time error)

Unrecognized switch error: "QU"

You are attempting to create an **.EXE** file or Quick library with an incorrect version of the Microsoft Overlay Linker. You must use the linker supplied on the QuickBASIC distribution disks to create an **.EXE** file or Quick library. (Compile-time error)

Valid options: [/RUN] *file* **/AH /B /C:***buf* **/G /NOHI /H /L [***lib***] /MBF /CMD** *string*

This message appears when you invoke QuickBASIC with an invalid option. (QB invocation error)

Variable-length string required

Only variable-length strings are permitted in a **FIELD** statement. (Compile-time error)

Variable name not unique

You are attempting to define the variable *x* as a user-defined type after *x.y* has been used. (Compile-time error)

Variable required

QuickBASIC encountered an **INPUT, LET, READ,** or **SHARED** statement without a variable argument. (Compile-time error)

Variable required

A **GET** or **PUT** statement didn't specify a variable when an operation is performed on a file opened in **BINARY** mode. (Run-time error)

ERR code: 40

WEND without WHILE

This error is caused when a **WEND** statement has no corresponding **WHILE** statement. (Compile-time error)

WHILE without WEND

This error is caused when a **WHILE** statement has no corresponding **WEND** statement. (Compile-time error)

Wrong number of dimensions

An array reference contains the wrong number of dimensions. (Compile-time error)

I.3 LINK Error Messages

This section lists and describes error messages generated by the Microsoft Overlay Linker, LINK.

Fatal errors cause the linker to stop execution. Fatal error messages have the following format:

location `: fatal error L1xxx :` *messagetext*

Nonfatal errors indicate problems in the executable file. LINK produces the executable file. Nonfatal error messages have the following format:

location : `error L2xxx:` *messagetext*

Warnings indicate possible problems in the executable file. LINK produces the executable file. Warnings have the following format:

location : `warning L4xxx:` *messagetext*

In these messages, *location* is the input file associated with the error, or LINK if there is no input file.

The following error messages may appear when you link object files with LINK:

Number	LINK Error Message

L1001 *option* **: option name ambiguous**

A unique option name did not appear after the option indicator (/). For example, the command

```
LINK /N main;
```

generates this error, since LINK cannot tell which of the three options beginning with the letter "N" was intended.

L1002 *option* **: unrecognized option name**

An unrecognized character followed the option indicator (/), as in the following example:

```
LINK /ABCDEF main;
```

L1003 **/QUICKLIB, /EXEPACK incompatible**

You specified two options that cannot be used together: /QUICKLIB and /EXEPACK.

L1004 *option* **: invalid numeric value**

An incorrect value appeared for one of the LINK options. For example, a character string was given for an option that requires a numeric value.

L1006 *option* **: stack size exceeds 65535 bytes**

The value given as a parameter to the /STACKSIZE option exceeds the maximum allowed.

L1007 *option* **: interrupt number exceeds 255**

For the /OVERLAYINTERRUPT option, a number greater than 255 was given as a value.

L1008 *option* **: segment limit set too high**

The limit on the number of segments allowed was set to greater than 3072 using the /SEGMENTS option.

L1009 *number* **: CPARMAXALLOC : illegal value**

The number specified in the /CPARMAXALLOC option was not in the range 1–65,535.

L1020 **no object modules specified**

No object-file names were specified to LINK.

L1021 **cannot nest response files**

A response file occurred within a response file.

L1022 **response line too long**

A line in a response file was longer than 127 characters.

L1023 **terminated by user**

You entered CTRL+C or CRTL+BREAK

L1024 **nested right parentheses**

The contents of an overlay were typed incorrectly on the command line.

L1025 **nested left parentheses**

The contents of an overlay were typed incorrectly on the command line.

L1026 **unmatched right parenthesis**

A right parenthesis was missing from the contents specification of an overlay on the command line.

L1027 **unmatched left parenthesis**

A left parenthesis was missing from the contents specification of an overlay on the command line.

L1043 **relocation table overflow**

More than 32,768 long calls, long jumps, or other long pointers appeared in the program.

Try replacing long references with short references, where possible, and recreate the object module.

L1045 **too many TYPDEF records**

An object module contained more than 255 TYPDEF records. These records describe communal variables. This error can appear only with programs produced by the Microsoft FORTRAN Compiler or other compilers that support communal variables. (TYPDEF is a DOS term. It is explained in the *Microsoft MS-DOS Programmer's Reference* and in other reference books on DOS.)

L1046 **too many external symbols in one module**

An object module specified more than the limit of 1023 external symbols.

Break the module into smaller parts.

L1047 **too many group, segment, and class names in one module**

The program contained too many group, segment, and class names.

Reduce the number of groups, segments, or classes, and recreate the object file.

L1048 **too many segments in one module**

An object module had more than 255 segments.

Split the module or combine segments

L1049 **too many segments**

The program had more than the maximum number of segments. Use the /SEGMENTS option, which has a default value of 128, to specify the maximum legal number of segments. The default is 128 in the /SEGMENTS option, which specifies the maximum legal number.

Relink using the /SEGMENTS option with an appropriate number of segments.

L1050 **too many groups in one module**

LINK encountered over 21 group definitions (GRPDEF) in a single module.

Reduce the number of group definitions or split the module. (Group definitions are explained in the *Microsoft MS-DOS Programmer's Reference* and in other reference books on DOS.)

L1051 **too many groups**

The program defined more than 20 groups, not counting DGROUP.

Reduce the number of groups.

L1052 **too many libraries**

An attempt was made to link with more than 32 libraries.,

Combine libraries, or use modules that require fewer libraries.

L1053 **out of memory for symbol table**

There is no fixed limit to the size of the symbol table. However, it is limited by the amount of available memory.

Combine modules or segments and recreate the object files. Eliminate as many public symbols as possible.

L1054 **requested segment limit too high**

LINK did not have enough memory to allocate tables describing the number of segments requested. (The default is 128 or the value specified with the /SEGMENTS option.)

Try linking again using the /SEGMENTS option to select a smaller number of segments (for example, use 64 if the default was used previously), or free some memory by eliminating resident programs or shells.

L1056 **too many overlays**

The program defined more than 63 overlays.

L1057 **data record too large**

A LEDATA record (in an object module) contained more than 1024 bytes of data. This is a translator error. (LEDATA is a DOS term, which is explained in the *Microsoft MS-DOS Programmer's Reference* and in other DOS reference books.)

Note which translator (compiler or assembler) produced the incorrect object module and the circumstances. Please report this error to Microsoft Corporation using the Product Assistance Request form included with the documentation.

L1063 **out of memory for CodeView information**

Too many linked object (".OBJ") files contain debugging information. Turn off the Produce Debug Code option in the Make EXE file dialog box.

L1070 **segment size exceeds 64K**

A single segment contained more than 64K of code or data.

Try compiling and linking using the large model.

L1071 **segment _TEXT larger than 65520 bytes**

This error is likely to occur only in small-model C programs, but it can occur when any program with a segment named _TEXT is linked using the /DOSSEG option. Small-model C programs must reserve code addresses 0 and 1; this range is increased to 16 for alignment purposes.

L1072 **common area longer than 65536 bytes**

The program had more than 64K of communal variables. This error occurs only with programs produced by compilers that support communal variables.

L1080 **cannot open list file**

The disk or the root directory was full.

Delete or move files to make space.

L1081 **out of space for run file**

The disk on which the executable file was being written was full.

Free more space on the disk and restart LINK.

L1083 **cannot open run file**

The disk or the root directory was full.

Delete or move files to make space.

L1084 **cannot create temporary file**

The disk or root directory was full.

Free more space in the directory and restart LINK.

L1085 **cannot open temporary file**

The disk or the root directory was full.

Delete or move files to make space.

L1086 **scratch file missing**

An internal error has occurred.

Note the circumstances of the problem and contact Microsoft Corporation using the Product Assistance Request form included with the documentation.

L1087 **unexpected end-of-file on scratch file**

The disk with the temporary output file from LINK was removed.

L1088 **out of space for list file**

The disk where the listing file is being written is full.

Free more space on the disk and restart LINK.

L1089 *filename* **: cannot open response file**

LINK could not find the specified response file.

This usually indicates a typing error.

L1090 **cannot reopen list file**

The original disk was not replaced at the prompt.

Restart LINK.

L1091 **unexpected end-of-file on library**

The disk containing the library probably was removed.

Replace the disk containing the library and run LINK again.

L1093 **object not found**

One of the object files specified in the input to LINK was not found.

Restart LINK and specify the object file.

L1101 **invalid object module**

One of the object modules was invalid.

If the error persists after recompiling, please contact Microsoft Corporation using the Product Assistance Request form included with the documentation.

L1102 **unexpected end-of-file**

An invalid format for a library was encountered.

L1103 **attempt to access data outside segment bounds**

A data record in an object module specified data extending beyond the end of a segment. This is a translator error.

Note which translator (compiler or assembler) produced the incorrect object module and the circumstances in which it was produced. Please report this error to Microsoft Corporation using the Product Assistance Request form included with the documentation.

L1104 *filename* **: not valid library**

The specified file was not a valid library file. This error causes LINK to abort.

L1113 **unresolved COMDEF; internal error**

Note the circumstances of the failure and contact Microsoft Corporation using the Product Assistance Request form included with the documentation.

L1114 **file not suitable for /EXEPACK; relink without**

For the linked program, the size of the packed load image plus packing overhead was larger than that of the unpacked load image.

Relink without the /EXEPACK option.

L1115 **/QUICKLIB, overlays incompatible**

You specified overlays and used the /QUICKLIB option. These cannot be used together.

L2001 **fixup(s) without data**

A FIXUPP record occurred without a data record immediately preceding it. This is probably a compiler error. (See the *Microsoft MS-DOS Programmer's Reference* for more information on FIXUPP.)

