MICROSOFT®

Microsoft® QuickBASIC

Compiler



Now with built-in full-screen editor.

For IBM® Personal Computers and Compatibles


Digitized by the Internet Archive
in 2023

https://archive.org/details/bwb_O9-AKU-166

Microsoft® QuickBASIC

Compiler

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 this software on magnetic tape, disk, or any other medium for any purpose other than the purchaser's personal use.

© Copyright Microsoft Corporation, 1985, 1986

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

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

Microsoft, the Microsoft logo, MS, MS-DOS, and GW-BASIC are registered trademarks of Microsoft Corporation. The High Performance Software is a trademark of Microsoft Corporation.

COMPAQ is a registered trademark of COMPAQ Computer Corporation.

IBM is a registered trademark of International Business Machines Corporation.

Document Number 990700003-200-R01-0486

Contents

Part 1 — User's Guide 1

1 Introduction 3

1.1 Welcome 5
1.2 System Requirements 6
1.3 Using This Manual 6
1.4 Notational Conventions 9
1.5 Key Names 12
1.6 Resources for Learning 12

2 Getting Started 15

2.1 Backing Up Your Disks 17
2.2 Disk Contents 17
2.3 Setting Up the Compiler 19
2.4 Starting QuickBASIC 21
2.5 Viewing the Help Window 21

3 Exploring QuickBASIC 23

3.1 If You Have a Mouse 25
3.2 The QuickBASIC Screen 26
3.3 Choosing Commands from Menus 28
3.4 Shortcut Command Keys 30
3.5 Selecting Options from Dialog Boxes 31
3.6 Making Selections 33
3.7 Scrolling 34
3.8 Practice Session 35
3.9 Using Debugging Mode 61
3.10 Creating a User Library 65

4 Compiling and Running a Program 69

4.1 Starting QuickBASIC 71
4.2 The File Menu 74
4.3 The View Menu 81
4.4 The Run Menu 84

5 Editing and Debugging 97

5.1 The QuickBASIC Editor 99
5.2 The Edit Menu 101
5.3 The Search Menu 104
5.4 Debugging Mode 109

6 Using Subprograms 113

6.1 Declaring Subprograms 115
6.2 Calling Subprograms 117
6.3 Passing Variables with CALL 118
6.4 Passing Arrays with CALL 119
6.5 Accessing Parameters with SHARED 121
6.6 Subprograms in User Libraries 125
6.7 Common Errors 126

7 Creating User Libraries 131

7.1 Building a User Library 133
7.2 Specifying a User Library 134
7.3 QuickBASIC User Library Routines 135

8 Interfacing with Assembly Language 137

8.1 Calling Assembly-Language Subroutines with CALL and CALLS 139
8.2 Coding Subroutines for the Compiler 143

8.3 Passing Arrays to Assembly-Language Subroutines 147
8.4 Calling System Services: Interrupt Support Routines 147
8.5 The Run-Time Memory Map 152

Part 2 — Reference 153

9 Language Elements 155

9.1 Character Set 157
9.2 The QuickBASIC Program Line 158
9.3 Data Types 161
9.4 Constants 162
9.5 Variables 165
9.6 Expressions and Operators 169
9.7 Type Conversion 178

10 Compiler-Interpreter Language Differences 181

10.1 New Programming Features 183
10.2 New BASIC Statements and Functions 186
10.3 Using Metacommands 187
10.4 Compiler-Interpreter Differences 189

11 Statement and Function Reference 195

Part 3 — Appendixes 521

A ASCII Character Codes 523

B QuickBASIC Reserved Words 525

C Differences between QuickBASIC Version 1.0 and Version 2.0 527

C.1 Compatibility between Version 1.0 and Version 2.0 529
C.2 Command Line 529
C.3 Options 529
C.4 IBM Enhanced Graphics Support Adapter 530
C.5 New Language Features 530
C.6 Using Microsoft® Word Files 531

D Using the Separate Compilation Method 533

D.1 Disk Setup 535
D.2 Compiling 538
D.3 Linking 540
D.4 Running a Program 544
D.5 The Run-Time Memory Maps 546