L2002 **fixup overflow near** *number* **in frame seg** *segname* **target seg** *segname* **target offset** *number*

The following conditions can cause this error:

- A group is larger than 64K.
- The program contains an intersegment short jump or intersegment short call.
- The name of a data item in the program conflicts with the name of a library subroutine included in the link.
- An **EXTRN** declaration in an assembly-language source file appeared inside the body of a segment, as in the following example:

```
code    SEGMENT public 'CODE'
        EXTRN   main:far
start   PROC    far
        call    main
        ret
start   ENDP
code    ENDS
```

The following construction is preferred:

```
EXTRN   main:far
code    SEGMENT public 'CODE'
start   PROC    far
        call    main
        ret
start   ENDP
code    ENDS
```

Revise the source file and recreate the object file. (See the *Microsoft MS-DOS Programmer's Reference* for information about frame and target segments.

L2003 **intersegment self-relative fixup at *offset* in segment *segname***

You tried to make a near call or jump to a far entry in segment *segname* at *offset*.

Change the call or jump to far or make the entry near.

L2004 **LOBYTE-type fixup overflow**

A LOBYTE fixup generated an address overflow. (See the *Microsoft MS-DOS Programmer's Reference* for more information.)

L2005 **fixup type unsupported**

A fixup type occurred that is not supported by the Microsoft linker. This is probably a compiler error.

Note the circumstances of the failure and contact Microsoft Corporation using the Product Assistance Request form included with the documentation.

L2011 *name* **: NEAR/HUGE conflict**

Conflicting NEAR and HUGE attributes were given for a communal variable. This error can occur only with programs produced by compilers that support communal variables.

L2012 *name* **: array-element size mismatch**

A far communal array was declared with two or more different array-element sizes (for instance, an array was declared once as an array of characters and once as an array of real numbers). This error occurs only with compilers that support far communal arrays.

L2013 **LIDATA record too large**

A LIDATA record contains more than 512 bytes. This error is usually caused by a compiler error.

L2024 *name* **: symbol already defined**

LINK has found a public-symbol redefinition. Remove extra definition(s).

L2025 *name* **: symbol defined more than once**

Remove the extra symbol definition from the object file.

L2029 **unresolved externals**

One or more symbols were declared to be external in one or more modules, but they were not publicly defined in any of the modules or libraries. A list of the unresolved external references appears after the message, as shown in the following example:

```
unresolved externals
EXIT in file(s):
 MAIN.OBJ (main.for)
OPEN in file(s):
 MAIN.OBJ (main.for)
```

The name that comes before `in file(s)` is the unresolved external symbol. On the next line is a list of object modules that have made references to this symbol. This message and the list are also written to the map file, if one exists.

L2041 **stack plus data exceed 64K**

The total size of near data and the stack exceeds 64K. Reduce the stack size to control the error.

LINK tests for this condition only if the /DOSSEG option is enabled. This option is automatically enabled by the library startup module.

L2043 **Quick library support module missing**

You did not specify, or LINK could not find, the object module or library required for creating a Quick library. In the case of QuickBASIC, the library provided is BQLB45.LIB.

L2044 ***name* : symbol multiply defined, use /NOE**

LINK has found a possible public-symbol redefinition. This error is often caused by redefining a symbol defined in a library.

Relink using the /NOEXTDICTIONARY option.

This error in combination with error `L2025` for the same symbol indicates a real redefinition error.

L4011 **PACKCODE value exceeding 65500 unreliable**

Packcode segment sizes that exceed 65,500 bytes may be unreliable on the Intel® 80286 processor.

L4012 **load-high disables EXEPACK**

The /HIGH and /EXEPACK options cannot be used at the same time.

L4015 **/CODEVIEW disables /DSALLOCATE**

The /CODEVIEW and /DSALLOCATE options cannot be used at the same time.

L4016 **/CODEVIEW disables /EXEPACK**

The /CODEVIEW and /EXEPACK options cannot be used at the same time.

L4020 *name* **: code-segment size exceeds 65500**

Code segments of 65,501–65,536 bytes in length may be unreliable on the Intel 80286 processor.

L4021 **no stack segment**

The program did not contain a stack segment defined with **STACK** combine type. This message should not appear for modules compiled with Microsoft QuickBASIC, but it could appear for an assembly-language module.

Normally, every program should have a stack segment with the combine type specified as **STACK**. You can ignore this message if you have a specific reason for not defining a stack or for defining one without the **STACK** combine type. Linking with versions of LINK earlier than Version 2.40 might cause this message, since these linkers search libraries only once.

L4031 *name* **: segment declared in more than one group**

A segment was declared to be a member of two different groups.

Correct the source file and recreate the object files.

L4034 **more than 239 overlay segments; extra put in root**

The program designates more than 239 segments to go in overlays. When this error occurs, segments beginning with number 234 are placed in the root, the permanently resident portion.

L4045 **name of output file is** *name*

The prompt for the run-file field gave an inaccurate default because the /QUICKLIB option was not used early enough. The output will be a Quick library with the name given in the error message.

L4050 **too many public symbols for sorting**

The number of public symbols exceeds the space available for sorting the symbols as requested by the /MAP option. The symbols are left unsorted.

L4051 *filename* **: cannot find library**

LINK could not find the specified file.

Enter a new file name, a new path specification, or both.

L4053 **VM.TMP : illegal file name; ignored**

VM.TMP appeared as an object-file name. Rename the file and rerun LINK.

L4054 *filename* **: cannot find file**

LINK could not find the specified file. Enter a new file name, a new path specification, or both.

I.4 LIB Error Messages

Error messages generated by the Microsoft Library Manager, LIB, have one of the following formats:

{*filename* | LIB} : fatal error U1*xxx*: messagetext
{*filename* | LIB} : error U2*xxx*: messagetext
{*filename* | LIB} : warning U4*xxx*: messagetext

The message begins with the input-file name (*filename*), if one exists, or with the name of the utility. If possible, LIB prints a warning and continues operation. In some cases errors are fatal, and LIB terminates processing.

LIB may display the following error messages:

Number	LIB Error Message

U1150 **page size too small**

The page size of an input library was too small, which indicates an invalid input .LIB file.

U1151 **syntax error : illegal file specification**

A command operator such as a minus sign (–) was given without a following module name.

U1152 **syntax error : option name missing**

A forward slash (/) was given without an option after it.

U1153 **syntax error : option value missing**

The /PAGESIZE option was given without a value following it.

U1154 **option unknown**

An unknown option was given. Currently, LIB recognizes only the /PAGESIZE option.

U1155 **syntax error : illegal input**

The given command did not follow correct LIB syntax as specified in Appendix G, "Compiling and Linking from DOS."

U1156 **syntax error**

The given command did not follow correct LIB syntax as specified in Appendix G, "Compiling and Linking from DOS."

U1157 **comma or new line missing**

A comma or carriage return was expected in the command line but did not appear. This may indicate an inappropriately placed comma, as in the line that follows:

```
LIB math.lib,-mod1+mod2;
```

The line should have been entered as follows:

```
LIB math.lib -mod1+mod2;
```

U1158 **terminator missing**

Either the response to the "Output library" prompt or the last line of the response file used to start LIB did not end with a carriage return.

U1161 **cannot rename old library**

LIB could not rename the old library to have a .BAK extension because the .BAK version already existed with read-only protection.

Change the protection on the old .BAK version.

U1162 **cannot reopen library**

The old library could not be reopened after it was renamed to have a .BAK extension.

U1163 **error writing to cross-reference file**

The disk or root directory was full.

Delete or move files to make space.

U1170 **too many symbols**

More than 4609 symbols appeared in the library file.

U1171 **insufficient memory**

LIB did not have enough memory to run.

Remove any shells or resident programs and try again, or add more memory.

U1172 **no more virtual memory**

Note the circumstances of the failure and notify Microsoft Corporation using the Product Assistant Request form included with the documentation.

U1173 **internal failure**

Note the circumstances of the failure and notify Microsoft Corporation using the Product Assistant Request form included with the documentation.

U1174 **mark: not allocated**

Note the circumstances of the failure and notify Microsoft Corporation using the Product Assistance Request form included with the documentation.

U1175 **free: not allocated**

Note the circumstances of the failure and notify Microsoft Corporation using the Product Assistance Request form included with the documentation.

U1180 **write to extract file failed**

The disk or root directory was full.

Delete or move files to make space.

U1181 **write to library file failed**

The disk or root directory was full.

Delete or move files to make space.

U1182 *filename* **: cannot create extract file**

The disk or root directory was full, or the specified extract file already existed with read-only protection.

Make space on the disk or change the protection of the extract file.

U1183 **cannot open response file**

The response file was not found.

U1184 **unexpected end-of-file on command input**

An end-of-file character was received prematurely in response to a prompt.

U1185 **cannot create new library**

The disk or root directory was full, or the library file already existed with read-only protection.

Make space on the disk or change the protection of the library file.

U1186 **error writing to new library**

The disk or root directory was full.

Delete or move files to make space.

U1187 **cannot open VM.TMP**

The disk or root directory was full.

Delete or move files to make space.

U1188 **cannot write to VM**

Note the circumstances of the failure and notify Microsoft Corporation using the Product Assistance Request form.

U1189 **cannot read from VM**

Note the circumstances of the failure and notify Microsoft Corporation using the Product Assistance Request form.

U1190 **interrupted by user**

The user pressed CTRL+C or CTRL+BREAK.

U1200 *name* **: invalid library header**

The input library file had an invalid format. It was either not a library file, or it had been corrupted.

U1203 *name* **: invalid object module near** *location*

The module specified by *name* was not a valid object module.

U2152 *filename* **: cannot create listing**

The directory or disk was full, or the cross-reference-listing file already existed with read-only protection.

Either make space on the disk or change the protection of the cross-reference-listing file.

U2155 *modulename* **: module not in library; ignored**

The specified module was not found in the input library.

U2157 *filename* **: cannot access file**

LIB was unable to open the specified file.

U2158 *libraryname* **: invalid library header; file ignored**

The input library had an incorrect format.

U2159 *filename* **: invalid format** *hexnumber*; **file ignored**

The signature byte or word *hexnumber* of the given file was not one of the following recognized types: Microsoft library, Intel library, Microsoft object, or XENIX® archive.

U4150 *modulename* **: module redefinition ignored**

A module was specified to be added to a library but a module with the same name was already in the library. Or, a module with the same name was found more than once in the library.

U4151 *symbol* **: symbol redefined in module** *modulename*, **redefinition ignored**

The specified symbol was defined in more than one module.

U4153 *number* **: page size too small; ignored**

The value specified in the /PAGESIZE option was less than 16.

U4155 *modulename* **: module not in library**

A module to be replaced is not in the library. LIB adds the module to the library.

U4156 *libraryname* **: output-library specification ignored**

An output library was specified in addition to a new library name. For example, specifying

```
LIB new.lib+one.obj,new.lst,new.lib
```

where `new.lib` does not already exist causes this error.

U4157 **Insufficient memory, extended dictionary not created.**

LIB was unable to create the extended dictionary. The library is still valid, but LINK cannot take advantage of extended dictionaries to link faster.

U4158 **Internal error, extended dictionary not created.**

LIB was unable to create the extended dictionary. The library is still valid, but LINK cannot take advantage of extended dictionaries to link faster.

" (double quotes), ending fields, 101
"" (null string), 91
$ (dollar sign), string-type suffix, 136
' (apostrophe), introducing comments, xxvii
* (asterisk) AS, fixed-length string with, 136–137
+ (plus)
  operator, combining strings, 138
  sign, LIB command symbol, 373
, (comma)
  fields, ending, 101
  variable separator, 89
/ (forward slash), LINK option character, 361
– (minus sign), LIB command symbol, 373
–+ (minus sign-plus sign), LIB command symbol, 373
–* (minus sign-asterisk), LIB command symbol, 373
* (asterisk)
  fixed-length string with AS, 136, 137
  LIB command symbol, 373
; (semicolon), LIB command symbol, 84, 90, 372

A

,A option, 353
  BASICA, 311
  $INCLUDE files, 348
ABS function, 266
ABSOLUTE, 384
Absolute coordinates. *See* Coordinates, absolute
/AH option, 328, 353
ALIAS, DECLARE statement, use in, 271
Aliasing, variable, 68
Alphabetizing strings, 140
AND operator, 7
AND option
  PUT graphics statement, with, 197
Angle measurements, 160
Animate mode, 330
Animation
  GET and PUT, 193, 200
  graphics statements, simple, 193
  image flicker, reducing, 203
  PALETTE USING, 177
  screen pages, 205
Apostrophe (')
  entering, xxv
  introducing comments, xxvii
Arc, 160, 162
Arctangent, ATN function, 266
Arguments
  correct type and number, checking for, 55
  LINK options, 362
  parameters
    agreement with, 55
    distinguished from, 48
  passing
    by reference, 60
    by value, 60–61
    described, 43
    limits on, 338
  SUB procedures, passing to, 47
Array-bound functions, 53
Arrays
  dynamic
    ERASE statement, 273
    REDIM statement, 286
  elements, passing, 52
  format
    argument list, 49
    FUNCTION statement, 52
    parameter list, 49, 52
    SHARED statement, 62
    SUB statement, 49, 52
  LBOUND function, 278
  limits (table), 337
  lower-bound function, 53
  memory allocation, 348
  procedures
    passing to, 49, 52
    sharing with, 62
  row-order option, 328
  static, ERASE statement, 273
  subscripts, specifying
    lower bound, 283
    maximum value, 272
    number of, 272
  UBOUND (upper-bound) function, 53, 294
  variables, 272
Arrow keys. *See* DIRECTION keys
ASC function, 137, 266, 305
ASCII
  character codes
    (table), 339, 341, 343
    CHR$ function, arguments to, 137, 266
    determining with the ASC function, 137, 266
    storing characters in memory, 137
  files, reading as sequential files, 100

Aspect ratio
  CIRCLE statement, 159
  correction for, 165
  defined, 164
  screen size, computing for, 165
  squares, drawing, 164
Assembly-language listings, 324
Assignment statements, 279
Asterisk (*)
  fixed-length string with AS, 136–137
  LIB command symbol, 373
ATN function, 266
Attributes
  colors, assigning, 176
  screen modes, EGA and VGA, 173–174, 176
  STATIC, 335
Automatic variables, 44, 69
AUX (device name), 356

B

B_OnExit routine, 387
Background color default, 152
Background music, 284
.BAS file, execution by QuickBASIC, 312
BASIC
  error codes, 273
  reserved words, 345
  run-time errors, 391
BASICA
  compatibility, 311
  QuickBASIC, converting to, 311
/BATCH option (LINK), 363
Batch file, when to use, 351
Baud rate, 120
BC command
  command line, invoking from, 351
  file-name usage, 352
  options
    (list), 353
    /A, 353
    /AH, 328, 353
    /C, 354
    /D, 328, 354
    /E, 328, 354
    /MBF, 318, 319, 354
    /O, 354
    /R, 328, 354
    /S, 328, 354
    /V, 328, 354
    /W, 247, 328, 354
    /X, 247, 328, 355
BC command (*continued*)
  options (*continued*)
    /ZD, 355
    /ZI, 355
    new (table), 328
    not used (table), 328
  versions, differences among, 329
BC.EXE, 350
BEEP statement, 266
Binary
  file access
    OPEN statement syntax, 335
    random access, contrasted with, 115
    versions, differences among, 322
  input, 107
  numbers, converting to hexadecimal, 183
  search, example, 129
Binary-access files
  creating, 97, 115
  opening, 97, 115
  reading, 116
  writing to, 116
Bit planes, 188
Bitwise operators, 7–8
  See also Logical operators
Blank COMMON block, 73
BLOAD statement, 266, 312
Block IF...THEN...ELSE statement, 276
Bold text, indicating keywords, xxiv
Boolean
  constants, 8
  expressions
    definition, 6
    logical operators, 7
    other expressions, comparing with, 6
    relational operators (table), 6
Boxes, LINE statement, 156
BRUN45.LIB, default for linker, 357
BSAVE statement, 266, 312
BUILDLIB utility, 329

C

/C option (BC), 354
CALL ABSOLUTE statement, 267, 384
CALL INT86OLD statement, 267, 384
CALL INT86XOLD statement, 267
CALL INTERRUPT statement, 267, 384
CALL INTERRUPTX statement, 267
CALL statement
  BASIC procedures, 266
  DECLARE, used with, 332

CALL statement (*continued*)
described, 299
non-BASIC procedures, 267
optional use of, 332
QuickBASIC/interpreter differences, 312
SUB procedure, calling, 47, 57
CALLS statement, 267
Carriage-return–line-feed sequence, 100
Cartesian coordinates. *See* View coordinates
Case
sensitivity, 99
significance, LINK options, 361, 368
CASE clause, 289
See also SELECT CASE statement
CASE$ function, 334
CDBL function, 267
CDECL, use in DECLARE statement, 271
CGA
palette, changing, 174
screen modes supported by, 172
CHAIN statement
described, 73, 267, 300
QuickBASIC/interpreter differences, 312
Chaining programs, statements used
See also individual statement names
CHAIN, 267
COMMON, 269
Characters
how stored, 137
limits, 338
CHDIR statement, 268
Checking between statements option, 328
Choosing
commands, differences among versions, 325
options, differences among versions, 325
CHR$ function, 137, 268, 305
CINT function, 268
CIRCLE statement
arcs, 160, 162
circles, 158
described, 268, 306
ellipses, 159
pie shapes, 162
Circles, drawing, 158
CLEAR statement, 72, 268, 332
CLNG function, 268, 332
CLOSE statement, 99, 268, 302
CLS statement, 269, 333
/CODEVIEW option (LINK), 367
CodeView debugger, LINK option for, 367
Color attributes. *See* Attributes
Color Graphics Adapter. *See* CGA
COLOR statement
background color, controlling, 152, 174
described, 269, 306, 333
foreground color, controlling, 152, 174
palette, changing with, 174
screen mode 1, syntax in, 174
Colors
background, controlling, 152, 174
CGA, using with, 172
clarity, tradeoff with, 172
foreground, controlling, 152, 174
graphics statements, specifying in, 173
PALETTE USING, changing with, 177
PALETTE, changing with, 176
Columns
changing number of on screen, 86
skipping, 86
COM devices, 118
COM statements, 269, 357
Comma (,)
ending fields, 101
variable separator, 89
Command line
BASIC program passing to, 269
Quick library, creating from, 385
COMMAND$ function
described, 269
limits, 338
Commands
BC. *See* BC command
Linker. *See* LINK
versions, differences among, 325
Commands QB. *See* QB command
Comments
introducing with apostrophe, xxvii
REM statement, 287
COMMON block
blank, 73
named, 67, 73
COMMON statement
AS clause, 332
chaining programs, 269
described, 299
$INCLUDE metacommand, used with, 73
QuickBASIC/interpreter differences, 312
SHARED attribute
defined, 62
described, 299
variables, making global, 64
sharing, 66, 73
Comparing strings
fixed- and variable-length, 140