E Error Messages 549

E.1 Invocation Errors 551
E.2 Compile-Time Errors 552
E.3 Run-Time Errors 562

Glossary 571

Index 579

Figures

Figure 3.1 QuickBASIC Screen 27
Figure 3.2 Shortcut Keys 30
Figure 4.1 The File Menu 74
Figure 4.2 The Load... Dialog Box 76
Figure 4.3 The Save... Dialog Box 79
Figure 4.4 The Print... Dialog Box 80
Figure 4.5 The View Menu 82
Figure 4.6 The Options... Dialog Box 82
Figure 4.7 The Error-Message Window 84
Figure 4.8 The Run Menu 85
Figure 4.9 The Compile... Dialog Box 89
Figure 5.1 The Edit Menu 102
Figure 5.2 The Search Menu 104
Figure 5.3 The Find... Dialog Box 105
Figure 5.4 The Change... Dialog Box 107
Figure 5.5 The Debugging Mode Display 110
Figure 8.1 The Stack after CALL Executes 141
Figure 8.2 Memory Map during Program Execution 152
Figure 11.1 Default Pixel Coordinates 514
Figure 11.2 Coordinates After WINDOW (–10,–10)-(10,10) 515
Figure 11.3 Coordinates After WINDOW SCREEN (–10,–10)-(10,10) 516
Figure D.1 The BRUN20.EXE Run-Time Module Environment 546
Figure D.2 The BCOM20.LIB Run-Time Library Environment 547

Tables

Table 2.1 Disk Contents 18

Table 2.2 QuickBASIC Required Files 19

Table 3.1 QuickBASIC Shortcut Keys 31

Table 3.2 Scrolling 34

Table 4.1 QuickBASIC Compile Options 73

Table 5.1 The QuickBASIC Editor 100

Table 5.2 Selecting Text with a Mouse 101

Table 5.3 Debugging Mode Function Keys 112

Table 8.1 INT86 and INT86X Register Values 148

Table 9.1 Variable-Type Memory Requirements 167

Table 9.2 Relational Operators and Their Functions 173

Table 9.3 Values Returned by Logical Operations 174

Table 10.1 Functions and Statements Described in This Chapter 183

Table 10.2 New BASIC Statements and Functions 186

Table 10.3 Operational Differences 191

Table 10.4 Error-Handling Exceptions 192

Table 10.5 Statements and Functions Prohibited by the Compiler 192

Table 10.6 Statements Requiring Modification 193

Table 10.7 Enhanced Statements and Functions 194

Table 11.1 Bits per Pixel Values for Different Screen Modes 287

Table 11.2 Keyboard Scan Codes 316

Table 11.3 SCREEN Color and Attribute Ranges 385

Table 11.4 Default Attributes: SCREEN 10, Monochrome Display 450

Table 11.5 Color Values: SCREEN 10, Monochrome Display 450

Table 11.6 SCREEN Mode Specifications 452

Table 11.7 Default Attributes and Colors for Most SCREEN Modes 453

Table 11.8 Default Foreground Colors 454

Table D.1 Separate Compilation Method Required Files 536

Part 1
User's Guide

1 Introduction 3

2 Getting Started 15

3 Exploring QuickBASIC 23

4 Compiling and Running a Program 69

5 Editing and Debugging 97

6 Using Subprograms 113

7 Creating User Libraries 131

8 Interfacing With Assembly Language 137

Chapter 1
Introduction

1.1 Welcome 5
1.2 System Requirements 6
1.3 Using This Manual 6
1.4 Notational Conventions 9
1.5 Key Names 12
1.6 Resources for Learning 12

1.1 Welcome

Welcome to the Microsoft® QuickBASIC Compiler. QuickBASIC provides a complete development environment, including a program editor and enhanced debugging mode, designed to speed up the compile-debug-edit cycle. QuickBASIC also has an easy-to-learn, menu-oriented user interface, in which you can select commands from menus using the keyboard, an optional mouse, or both. Once you start QuickBASIC you can compile, run, debug, edit, and recompile programs without leaving QuickBASIC. During compilation, QuickBASIC "remembers" error locations, so that when compilation is finished a special search command can help you find and correct errors quickly.