Comparing strings (*continued*)
relational operators, with, 140
Compatibility
BASICA and GW-BASIC, 311
versions, files, 336
Compile options, 328
Compile-time error messages, 391
Compiling from DOS, 350
Complex numbers, defined, 212
CON (device name), 356
Concatenation, defined, 139
CONS, 118
CONST statement, 269, 333
Constants
input list, 89
procedures, passing to, 49
string, literal and symbolic, 136
symbolic, xxvi, 269
Control-flow statements
See also individual statement names
CALL, 266–267
CALL ABSOLUTE, 267
CALLS, 267
CHAIN, 267
DEF FN, 271
DO...LOOP, 272
FOR...NEXT, 274
FUNCTION, 275
GOSUB...RETURN, 276
GOTO, 276
IF...THEN...ELSE, 276
ON...GOSUB, 282
ON...GOTO, 282
RETURN, 287
SELECT CASE, 289
WHILE...WEND, 295
Control-flow structures
BASIC, used in (table), 298
decision, 9
defined, 5
indenting, xxvii
looping, 19
new, 6
Controlling
linker, 361
segments, 365
stack size, 368
Converting
BASICA and GW-BASIC programs, 311
data files, 318
IEEE format, program for, 320
Coordinates
absolute
STEP, specifying with, 154
VIEW SCREEN, specifying with, 166
physical
GET statement, 195
pixels, locating, 168
view coordinates, translating to, 171
pixels, locating, 151
relative
defined, 154
STEP, 154
VIEW, 166
view
GET statement, 195
physical coordinates, translating to, 171
WINDOW, defining with, 168
viewport, specifying values outside, 168
COS function (cosine), 270
/CPARMAXALLOC option (LINK), 368
Cross-reference-file listing, 372
CSNG function, 270
CSRLIN function, 270, 301
Cursor
graphics, 154, 171
text
defined, 92
LOCATE, positioning with, 93
location, finding, 94
shape, changing, 93
CV*type* statement, 109, 305
CVD function, 270
CVDMBF function, 270, 320, 333
CVI function, 270
CVL function, 270, 333
CVS function, 270
CVSMBF function, 270, 320, 333

D

/D option (BC), 328, 354
Data files
advantages of, 95
closing, 99
creating, 96
defined, 95
file numbers, 97
file-access errors, trapping, 248
opening, 96, 338
organization, 95
random-access, 96

Data files (*continued*)
reading, 105–107
records, adding, 103
sequential, 96
DATA statement, 270
Data types
specifying, 272
TYPE statement, 294
Data-file buffer, 99
Date and time
functions
DATE$, 270
TIME$, 293
statements
DATE$, 271
TIME$, 293
DATE$ function, 270
DATE$ statement, 271
Debug option, 328
Debugging
/CODEVIEW (LINK) option, 367
statements, 293
versions, differences among, 330
Decision structures
defined, 9
IF...THEN...ELSE
block, 10
single-line, 9
SELECT CASE, 11, 13
Declarations
See also individual statement names
CONST statement, 269
DECLARE statement (BASIC procedures), 271
DECLARE statement (non-BASIC procedures), 271
DEF*type* statements, 272
DIM statement, 272
DECLARE statement
arguments, checking with, 55
AS clause, 332
described, 271, 299
include files, 57
QuickBASIC, not generated by, 55
versions, differences among, 333
where required, 56
Declaring arrays, limits, 337
DEF FN functions
exit from, alternative, 274
FUNCTION procedure, contrasted with, 42
local variables in, 42
DEF FN statement, 271, 300
DEF SEG statement, 271
DEF*type* statements
described, 272
QuickBASIC/interpreter differences, 312
DEFDBL statement, 312
DEFINT statement, 312
DEFLNG statement, 333
DEFSNG statement, 312
DEFSTR statement, 312
Degrees
compared with radians, 160
converting to radians, 161
Device I/O, contrasted with file I/O, 119
Device-status information, 273
Devices
COM, 118
CONS, 118
function handling
IOCTL$, 277
LPOS, 280
PEN, 284
I/O modes, valid, 119
KYBD, 118
LPT, 118–119
names, 365
SCRN, 118–119
statement handling
IOCTL, 278
OPEN COM, 283
OUT, 283
WAIT, 295
Differences among versions
(table), 315
file compatibility, 336
DIM statement
AS clause, 332
described, 272
QuickBASIC/interpreter differences, 312
SHARED attribute
defined, 62
example, variable aliasing, 68
prohibited, 45
variables, making global, 64
TO clause, 333
Dimensioning arrays, limits, 337
DIRECTION keys, trapping, 235
Directory statements
See also individual statement names
CHDIR, 268
FILES, 274
MKDIR, 281
NAME, 282
RMDIR, 287

Display memory, PCOPY statement, 283
DO UNTIL statement, 272
DO WHILE statement, 272
DO...LOOP statement
  described, 298, 333
  exit from, alternative, 31, 274
  flow control, 272
  keyword
    UNTIL, 31
    WHILE, 31
  loop test, location of, 30
  syntax, 27
  WHILE...WEND, contrasted with, 26
Document conventions, xxiv
Dollar sign ($), string-type suffix, 136
DOS, 267, 273, 290
DOS-level commands, xxiv
/DOSSEG option (LINK), 368
Dotted lines, drawing, 157
Double quotes ("), ending fields, 101
Double-precision numbers
  converting to, 267
  size limits, 337
DRAW statement
  described, 272, 306
  figures, rotating, 206
  QuickBASIC/interpreter differences, 312
  VARPTR$ function, using, 295
Dumb terminal, defined, 132
Dummy arguments, POS function, 94
$DYNAMIC metacommand, 348
Dynamic arrays
  $DYNAMIC metacommand, 348
  ERASE statement, 273
  REDIM statement, 286

E

/E option
  BC, 247, 328, 354
  QB, 328
Editing, differences among versions, 328
EGA, changing palette, 174, 176
Ellipses, drawing, 159
ELSE clause, 9
ELSEIF clause, 11
END CASE clause, 289
END DEF statement, 272
END FUNCTION statement, 272
END IF statement, 272
END SELECT statement, 272
END statement, 272
END SUB statement, 272
END TYPE statement, 272
End-of-line markers, 34
Enhanced Graphics Adapter. *See* EGA
ENTER key, equivalent to SPACEBAR, 325
ENVIRON statement, 273
ENVIRON$ function, 273
Environment variables
  described, 273
  LIB, 359
  LINK, 362
  TMP, used by LINK, 359
Environment-string table, 273
EOF function, 102, 119, 133, 273, 303
EQV operator, 7
ERASE statement, 273
Erasing data, preventing, 102
ERDEV$ function, 273
ERL function, 273, 308
ERR
  code, 392
  function, 227, 248, 273, 308
ERRDEV function, 308
ERRDEV$ function, 308
Error codes, 227, 273, (table) 392
Error messages
  compile-time, 391
  described, 391
  invocation, 391
  LIB, 431
  LINK, 420
  numbers, limits (table), 338
  redirection, 324
  run-time, 391
ERROR statement, 273, 308
Error trapping
  activating, 226
  ERR, identifying errors with, 227
  ERROR statement, 273
  error-handling routine, 226–227
  file-access errors, 248
  multiple modules, 242–243, 245
  Quick libraries, 244
  screen modes, inappropriate, 150
  statements and functions (table), 308
  SUB or FUNCTION procedures, 241
  syntax, event-trapping syntax, contrasted with, 233
Error-handling routines
  ERR, identifying errors with, 227
  parts, 226
  specifying, 226

Error-handling statements
See also individual statement names
ERDEV, 273
ERR, ERL, 273
ERROR, 273
ON ERROR, 282
RESUME, 287
Error-message window, 325
Event polling, event trapping, contrasted with, 232
Event trapping
See also specific event
activating, 233
background music, 239
described, 231
disabling, 234
keystrokes, 235
multiple modules, 242
options, command-line, 247
polling, contrasted with, 232
routines, event-handling, 233
statements and functions, summary of (table), 308
SUB or FUNCTION procedures, within, 241
suspending, 234
syntax, error-trapping syntax, contrasted with, 233
trappable events, 233
types of
COM, 233
KEY, 233
PEN, 233
PLAY, 234
STRIG, 234
TIMER, 234
Event trapping, Quick library, 382
Event-handling routine, 233
Event-trapping option, 327
Event-trapping statements
See also individual statement names
COM, 269
KEY LIST, 278
KEY OFF, 278
KEY ON, 278
KEY(*n*), 278
ON *event*, 282
ON UEVENT, 282
PEN ON, OFF, and STOP, 284
PLAY ON, OFF, and STOP, 284
STRIG, 292
TIMER ON, OFF, and STOP, 293
UEVENT, 294
Executable files
compact, 389
packing, 364
/EXEPACK option (LINK), 364
EXIT DEF statement, 274, 300, 333
EXIT DO statement, 274, 298, 333
EXIT FOR statement, 23, 274, 298, 333
EXIT FUNCTION statement, 274, 299, 333
EXIT statement, 274, 333
EXIT SUB statement, 274, 299, 333
Exiting, functions and procedures, 299–300
EXP function, 274
Expressions
Boolean, 6
false, 7
lists of, printing, 83
procedures, passing to, 50
string *See* Strings
true, 7
Extensions, file-name, 352
Extra files produced by QuickBASIC, 385

F

Factorial function, 71
False expressions, 7, 102
FIELD statement
described, 274
random-access records, defining, 109
TYPE...END TYPE, contrasted with, 109
Fields
defined, 95
records
random-access, 108
sequential, 100
sequential files, delimited in, 100
File
names
characters, allowable, 98
OPEN statement, 98
numbers
CLOSE, freeing with, 99
FREEFILE, getting with, 97
freeing with CLOSE, 99
OPEN statement, 97
values, allowable, 97
pointers, 115–116
File access
LOCK statement, 280
UNLOCK statement, 280, 294
File conversion, 270

File handling
functions
See also individual function names
EOF, 273
FILEATTR, 274
FREEFILE, 275
LOC, 279
LOF, 280
SEEK, 289
statements
See also individual statement names
CHDIR, 268
CLOSE, 268
FIELD, 274
GET, 275
INPUT #, 277
KILL, 278
LOCK, 280
NAME, 282
OPEN, 283
RESET, 287
SEEK, 289
UNLOCK, 280
File I/O
defined, 95
device I/O contrasted with, 119
File names
case sensitivity, 99
described, xxiv
restrictions, 98
truncation, 98
File-access modes
APPEND, 97, 103
BINARY, 97, 115
INPUT, 97, 102
OUTPUT, 97
RANDOM, 97
File-name extensions, 352
File-naming conventions
BASIC programs, 98
BC command, 352
DOS, 98
LINK, 358
Quick libraries, 384
FILEATTR function, 274, 303, 334
Files
$INCLUDE, 347–348
ASCII format, 311
attributes, 274
extra, produced by QuickBASIC, 385
length, LOF, 280
limits (table), 337
Files (*continued*)
map (LINK), 365, 367
random-access, statements and functions
See also individual statement and function names, 274
FIELD, 274
LSET, 281
PUT, 286
RSET, 288
sequential, statements and functions
See also individual statement and function names, 277
INPUT #, 277
LINE INPUT #, 279
PRINT #, 285
WRITE #, 296
versions, compatibility among, 336
FILES statement, 274, 302
Filling. *See* Painting
FIX function, 274
Fixed-length strings
parameter lists, 49
record elements, 137
variable length, contrasted with, 140
variables, stand-alone, 137
Floating point, precision within Quick libraries, 383
FOR...NEXT loops
described, 19
execution, pausing, 24
exit from, alternative, 274
nesting, 22
STEP keyword, 19
FOR...NEXT statement, 24, 274, 298
Foreground color
default, 152
screen mode 1, 174
Formatting text output, 85
Forward reference problem, 55
Forward slash (/), LINK option character, 361
Fractal, 212
FRE function, 275
FRE statement, 72
FREEFILE function, 97, 275, 302, 334
Function keys, trapping, 235
FUNCTION procedures
calling, 45
DECLARE statements, 271
DEF FN function, contrasted with, 42
described, 44, 299, 334
error trapping in, 241
event trapping in, 241
exit from, alternative, 274

FUNCTION statement
AS clause, 332
described, 275
include files, restrictions with, 59, 348
STATIC attribute, 335
FUNCTION...END FUNCTION statements. *See* FUNCTION procedures
Functions, user defined, 271. *See individual function names*

G

GET statement
described, 275, 302
file I/O
binary-access, 115
random-access, 113–114, 126
graphics
animation, 200, 307
array, calculating size of, 195
memory, copying images to, 194
syntax, 194
records, user-defined, 334
Global variables
DEF FN function, 42
FUNCTION procedure, 42
GOSUB routine, 41
SHARED attribute, 64
variable aliasing, 68
GOSUB statement, 276, 300
GOSUB subroutines
drawbacks in using, 41
SUB procedures, compared with, 40
GOTO statement
$INCLUDE metacommand, 348
described, 276
Graphics
functions
See also individual statement names, 285
PALETTE USING, 176
PMAP, 171, 285
POINT, 171, 285
VIEW, 165
WINDOW, 168
statements
See also individual statement names, 266
BLOAD, 266, 312
BSAVE, 266, 312
CIRCLE, 158, 268
COLOR, 174, 269
DRAW, 272
GET, 275
Graphics (*continued*)
statements (*continued*)
LINE, 154, 279
PAINT, 283
PALETTE, 283
PALETTE USING, 283
PRESET, 285
PSET, 152, 286
PUT, 200, 286
VIEW, 295
WINDOW, 296
Graphics cursor. *See* Cursor, graphics
Graphics screen modes. *See* Screen modes
Graphics statements and functions, summary (table), 306
Graphics viewport. *See* Viewport, graphics
GW-BASIC, 311

H

/H option (QB), 328
/HELP option (LINK), 363
HEX$ function, 276
Hexadecimal numbers, 183, 276
High-resolution-display option (QB), 328

I

I/O
devices, from and to, 118
files, from and to, 95
ports, 283
standard, 82, 88
IEEE format
accuracy, 318
converting to, 317, 319
/MBF option, using with old programs, 319
numbers
converting from, 282
converting to, 270
exponential, printing, 318
Microsoft Binary, differences from, 318
printing, 318
ranges, 318
IF...THEN...ELSE statement
block form, 11–12
described, 276, 298
SELECT CASE statement, contrasted with, 13
single-line form, 9
/IGNORECASE option (LIB), 375
Ignore case
LIB, 375

Ignore case (*continued*)
LINK, 368
Images
GET, saving to memory with, 194
PUT, copying to screen with, 196
Immediate window, limits (table), 338
IMP operator, 7
$INCLUDE metacommand
COMMON, 73
description, 347
procedure declarations, 57, 60
restrictions, 348
Include files
COMMON, 73
nesting limits (table), 338
procedures
declaring, 57
not allowed, 329
statements not allowed, 59
/INFORMATION option (LINK), 363
INKEY$ function, 91, 276, 301
INP function, 277
INPUT # statement
described, 277, 302
example, 102
INPUT$, contrasted with, 106
LINE INPUT #, contrasted with, 105
Input functions
See also individual statement names
COMMAND$, 269
INKEY$, 276
INP, 277
INPUT$, 277
Input list, 89
Input past end error, 104
INPUT statement
carriage-return–line-feed sequence, suppressing
after input, 90
defined, 88
described, 277, 301
error message, invalid-input, 89
example, 121
FIELD statement, 274
LINE INPUT, contrasted with, 90
prompting, 89
variable list, format of, 89
Input statements
See also individual statement names
DATA, 270
INPUT, 277
INPUT #, 277
LINE INPUT, 279
INPUT statement (*continued*)
LINE INPUT #, 279
READ, 286
RESTORE, 287
WAIT, 295
INPUT$ function
data, reading
communications device, 121
files, 106
standard input, 91, 107
described, 277, 301–302
example, 121
INPUT #, contrasted with, 106
LINE INPUT #, contrasted with, 106
modem, communicating with, 133
Input/Output. *See* I/O
Insert mode, differences among versions, 323
INSTR function, 141, 147, 277
INT function, 277
INT86, INT86X replacements
CALL INT86OLD statements, 267
CALL INTERRUPT statements, 267
INT86OLD, 384
Integers
converting to, 268, 274, 277
FIX function, 274
limits (table), 337
Interpreted BASIC, compatibility, 311
INTERRUPT, 384
Interrupt support routines, 384
Invocation error messages, 391
IOCTL statement, 278
IOCTL$ function, 277
Italic text, showing placeholders, xxv

J

Joysticks, 291–292

K

KEY LIST statement, 278
KEY OFF statement, 278
KEY ON statement, 278
Key trapping
activating, 235
keys
DIRECTION, 235
function, 235
user-defined, 236–237
KEY(*n*) statements, 278
KYBD, 118

Keyboard scan codes, 339, 341, 343
Keyboard, reading input from, 88
Keys
  DIRECTION, key trapping, 235
  ENTER, 325
  SPACEBAR, 325
  debugging, differences among versions, 330
  editing, differences among versions, 327
Keywords, format in program examples, xxvi
KILL statement, 278, 302

L

/L option (QB), 382, 384
Labels, xxvii
Language differences among versions, 330
Last point referenced, 154, 171
LBOUND function, 53, 278
LCASE$ function, 145, 278, 304
LEFT$ function, 142, 279, 304
LEN function
  described, 110, 279, 305, 334
  sample program application, 112, 147
  use of, 137–138
LET statement, 279
LIB
  See also Libraries, stand-alone
  case sensitivity, 375
  command symbols
    asterisk (*), 373
    minus sign-plus sign (–+), 373
    plus sign (+), 373
  default responses, 372
  described, 369
  invoking, 370
  libraries, combining, 373
  library modules, 373
  listing files, 372
  options
    /IGNORECASE (/I), 375
    /NOEXTDICTIONARY (/NOE), 375
    /NOIGNORECASE (/NOI), 375
    /PAGESIZE (/P), 375
    case sensitivity, 375
    dictionary, no extended, 375
    page size, specifying, 375
  response file, 370
  search path, 359, 383
LIB error messages, 431
LIB.EXE, 350
Libraries
  See also Quick libraries
Libraries (*continued*)
  default, ignoring, 359, 364
  described, 377
  LINK, specifying for, 359
  search path, 359
  stand-alone
    combining, 373
    defined, 377
    described, 385
    LIB command line, 372
    listing, 372
    modules, extracting and deleting, 373
    object modules, deleting from, including, and replacing, 373
    parallel libraries, creating, 386
  types, contrasted, 377
Library Manager. *See* LIB
Limits, file size and complexity (table), 337
/LINENUMBERS option (LINK), 367
LINE INPUT # statement
  described, 279, 302
  INPUT #, contrasted with, 105
  INPUT$, contrasted with, 106
LINE INPUT statement
  described, 279, 301
  INPUT, contrasted with INPUT, 90
Line printer, 280
LINE statement
  coordinate pairs, order of, 154
  described, 279, 306
  lines and boxes, drawing, 156–157
  sample applications, 207, 212, 218
  syntax, 153
Line styling, 157
Lines, drawing, 153
LINK
  command line, invoking from, 355
  defaults, 357
  libraries, specifying , 359
  options
    See also Libraries
    /BATCH (/B), 363
    /CODEVIEW (/CO), 367
    /CPARMAXALLOC (/CP), 368
    /DOSSEG (/DO), 368
    /EXEPACK (/E), 364
    /HELP (/HE), 363
    /INFORMATION (/I), 363
    /LINENUMBERS (/LI), 367
    /MAP (/M), 365
    /NODEFAULTLIBRARYSEARCH (/NOD), 359, 364
    /NOIGNORECASE (/NOI), 368