QuickBASIC extends interpreted BASIC to help you write programs that run faster and are easier to maintain and debug. Here are some of the new features of QuickBASIC:

- **Subprograms.** Subprograms allow you to separate programs into distinct modules. You can use subprograms that perform frequently used tasks over and over in many different programs.
- **Structured programming support.** QuickBASIC's multiline **IF...THEN...ELSE** statement provides more flexibility than single-line **IF..THEN...ELSE**. It allows more complicated case structures while preserving the readability of your program.
- **Alphanumeric labels.** The compiler does not require line numbers and allows the use of alphanumeric labels. With alphanumeric labels, you can give your statements and subroutines descriptive names. This makes your program easier to read and can provide simpler debugging.
- **Flexible array dimensioning.** Dynamic arrays allow you to allocate memory for the array at run time, providing more efficient use of memory.
- **Large numeric arrays.** QuickBASIC allows as many 64K (kilobyte) numeric arrays as will fit into memory.
- **Metacommands.** Metacommands allow you to control dimensioned array allocation and processing of additional source files, from within a BASIC program.

1.2 System Requirements

The Microsoft QuickBASIC Compiler requires the following:

- An IBM® Personal Computer or "compatible" running MS-DOS or PC-DOS Version 2.0 or later
- At least one floppy-disk drive (two are recommended)
- 256K RAM available memory

1.3 Using This Manual

Part 1 of this manual, the "User's Guide," explains how to set up QuickBASIC, how to compile and run QuickBASIC programs on your system, and discusses how to use some of QuickBASIC's new programming and language features. It also describes what you need to do to compile programs originally written for the BASIC interpreter. Refer to the "User's Guide" when you have questions about starting QuickBASIC, using the editor, or creating programs in general.

Part 2 of this manual, the "Reference," defines the BASIC language and provides a complete alphabetical reference of the statements and functions supported by QuickBASIC. Use the "Reference" when you have questions about the syntax and use of the BASIC language.

Part 3 of this manual, the "Appendixes," contains additional information you may find helpful, such as a table of ASCII character codes, a list of QuickBASIC reserved words, a description of the separate compilation method, and explanations of the QuickBASIC error messages.

This manual is intended for programmers who are familiar with the Microsoft BASIC language and with the MS-DOS or PC-DOS (hereafter referred to as DOS) operating system. It does not teach you how to program in BASIC or assembly language. For a list of suggested texts, see Section 1.6, "Resources for Learning."

The following list shows where to look for information on specific topics:

Information	Location
Installing the QuickBASIC software	Chapter 2, "Getting Started"
Setting environment variables for use with QuickBASIC	Chapter 2, "Getting Started," and Appendix D, "Using the Separate Compilation Method"
Compiling a QuickBASIC program	Chapter 4, "Compiling and Running a Program"
Executing a QuickBASIC program	Chapter 4, "Compiling and Running a Program"
Using the QuickBASIC editor	Chapter 5, "Editing and Debugging"
Using debugging mode	Chapter 5, "Editing and Debugging"
Subprograms and how to use them	Chapter 6, "Using Subprograms"
Using the user libraries and creating them	Chapter 7, "Creating User Libraries"
Incorporating assembly-language subroutines into your BASIC programs	Chapter 8, "Interfacing with Assembly Language"
Using alphanumeric labels	Chapter 9, "Language Elements"
Declaring constants and variables	Chapter 9, "Language Elements"
Data types in QuickBASIC	Chapter 9, "Language Elements"
Using arithmetic, relational, logical, and functional operators in BASIC expressions	Chapter 9, "Language Elements"

Compiling BASIC interpreter programs	Chapter 10, "Compiler-Interpreter Language Differences"
Using the new QuickBASIC language features	Chapter 10, "Compiler-Interpreter Language Differences"
Using dynamic and static arrays	Chapter 10, "Compiler-Interpreter Language Differences"
Using large numeric arrays	Chapter 10, "Compiler-Interpreter Language Differences"
Metacommands and how to use them	Chapter 10, "Compiler-Interpreter Language Differences"
Syntax for BASIC statements and functions	Chapter 11, "Statement and Function References"
Using the **BCOM20.LIB** alternate run-time library	Appendix D, "Using the Separate Compilation Method"
Linking with **SMALLERR.OBJ** and **GWCOM.OBJ**	Appendix D, "Using the Separate Compilation Method"
Invocation, compile-time, and run-time error messages	Appendix E, "Error Messages"
Selected terms used in this documentation	"Glossary"