LINK (*continued*)
options (*continued*)
/NOPACKCODE (/NOP), 364
/PACKCODE (/PAC), 367
/PAUSE (/PAU), 363
/QUICKLIB (/Q), 364
/SEGMENTS (/SE), 365
/STACK (/ST), 368
abbreviations, 362
arguments, numerical, 362
BC, unsuitable for, 369
case sensitivity, 361, 368
CodeView debugger, debugging with, 367
command line, order on, 361
default libraries, ignoring, 359, 364
displaying, 363
information, displaying, 363
line number, displaying, 367
LINK environment variable, 362
linker prompting, preventing, 363
map file, 365
paragraph space, allocating, 368
pausing, 363
Quick libraries, creating, 364
segments, 365
segments, ordering, 368
stack size, setting, 368
output file, temporary , 359, 363
Quick libraries, other language routines, 386
response file, 355, 358
LINK environment variable, 362
LINK error messages, 420
LINK.EXE, 350
Linker. *See* LINK
Linking. *See* LINK
Linking from DOS, 350
Listing files, cross-reference (LIB), 372
Literal constants, string, 136
Loading Quick libraries, 382
LOC function
described, 279, 303
devices, with, 119
modem, communicating with, 133
SEEK, contrasted with, 116
Local variables, 69
LOCATE statement
cursor
changing shape of, 93
positioning, 93
described, 280, 301
example, 122
LOCK statement, 280
LOF function
described, 280, 303
devices, 119
example, 126
files, 112
LOG function, 280
Logical operators, 7
Long integers
advantages of, 321
converting to, 268, 332
limits (table), 337
Looping structures
defined, 19
DO...LOOP, 26
FOR...NEXT, 19
WHILE...WEND, 25
Lowercase letters
file names, 99
uppercase, converting to, 145
LPOS function, 280
LPRINT statement
described, 280
SPC function, 290
LPRINT USING statement, 280
LPT devices, 118–119
LSET statement, 110, 281, 304, 334
.LST files. *See* Listing files
LTRIM$ function, 304
blanks, leading, printing numbers without, 146
described, 281, 334
spaces, leading, stripping, 139
spaces, leading, stripping , 144

M

Main module, 252
.MAK files, Quick libraries, use with, 383
Make Library command, 381–382
Mandelbrot set, 212
.MAP files, 365
Map files (LINK), 365–367
Math functions
ABS, 266
ATN, 266
COS, 270
CVSMBF, 270
EXP, 274
LOG, 280
MKSMBF$, MKDMBF$, 282
SIN, 290
SQR, 291
TAN, 293

/MBF option
BC, 318–319, 354
QB, 318–319, 328
Memory
available, calculating, 388
requirements, Quick libraries, 388
Memory functions, 284–285
Memory management
functions
See also individual function name, 275
FRE, 275
SETMEM, 289
statements
See also individual function name, 268
CLEAR, 268
DEF SEG, 271
ERASE, 273
PCOPY, 283
Menu commands, differences among versions, 325
Metacommands
$DYNAMIC, 348
$INCLUDE, 347.
See also $INCLUDE metacommand
$STATIC, 348
described, 347
Microsoft Binary format
IEEE format, differences from , 318
numbers, 270, 282
Microsoft Library Manager. *See* LIB
MID$ function, 144, 147, 281, 304
MID$ statement, 147, 281, 304
Minimize String Data option, 328
Mixed-language programming
ALIAS, use of, 271
CALL, CALLS statement (non-BASIC), 267
CDECL, use of, 271
DECLARE statement (non-BASIC), 271
Quick libraries, 257, 385
MK*type* statement, 305
MK*type*$ statement, 109
MK*type*MBF$ statement, 109
MKD$ function, 281
MKDIR statement, 281
MKDMBF$ function, 320, 335
MKI$ function, 281
MKL$ function, 281, 334
MKS$ function, 281
MKSMBF$ function, 282, 320, 335
Modem, communicating with, 133
Module-level code, 252
Modules. *See* Multiple modules
Monochrome screen mode, 172
Moving images with PUT, 196
Multiple modules
advantages, 251
error trapping, 242–243, 245
event trapping, 242
loading, 254
main module, 252
programming style, 259
Quick libraries, 257
size limits (table), 338
variables, sharing, 66, 256
versions, differences among, 323
Music
background, 239, 284
statements, 284
Music event trapping
ON PLAY GOSUB statement, 239
PLAY ON statement, 239

N

NAME statement, 282, 302
Named COMMON block, 67, 73
Names, devices, 365
New line. *See* Carriage-return–line-feed sequence
NEXT statement, 274, 336
No extended dictionary, library, 375
NOCOM.OBJ file, 357
/NODEFAULTLIBRARYSEARCH option (LINK), 359, 364
/NOEXTDICTIONARY option (LIB), 375
/NOIGNORECASE option (LIB), 375
/NOIGNORECASE option (LINK), 368
/NOPACKCODE option (LINK), 364
NOT operator, 7–8
Notational conventions, xxiv
NUL (device name), 356
NUL.MAP file, 357
Null string (""), 91
Numbers
positive, printing without leading blank, 146
random-access files, storage in, 108
screen, printing on, 83, 146
strings, representing as, 146
Numeric conversions
CVD function, 270
CVI function, 270
CVL function, 270
CVS function, 270
double-precision, 267
integer, 268, 274, 277
single-precision, 270

Numeric functions
See also individual function names
CDBL, 267
CINT, 268
CLNG, 268
CSNG, 270
CVD, 270
CVI, 270
CVL, 270
CVS, 270
FIX, 274
INT, 277
RND, 288
SGN, 289

O

/O option (BC), 354
Object
files, 336, 372
modules, 372–373
OCT$ function, 282
Octal conversion, 282
ON *event* GOSUB statements, 247
ON *event* statements, 282, 308
ON *expression* statement, 17
ON COM statement, 282
ON ERROR GOTO statement
described, 282, 308
example, 248
syntax, 226
ON KEY statement, 282
ON PEN statement, 282
ON PLAY GOSUB statement, 240
ON PLAY statement, 282
ON STRIG statement, 282
ON TIMER statement, 282
ON UEVENT statement, 282, 335
ON...GOSUB statement, 282
ON...GOTO statement, 282
OPEN COM statement, 120, 133, 283
OPEN statement
described, 283, 302, 335
file names in, 98
file numbers in, 97
file-access modes
APPEND, 97, 102–103
BINARY, 97, 115
INPUT, 97, 102
OUTPUT, 97, 101, 103
RANDOM, 97
LEN clause, 109
Operators
logical, 7
relational, 140
relational, (table), 6
OPTION BASE statement, 45, 283
Options. *See* entries for BC command, LIB, LINK
OR operator, 7, 197
OUT statement, 283
Output
functions
See also individual function names
LPOS, 280
POS, 285
TAB, 293
line width, 296
statements
See also individual function names
BEEP, 266
CLS, 269
LPRINT, 280
OUT, 283
PRINT, 285
PRINT #, 285
PRINT # USING, 285
PRINT USING, 286
PUT, 286
WRITE, 296
WRITE #, 296
Overlay linker. *See* LINK
Overtype mode, differences among versions, 323

P

/PACKCODE option (LINK), 367
/PAGESIZE option (LIB), 375
Page size, library, 375
Pages. *See* Screen pages
PAINT statement
argument
background, 187
border, 180, 187
described, 283, 307
numeric expression, 179
See also Painting
shapes, filling
colored, 179
patterned, 181, 207, 218
string expression, 181
tiling
monochrome, defining, 181
multicolor, defining, 188

Painting
colors, 179–180
exterior, 179
interior, 179, 182
patterns
described, 181, 207, 218
multicolor, 188
PALETTE statement, 176, 283, 306, 333
PALETTE USING statement, 176, 212, 283
Palettes
COLOR, changing with, 174
PALETTE, changing with, 176
screen mode 1, 174
Paragraph space, 368
Parameter list, defined, 55
Parameters
arguments
agreement with, 48, 55
distinguished from, 48
DECLARE, format in, 55
number and type, declaring, 55
Passing by reference, 60
Passing by value
DEF FN, used in, 43
defined, 61
Path names, limits (table), 338
Pattern tiles, editing on screen, 218
Patterns
monochrome, 182
multicolor, 188
shapes, filling, 181, 207, 218
tile size, 181
/PAUSE option (LINK), 363
Pausing program execution, FOR...NEXT statement, 24
PCOPY statement, 283, 307
PEEK function, 284
PEN function, 284
PEN OFF statement, 284
PEN ON statement, 284
PEN STOP statement, 284
Peripheral devices, 118
Physical coordinates, 171, 285
Pie shapes, drawing, 162
Pixels
coordinates, locating with, 151
defined, 151
PRESET, plotting with, 152
PSET, plotting with, 152
text cursor, 93
Place markers, limits (table), 338
PLAY function, 284
PLAY OFF statement, 284
PLAY ON statement, 240, 284
PLAY statement
background music option, 239
described, 284
QuickBASIC/interpreter differences, 312
VARPTR$ function, using, 295
PLAY STOP statement, 284
Plus (+)
operator, combining strings, 138
sign, LIB command symbol, 373
PMAP function, 171, 285, 306
PMAP statement, 212
POINT function, 171, 285, 306
POKE statement, 285
Polling, 232
POS function, 94, 122, 285, 301
Positive numbers, printing without leading blank, 146
PRESET option, with PUT graphics statement, 197
PRESET statement, 306
color option, using with, 152
described, 152, 285
PRINT # statement
described, 285, 302
record fields, delimiting, 104
WRITE #, contrasted with, 103
PRINT # USING statement, 285
PRINT statement
described, 83, 285, 301
example, 122
SPC function, 290
text, wrapping, 84
PRINT USING # statement, 302
PRINT USING statement, 85, 122, 286, 301
Print zone, 83, 103
Printing
numbers
negative, 83
positive, 83, 146
text
printer, to, 119
screen, to, 83, 119
PRN (device name), 356
Procedures
arguments
passing by reference, 60
passing by value, 61
arrays, 49, 52
benefits, 40
calling, 45, 255
constants, 49
defining, 44
described, 275