1.4 Notational Conventions

The following notational conventions are used in this manual:

Notation	Description
`Examples`	The typeface shown in the left column is used to simulate the appearance of information that would be printed on your screen. For example, the following program statement is printed in this special typeface: `INPUT "What is your name";A$` When discussing this statement in text, words appearing in the statement, such as `INPUT` and `A$`, also appear in the special typeface.
KEYWORDS	Bold capital letters indicate BASIC keywords. These keywords are a required part of the statement syntax, unless they are enclosed in double brackets as explained below. In programs you write, you must enter keywords exactly as shown. However, you can use uppercase (capital) letters and/or lowercase letters. In the following statement, **IF** and **THEN** are BASIC keywords: **IF** *expression* **THEN** *statement* Bold capital letters are also used to indicate register names, **BP**, for example; file names, such as **DEMO.BAS**; and environment variables, **PATH**, for example.
command names	Bold type within the text indicates command names, as in the following sentence: It is not necessary to specify the library name on the **buildlib** command line.
Apostrophes ' '	An apostrophe is entered as a single right quotation mark ('), not a single left quotation mark (‘). Note that in the typeface used in examples, such as `'index variable`, apostrophes look like this: `'`.

placeholders

Items in italics are placeholders for types of information you must supply, such as a *filename*. In the following statement, *linelabel* is italicized to show that this is a general form for the **GOTO** statement:

GOTO *linelabel*

In an actual program statement, the placeholder *label* must be replaced by a specific line label, as in the following example:

```
GOTO fixup
```

[[optional items]]

Items inside double square brackets are optional. The following statement, for example, shows that entering an *argumentlist* is optional in the **CALL** statement.

CALL *name* [[(*argumentlist*)]]

Either of the following **CALL** statements is correct:

```
CALL PROG2
CALL NEXTPROG(VALUES,INDEX)
```

{*choice1*|*choice2*}

This syntax indicates that you have a choice between two or more items. Braces enclose the choices, and vertical bars separate the choices. You must choose one of the items, unless all of the items are also enclosed in double square brackets. For example, the following is the syntax of the **RETURN** statement:

RETURN [[{*linenumber*|*linelabel*}]]

This indicates the arguments are optional, but if you use an argument it must be either a line number or a line label.

Repeating elements...

Three dots following an item indicate that more items having the same form may appear. For example, this is the syntax of the **DATA** statement:

DATA *constant1*[[,*constant2*]]...

The dots following [[*constant2*]] indicate you can enter more arguments, provided the arguments are separated by commas.

Program
.
.
.
Fragment

A column of dots is used in program examples to show a portion of the program that has been omitted. For example, in the following program fragment, only two lines are shown:

```
SUB PROG2 STATIC
.
.
.
END SUB
```

KEY NAMES

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

Input text

This type is used to indicate input you enter in response to a prompt. In the following example, "John" is entered in response to the "Enter your name" prompt:

```
Enter your name: John
```



This illustration is used to distinguish mouse-related information from the rest of the text.

Note

Bold type and italics are also used occasionally in the text for emphasis.

1.5 Key Names

The names of the keys referred to in this manual correspond to the names printed on the IBM Personal Computer key tops. If you are using a different machine, these keys may have slightly different names.

The group of cursor movement (sometimes called "arrow") keys located on the numeric keypad to the right of the main keypad are called the DIRECTION keys. Each individual DIRECTION key is referred to either by the direction of the arrow on the key top (LEFT, RIGHT, UP, DOWN), or by the name on the key top (PGUP, PGDN).

The carriage-return key is referred to as ENTER.

1.6 Resources for Learning

This manual provides complete instructions for using the Microsoft QuickBASIC Compiler. However, no instruction material for BASIC or assembly-language programming is included.

Books about BASIC

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

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

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

Books about Assembly Language

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

Abel, Peter. *Assembler for the IBM PC and PC-XT.* Reston, VA: Reston Publishing Company, Inc., 1984.

Lafore, Robert. *ASSEMBLY LANGUAGE PRIMER for the IBM PC & XT.* New York, NY: New American Library, 1984.