Procedures (*continued*)
expressions, 50
format
argument-list, 49
parameter-list, 44, 49
include files, declarations in, 255
libraries, user , 252
limits (table), 338
modules, multiple, 55, 251
moving, 254
passing arguments by reference, 60
Quick libraries, 60
records, 49, 53
recursive, 71, 75
statements not allowed, 45
statements
FUNCTION...END FUNCTION, 44–45
SUB...END SUB, 44, 46
summary (table), 299
STATIC variables, 44, 69
variables, automatic, 44, 69
variables
global, 64
local, 42, 69
sharing, 256
Procedures-only module, 254
Program
suspension, 290
termination, 272
Programming environment, differences in versions, 324
Programming style, xxvi
Programming using multiple modules, 259
Programs, BASICA, GW-BASIC, converting to QuickBASIC, 311
ProKey, using QuickBASIC with, 323
PSET option, 197
PSET statement
color option, 152
described, 152, 286, 306
example, 212
STEP option, 154
PUT statement
buffer allocation, FIELD statement, 274
described, 200, 286, 302
file I/O
binary-access, 114–115
random-access, 111–112, 126
graphics
animation, 200, 307
images, copying to screen, 196
interaction with background, controlling, 197
PUT statement (*continued*)
graphics (*continued*)
syntax, 196
records, user-defined, 334

Q

QB command
/AH option, 328
/H option, 328
/L option, 382, 384
/MBF option, 318–319, 328
/RUN option, 328, 383
versions, differences among, 328
QB.QLB
library, 383
loading, automatic, 384
QLBDUMP.BAS, 384
Quick libraries
advantages, 378
CALL ABSOLUTE statement, 384
CALL INT86OLD statement, 384
CALL INTERRUPT statement, 384
compatibility among versions, 336
compilation, 257
contents
described, 257–258, 378
listing, 384
reading, 116
creating
command line, from, 385
described, 258, 378
files needed, 380
LINK, using, 364
QuickBASIC, from within, 380
default, 383–384
described, 377
end user, delivery to, 385
errors, trapping in, 244
executable files, making compact, 389
files
needed to create, 380
produced by, 385
floating-point precision, 383
include files, declaring procedures with, 60
loading, 382, 384
.MAK file, updating, 383
memory requirements, 388
mixed languages, 257
naming, 382, 384
object code, linking, 258
routines, other languages, 380

Quick libraries (*continued*)
search path, 383
updating previous, 380
use of, 257
/QUICKLIB option (LINK), 364

R

/R option (BC), 328, 354
Radians, 160
Random access, contrasted with binary access, 115
Random numbers, 286, 288
Random-access files
creating, 97
number storage, 108
opening, 97, 108
records
adding, 111
organizing, 108
reading, 113
sequential files, contrasted with, 96, 108
statements and functions
See also individual statement and function names
FIELD, 274
GET, 275
LSET, 281
PUT, 286
RANDOMIZE statement, 286
READ statement, 286
Record numbers
limits (table), 338
random-access files, indexing in, 114
Record numbers, random-access files, indexing in, 125
Records
binary-access files, writing, 115
data, overwriting, 96
defined, 95
defining, 109
described, 109
fixed-length, 97, 113
procedures, passing to, 49, 53
random-access files
adding to, 111
appending, 112
reading, 113
storing, 108
writing, 108
sequential files
appending, 96, 103
reading, 102
Records (*continued*)
sequential files (*continued*)
storing, 100
variable length, 100, 281
Rectangles, specifying coordinates, 156
Recursive procedures, 71–72
REDIM statement
described, 286
SHARED attribute, 62, 64, 299
Relational operators
Boolean expressions, 6
SELECT CASE statement, 15
string comparisons, 140
Relative coordinates. *See* Coordinates, relative
REM statement, 287
RESET statement, 287
Response file
example, 358
LIB, 370
LINK, 355
RESTORE statement, 228, 287
Resume Next option, 328
RESUME NEXT statement
described, 287
example, 248
RESUME, contrasted with, 229
RESUME statement
compiler option required, 247
described, 287, 308
example, 248
QuickBASIC/interpreter differences, 312
RESUME NEXT, contrasted with, 229
RETURN statement, 287, 308
RIGHT$ function, 143, 287, 304
RMDIR statement, 287
RND function, 288
Rotating figures with DRAW, 206
Routine, B_OnExit, 387
Rows, changing number of, 86
RSET statement, 110, 288, 304
RTRIM$ function, 139, 143, 288, 304
/RUN option (QB), 328, 383
Run menu, Make Library command, 381–382
RUN statement
data files, closing, 99
described, 288, 312
Run-time error messages, 391

S

/S option (BC), 328, 354
SADD function, 288

SAVE statement (BASICA), 311
Saving
  images, with GET, 194
  programs, ASCII format, 311
Scan codes
  keyboard, 339, 341, 343
  trapping keys, use in, 236
Screen
  configuration
    graphics mode, 151
    text mode, 82
  functions
    See also individual function names
    CSRLIN, 270
    POS, 285
    SCREEN, 288
  resolution, SCREEN statement, 151
  statements
    See also individual statement names
    CLS, 269
    COLOR, 269
    LOCATE, 280
    PCOPY, 283
    SCREEN, 288
    VIEW PRINT, 295
    WIDTH, 296
SCREEN function, 288
Screen modes, 150–151
Screen pages, 205
SCREEN statement
  aspect ratio, effect on, 164
  described, 288, 306, 333
  example, 207, 212
  screen
    page, 205
    resolution, adjusting, 151
  text mode, rows in, 86
SCRN, 118–119
Scrolling, 86
Search paths
  libraries, 359
  Quick libraries, 383
Searching
  binary, 129
  strings, 141
SEEK function, 116, 289, 303
SEEK statement, 96, 116, 289, 303, 335
Segments, 365
  lists, map files, 365
  number allowed, 365
  order, 368
  packing, 364
SELECT CASE statement
  CASE clause, 15
  CASE ELSE clause, 16
  described, 289, 298
  END SELECT clause, 15
  IF...THEN...ELSE statement, contrasted with, 13
  ON *expression* GOSUB, contrasted with, 17
  routine, error handling, 227
  syntax, 14
  versions, differences among, 335
Semicolon (;), LIB command symbol, 84, 90, 372
Separate-compilation method. *See* BC command
Sequential files
  adding data to, 103
  creating, 96
  fields, 100
  opening, 96–97, 102–103
  random-access files, contrasted with, 96, 108
  records, 100, 102
  statements and functions
    See also individual statement and function names
    INPUT #, 277
    LINE INPUT #, 279
    PRINT #, 285
    WRITE #, 296
Serial communication
  defined, 120
  input buffer, 120–121
Serial ports, opening for I/O, 120
Set Main Module command, differences among versions, 326
SETMEM function, 289, 335
SGN function, 289
SHARED attribute
  COMMON, 62, 64, 66
  DIM
    described, 62, 299
    example, 68
    prohibited, 45
    sharing, 64
  REDIM, 62, 64, 299
SHARED statement, 62, 290, 299, 332
Shared variables, between modules, 269
Shell escape (SHELL statement), 290
Shifted keys, trapping, 237
SideKick, using QuickBASIC with, 323
SIN function, 290
Sine, SIN function, 290
Single-precision numbers
  converting to, 270
  size limits (table), 337

Skipping
  columns, 86
  spaces, 85
SLEEP statement, 290, 335
Sorting, examples, 8, 33, 131
SOUND statement, 290
Source files
  format, 311
  versions, compatibility among, 336
SPACE$ function, 145, 290, 305
Spaces
  skipping, 85
  trimming, 139, 143–144
SPC function, 290, 301
SPC statement, 85–86
SQR function, 291
Squares, drawing, 164
/STACK option (LINK), 368
Stack size, recursive procedures, adjusting for, 72
Stack size, setting, 368
Stand-alone programs, creating outside QuickBASIC environment, 350
Standard I/O
  defined, 82, 88
  statements used in BASIC (table), 301
Standard input
  defined, 88
  reading, 88, 90–91
  redirecting, 88
Standard output, 82
Standard places, libraries, 359
Statement, modification required, 312
Statement block, 6
Statements. *See individual statement names*
STATIC
  arrays, dimensioned, implicitly, 348
  arrays, memory allocation, 348
  attribute, 335
  statement, 69, 291, 300
  variables, 69
Static arrays
  STATIC metacommand, 348
  ERASE statement, 273
STEP option, 154
STICK function, 291
Stop bits, 120
STOP statement, 291
STR$ function, 146, 291, 304
STRIG function, 291
STRIG OFF statement, 291
STRIG ON statement, 291
STRIG(*n*) statements, 292
String
  expressions
    defined, 136
    sequential files, delimiting in, 103
  functions
    See also individual function names, 294
    ASC, 266
    CHR$, 268
    DATE$, 270
    HEX$, 276
    INPUT$, 277
    INSTR, 277
    LCASE$, 278
    LEFT$, 279
    LEN, 279
    LTRIM$, 281
    MID$, 281
    RIGHT$, 287
    RTRIM$, 288
    SADD, 288
    SPACE$, 290
    STR$, 291
    STRING$, 292
    UCASE$, 294
    VAL, 294
  processing. *See* Strings
  statements
    See also individual statement names
    LSET, 281
    MID$, 281
    RSET, 288
  variables, 136, 279
    See also Strings
STRING$ function, 145, 292, 305
String-handling functions, new, 334
Strings
  alphabetizing, 140
  characters, retrieving
    left side, 142
    middle, 144
    right side, 143
  combining, 138
  comparing, 7, 140
  constants, 136
  defined, 135
  expressions, 136
  fixed-length
    AS STRING, 136
    record elements, 137
    variable length, contrasted with, 140
  generating, 145
  limits (table), 337–338

Strings (*continued*)
numbers, representing as, 146
replacing, 147
spaces, trimming
left side, 144
right side, 143
statements and functions, summary of (table), 304
substring, searching for, 141
variable-length
AS STRING, 136
defined, 137
fixed-length, contrasted with, 140
maximum size, 137
variables, 136
SUB
procedures
$INCLUDE metacommand, 348
CALL, 47
DECLARE statements, 271
error trapping in, 241
event trapping in, 241
exit from, alternative, 274
GOSUB, compared, 40
include files, placing in, 329
variables, local, 69
statement
AS clause, 332
described, 292, 299
include files, not allowed in, 59
STATIC attribute, 335
SUB...END SUB procedure. *See* SUB procedures
Subprograms
CALL, CALLS statements, 266–267
SUB statement, 292
variables, 290–291
See also SUB procedures
Subroutines, 282, 287
See also GOSUB...RETURN subroutine
Subscripts
arrays, limits (table), 337
lower bound, specifying, 283
maximum value, specifying, 272
number, specifying, 272
upper bound for, 294
SuperKey, using QuickBASIC with, 323
SWAP statement, 292
Symbol tables, in map files, 366
Symbolic constants
CONST, 269
defined, 333
format in program examples, xxvi
string, 136
Syntax checking command, differences among versions, 322
Syntax notation
choices, xxv
optional items, xxv
placeholders, xxv
System calls, 267
SYSTEM statement, 292

T

TAB function, 293, 301
TAB statement, 86, 122
Text
screen, printing on, 83
wrapping, 84
Text boxes, limits (table), 338
Text viewport. *See* Viewport
Tiling, 181
See also Painting
Time and date functions. *See* Date and time, functions
TIME$ function, 293
TIME$ statement, 293
TIMER function, 293
TIMER OFF statement, 293
TIMER ON statement, 293
TIMER STOP statement, 293
Timing function, 293
TMP environment variable, LINK, used by, 359
Trapping
See also Error trapping, Event trapping
errors, 225
events, 231
multiple modules, 242
options, command-line, required by compiler, 246
Trigonometric functions
ATN, 266
COS, 270
SIN, 290
TAN, 293
TROFF statement, 293
TRON statement, 293
True expressions, 7, 102
TYPE command (DOS), 103
TYPE statement
described, 294
versions, differences among, 336
TYPE...END TYPE statement
example, 126
FIELD, contrasted with, 109
fixed-length strings in, 137
random-access records, defining, 109

Typeface
key names, xxv
keywords, xxiv
placeholders, xxv
program, xxiv
Typographic conventions, xxiv

U

UBOUND function, 53, 294
UCASE$ function, 145, 294, 304, 334
UEVENT statement, 294, 336
UNLOCK statement, 280, 294
Uppercase letters
file names, 99
lowercase, converting to, 145
User libraries, 329
See also Quick Libraries
User-defined
data types, 294
events, 282, 294
types, 317
Utility, BUILDLIB, 329

V

/V compiler option
described, 247, 354
versions, differences among, 328
VAL function, 146–147, 294, 304
Variable aliasing, 68
Variable-length strings
defined, 137
fixed-length, compared with, 140
GET statement, 115
maximum size, 137
PUT statement, 115
Variables
arrays
described, 272
elements of, passing, 52
entire, passing, 52
automatic, 44, 69
data type, 272
global
described, 41–42
function definitions, 291
sharing, 64
subprograms, 290–291
variable aliasing, 68
local
described, 69
Variables (*continued*)
local (*continued*)
function definitions, 291
subprograms, 290–291
multiple modules, sharing, 66, 256
names
limits (table), 337
program examples, format in, xxvi
procedures
passing to, 60–61
sharing all in a module, 62, 64
programs, sharing, 73
simple, passing, 51
STATIC, 44, 69
string, 135, 136, 279
type, declaring, 51
values, exchanging, SWAP, 292
VARPTR function
described, 295
versions, differences among, 336
VARPTR$ function
described, 295
DRAW, use with, 312
PLAY, use with, 312
VARSEG function
described, 295
versions, differences among, 336
Version differences, file compatibility, 336
VGA (Video Graphics Adapter), changing palette, 176
View coordinates
physical coordinates
translating to, 171
mapping to, 285
WINDOW, defining with, 168, 296
VIEW PRINT statement, 86, 295, 301
VIEW SCREEN statement, 166
VIEW statement, 165, 212, 218, 295, 306
Viewport
graphics
advantages, 165
VIEW SCREEN, defining with, 166
VIEW, defining with, 165
text, 86
VM.TMP file, 360

W

/W option (BC)
described, 247, 354
versions, differences among, 327
WAIT statement, 295
Watch expressions, limits (table), 338

Watchpoints, limits (table), 338
WEND statement
  described, 295
  versions, differences among, 336
WHILE statement, 295
WHILE...WEND statement
  described, 298
  DO...LOOP, contrasted with, 26
  syntax, 25
WIDTH statement
  columns, changing number of, 86
  described, 296, 301–302, 333, 336
  rows, changing number of, 86
WINDOW SCREEN statement, WINDOW contrasted with, 168
WINDOW statement
  coordinates, 168
  described, 296, 306
  example, 212
  GET, effect on, 195
Windows
  error-message, 325
  Immediate, limits (table), 338
  versions, differences among, 325
WordStar keys, similarity to, 328
Wrapping text, 84
WRITE # statement
  described, 296, 302
  PRINT #, contrasted with, 103
WRITE statement, 296

X

/X option
  described, 247, 355
  differences among versions, 328
XOR operator, 7
XOR option, PUT graphics statement, 197

Z

/ZD option (BC), 355
/ZI option (BC), 355

Documentation Feedback – QuickBASIC 4.5

Help us improve future documentation. When you become familiar with our product, please complete and return this form to the address given on the reverse side. Comments and suggestions become the property of Microsoft Corporation.

Operating system _____ Version no. _____
Computer: Brand _____ Model no. _____ Memory (K) _____
RAM disk ❑ Yes ❑ No Hard disk ❑ Yes ❑ No
Programming experience: Total years _____ Years in this language _____
How long have you had this product? _____ (months)

Please rate the quality of the following documentation features on a scale from 1 (poor) to 5 (excellent):

Rating *Comments*

____ Organization _____
____ Accessibility of information _____
____ Installation instructions _____
____ Examples _____
____ Index _____
____ Illustrations _____
____ Binding _____
____ Overall impression _____

If you found errors in the documentation, please give manual name, page number, and description:

Was it easy to understand the documentation?
❑ Difficult ❑ Average ❑ Easy
Comments:

Do you want more information on programming?
❑ No ❑ Yes On what topics?

Rate how useful the following components are:
(1 = not useful, 2 = somewhat useful, 3 = useful, 4 = very useful, 5 = extremely useful)
Rating
____ QB Advisor/On-Line Help
____ QB Express
Comments:

How easy was it to learn the following from QuickBASIC documentation?
(1 = difficult, 2 = somewhat difficult, 3 = average, 4 = somewhat easy, 5 = easy)
Rating
____ How to use the QuickBASIC environment
____ How to program in BASIC
____ How to use the QB Advisor

How do you divide your program development time (percent) between the following methods?
____ Percent within the QuickBASIC programming environment (editor, debugger)
____ Percent using own editor and compiling with the BC command

Which manual do you use most frequently?

For the manual you use most frequently, which chapters or sections do you use most often?

Use the back of this form for other suggestions and comments.

(Over)

Name

Address

City/State/Zip

Phone ( ) — (home) ( ) — (work)

Additional comments:

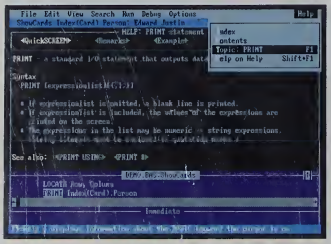
Please mail this form to:

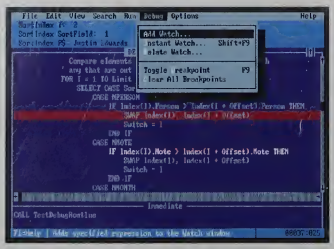
Microsoft Corporation
User Education
Data Access Business Unit

One Microsoft Way
Redmond, WA 98052-9953

Microsoft® QuickBASIC™

When you can't find the programs you want, don't get mad. Get Microsoft QuickBASIC version 4.5 and write them yourself. The Microsoft QuickBASIC system is the easiest way to learn how to program. And the industry's most innovative and comprehensive learning tools make it the fastest way, too!





Sometimes you just can't find the right software. Maybe it's a program that performs an unusual task or executes a common one in a different way—a special utility, a unique conversion program, or even a complete application. With Microsoft QuickBASIC 4.5, you can learn to do it yourself instead of doing without. And you can do it fast.

Microsoft QuickBASIC version 4.5 is a complete BASIC learning system. The new interactive, on-disk tutorial, Microsoft QuickBASIC Express, quickly and easily introduces you to the Microsoft QuickBASIC environment. A new step-by-step tutorial guides you through an actual application. And numerous example programs help you master BASIC programming.

Once you begin to write your own programs, Microsoft QuickBASIC 4.5 boosts your productivity with innovative technology. Microsoft QuickBASIC Advisor is a hypertext-based, electronic help system with *instant* cross-referencing that lets you quickly work through related topics as fast as you can click the mouse or press the F1 key. It's a *complete* reference at your fingertips—no more time-consuming hunting through pages of information. With all this help, there's no excuse for doing without the programs you want.

- Automatic setup makes installation easy—even on a floppy-disk-based system.
- Easy Menus simplify the menu-driven user interface.
- Superior integrated debugging helps you get your program up and running faster because you can see exactly what it's doing.
- No recompiling! Just stop your program, correct errors, and continue running.
- "Instant" compilation at 150,000 lines per minute* means no waiting for long compiles.

*On an IBM PC/AT® running at 8 MHz.

System Requirements

- 384K available user memory
- MS-DOS® or PC-DOS operating system version 2.1 or higher
- One 3½" disk drive or two double-sided 5¼" disk drives

Note:
Use of the Microsoft Mouse is optional.

Microsoft

Making it all make sense™

Microsoft, the Microsoft logo, and MS-DOS are registered trademarks and Microsoft QuickBASIC and *Making it all make sense* are trademarks of Microsoft Corporation.
© 1990 Microsoft Corporation. All rights reserved. Made in the United States of America. Includes non-U.S. recording media.

Version 4.5
1190 Part No.19982