Books about MS-DOS

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

Norton, Peter. *The Peter Norton Programmer's Guide to the IBM PC.* Bellevue, WA: Microsoft Press, 1985.

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

These books are listed for your convenience only. With the exception of its own publications, Microsoft Corporation does not endorse these books or recommend them over others on the same subject.

Chapter 2
Getting Started

2.1 Backing Up Your Disks 17
2.2 Disk Contents 17
2.3 Setting Up the Compiler 19
2.3.1 Example 1: Two Floppy-Disk Drives 19
2.3.2 Example 2: Hard-Disk Setup 20
2.3.3 Example 3: One Floppy-Disk Drive 20
2.4 Starting QuickBASIC 21
2.5 Viewing the Help Window 21

This chapter tells you how to set up your program development environment and how to start QuickBASIC. Before you begin using the Microsoft QuickBASIC Compiler, follow the preliminary procedures described in this chapter. The preliminary procedures include the following:

- Copying (or "backing up") your product disks
- Organizing your QuickBASIC software

If you are unfamiliar with any of the DOS procedures mentioned, please consult your DOS manual for instructions.

2.1 Backing Up Your Disks

The first thing you should do when you have unwrapped your package is make working copies of the QuickBASIC disks using the DOS **DISKCOPY** command. Put the original disks in a safe place and use the copies from now on. If the copies are ever damaged or destroyed, you can make new copies from the original disks.

2.2 Disk Contents

Your Microsoft QuickBASIC Compiler disks contain all the files listed in Table 2.1.

Table 2.1
Disk Contents

Disk 1 File Names	Description
BUG.BAS	Example program for use with debugging mode practice session
CALLSHAP.BAS	Example program for use with user library practice session
DEMO.BAS	Example program for use with QuickBASIC practice session
EX.BAS	Example program for use with user library practice session
QB.EXE	The Microsoft QuickBASIC Compiler
README.DOC	Important information that became available after this manual was printed; not always included
SQUARE.BAS	Example program for use with user library practice session
TRIANGLE.BAS	Example program for use with user library practice session
UPDATE.DOC	List of recent software improvements; not always included
USERLIB.ASM	Assembly-language source code for **USERLIB.OBJ**

Disk 2 File Names	Description
BCOM20.LIB*	The Microsoft QuickBASIC alternate run-time library
BRUN20.EXE*	The Microsoft QuickBASIC run-time module
BRUN20.LIB*	The Microsoft QuickBASIC run-time module library
BUILDLIB.EXE	A program that builds user libraries
GWCOM.OBJ*	Object file containing communications support routines
LINK.EXE*	The Microsoft Overlay Linker
PREFIX.ASM	Assembly language source code for **PREFIX.OBJ**
PREFIX.OBJ	Determines segment ordering when user libraries containing assembly-language routines are built
SMALLERR.OBJ*	Object file that produces smaller run-time error messages
USERLIB.OBJ	Unbuilt user library containing software interrupt support

* Used only with the compiling and linking method described in Appendix D, "Using the Separate Compilation Method."

Your disks may contain other example programs from this manual and some general purpose BASIC programs. All the BASIC programs have the extension **.BAS**.

2.3 Setting Up the Compiler

Before you compile any programs, make sure that QuickBASIC can locate all the files it needs to process your program. Table 2.2 describes the required files.

Table 2.2
QuickBASIC Required Files

To:	You need:
Compile and run a BASIC program	**QB.EXE**
Build a user library	**BUILDLIB.EXE**
Build a user library containing assembly-language subroutines	**PREFIX.OBJ**

2.3.1 Example 1: Two Floppy-Disk Drives

The following disk setup is suggested for a system with two floppy-disk drives. The disk in Drive A should be a DOS system disk, formatted with the **/S** option:

Drive A	Drive B
QB.EXE	BASIC source files
BUILDLIB.EXE	
PREFIX.OBJ	

2.3.2 Example 2: Hard-Disk Setup

If you have a hard disk, you may want to place the executable and source files in separate directories, and set your DOS **PATH** environment variable accordingly. (See your DOS manual for a description of the **PATH** environment variable.) The following is a sample hard-disk setup:

C:\BIN\QB.EXE

C:\BIN\BUILDLIB.EXE

C:\SRC\PREFIX.OBJ

Using the above setup, your BASIC source files would go in the directory **SRC**. The **PATH** environment variable would have the following value:

```
PATH=C:\BIN
```

If you are compiling with batch files created for QuickBASIC Version 1.0, or if you want to create programs that can be executed outside the QuickBASIC environment, see Appendix D, "Using the Separate Compilation Method," for instructions on how to set up your disk.

2.3.3 Example 3: One Floppy-Disk Drive

If you have a computer with one floppy-disk drive, there are two ways to set up your system:

1. Put QuickBASIC, your BASIC source files, and any other files you need (such as **BUILDLIB.EXE** and the user library) on a single disk. Space is limited on a disk, so this may not be convenient if you have very large source files or user libraries.

2. Put QuickBASIC on one disk and your other files on another disk. Using this method, you always start QuickBASIC without a filename argument, then swap disks and use the Load command from QuickBASIC's File menu to load the program.

2.4 Starting QuickBASIC

The best way to learn to use QuickBASIC is to work through the practice session in Section 3.8 before you start using QuickBASIC on your own. However, if you are in a hurry to begin, this section is a brief explanation of how to start and quit QuickBASIC.

To start QuickBASIC, type

```
qb
```

and press ENTER.

Information about choosing commands appears when QuickBASIC starts. Use the commands described in Chapter 4, "Compiling and Running a Program," and Chapter 5, "Editing and Debugging," to load, run, and edit program files.

To leave QuickBASIC and return to the operating system

1. Press ALT-F to display the File menu.
2. Press ALT-Q to execute the Quit command.

Other ways to leave QuickBASIC are described in Chapter 4, "Compiling and Running a Program."

If you leave QuickBASIC without writing your program to a disk, you are asked if you want to save it. Type the name you want to give the file, then press ENTER to save the program.

2.5 Viewing the Help Window

QuickBASIC provides a keyboard command summary called the Help window. To view the Help window at any time, press F1. Press any key to remove the window.

If you have a mouse, click on the box in the upper left corner to view the Help window. Click on the box again to remove the window.

Chapter 3
Exploring QuickBASIC

3.1 If You Have a Mouse 25
3.2 The QuickBASIC Screen 26
3.3 Choosing Commands from Menus 28
3.4 Shortcut Command Keys 30
3.5 Selecting Options from Dialog Boxes 31
3.6 Making Selections 33
3.7 Scrolling 34
3.8 Practice Session 35
3.8.1 Compiling and Running a Program 35
3.8.1.1 Starting QuickBASIC 35
3.8.1.2 Loading DEMO.BAS 36
3.8.1.3 Compiling DEMO.BAS 38
3.8.1.4 Running DEMO.BAS 41
3.8.2 Using the QuickBASIC Editor 43
3.8.2.1 Copying and Pasting 43
3.8.2.2 Replacing Text 46
3.8.2.3 Deleting and Inserting 47
3.8.2.4 Searching and Replacing 51
3.8.3 Finding and Correcting Errors 55
3.8.4 Saving the Program 59
3.8.5 Leaving QuickBASIC 60
3.8.6 Exploring Further 61

3.9 Using Debugging Mode 61
3.9.1 The Debugging Mode Display 61
3.9.2 Changing the Display 62
3.9.3 Executing the Program 63
3.9.4 Subprogram Tracing 64
3.9.5 Continuous Execution 65
3.9.6 Leaving Debugging Mode 65
3.10 Creating a User Library 65
3.10.1 Creating Object Files 65
3.10.2 Building the Library 66
3.10.3 Using the Library 66

This chapter introduces you to the QuickBASIC user interface and to the operation of the compiler and editor. The following information is included in this chapter:

- Section 3.1 explains the mouse-specific terms used in this manual.
- Section 3.2 describes the QuickBASIC screen.
- Sections 3.3–3.7 explain how to choose commands and options, make selections, and scroll text.
- Section 3.8 is a brief tutorial introduction to QuickBASIC. You can follow any or all of the tutorial sections to get started quickly with the compiler, then refer to Chapters 4 and 5 when you are ready to learn more. You can also turn directly to Chapters 4 and 5 for a complete description of the compiling and editing functions.
- Sections 3.9 and 3.10 show you how to use QuickBASIC's debugging mode and how to create a user library. Before you read these sections, try compiling and running some QuickBASIC programs.

3.1 If You Have a Mouse



You can use QuickBASIC with or without a mouse. Where appropriate, this manual describes a procedure with the keyboard and with the mouse, so you can use either or both techniques.

This section is for mouse users only. It defines the mouse-related terms used in this manual. If you do not have a mouse, skip this section and go on to Section 3.2, "The QuickBASIC Screen."

QuickBASIC can be used with either a single- or multiple-button mouse. If you have a mouse with more than one button, use the *left-most* button. QuickBASIC uses only the left-most button.

Note

Microsoft QuickBASIC is designed for use with the Microsoft Mouse. Many manufacturers advertise their pointing devices as being compatible with the Microsoft Mouse. QuickBASIC may work with some of these devices if they are fully compatible, especially if they emulate the Microsoft Mouse function calls. Check with your mouse manufacturer for details.

Moving the mouse across a flat surface moves the *pointer*, which is the square block on the screen that shows the current mouse position. The following mouse-specific terms are used in this manual:

Term	Definition
Point	Move the mouse until the the pointer rests on what you want to point to.
Press	Hold down the mouse button.
Click	Quickly press and release the mouse button.
Drag	Hold down the mouse button while moving the mouse across a flat surface, such as a desk top.
Double click	Click the mouse button twice in rapid succession.

3.2 The QuickBASIC Screen

When you start QuickBASIC without a file-name argument, a brief summary of how to choose commands is displayed. After you press ESC, the screen appears as shown in Figure 3.1:

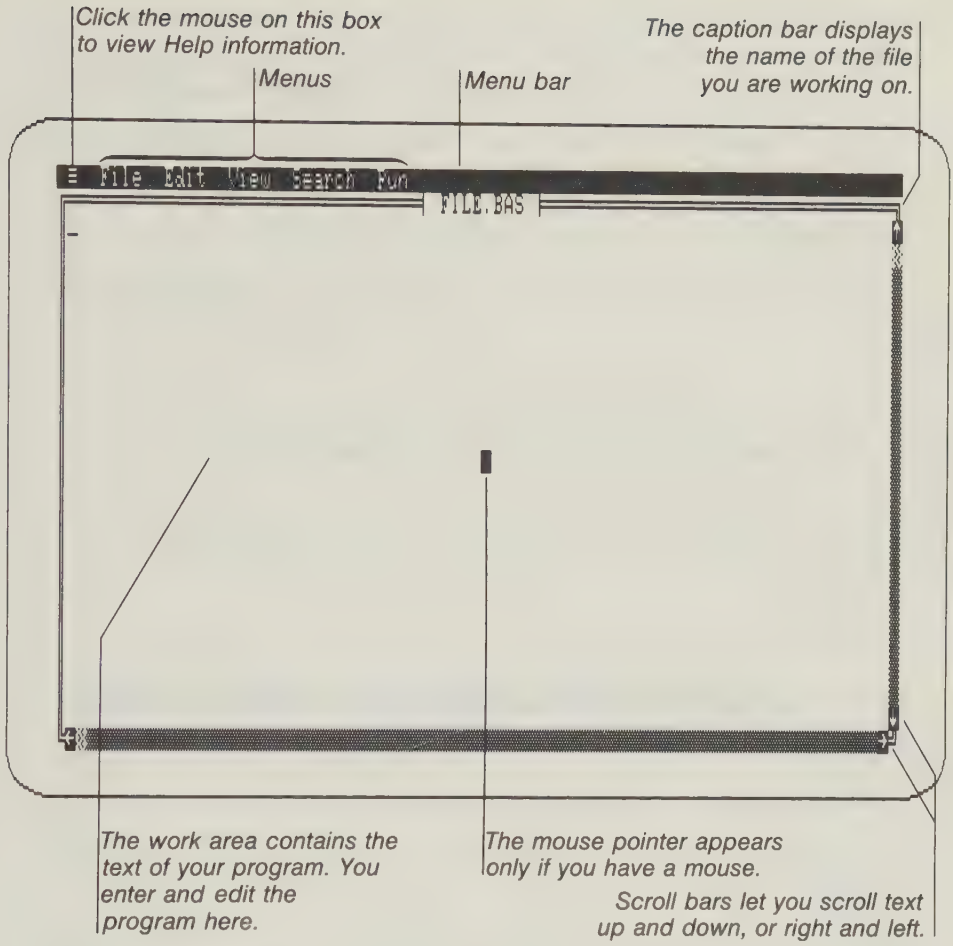


Figure 3.1 QuickBASIC Screen

The elements in Figure 3.1 are described as follows:

The *flashing underscore* (called the "cursor") shows where you may enter text on the screen.

If you have a mouse installed, you will also see a square block.



The *work area* contains the text of your program. You enter and edit your program in the work area.

The *menu bar* contains the names of the QuickBASIC command menus.

The *caption bar* displays the name of the file you are working on. If no file is loaded, or if you are working on a new, unnamed file, the two vertical lines extend all the way across the screen.

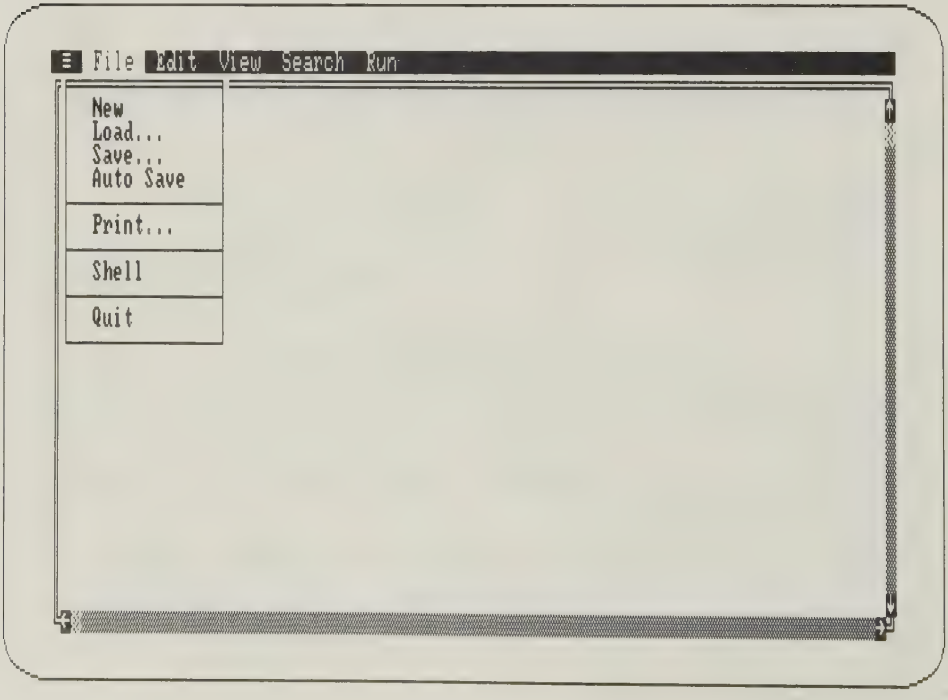
The *scroll bars* show your relative position in the file.



If you have a mouse, scroll bars let you move the text up and down or right and left.

3.3 Choosing Commands from Menus

QuickBASIC commands are organized in *menus* on the menu bar. The File menu looks like this:



Commands followed by dots (. . .) prompt you for additional information when they are selected. Commands without dots are executed as soon as they are chosen.

There are several techniques you can use to display menus and choose commands. Use the techniques you find most convenient.

To choose a command with the keyboard, perform the following steps:

1. Hold down the ALT key and press the initial letter of the menu name. This opens the menu.

2. Use the DIRECTION keys to highlight the command.

   You can also highlight a command by pressing the command's initial letter. If the menu has more than one command with the same initial, you will need to press the letter twice to highlight the second command. For example, on the File menu there are two commands that begin with S: Save... and Shell. You will need to press S twice to get to the Shell command.

3. Press ENTER to carry out the command.

Here is a quicker way to choose a command with the keyboard:

1. Press and hold down the ALT key.

2. Press a letter to display a menu, but still hold down the ALT key.

3. Press a letter to choose a command, then release both keys to carry out the command.

You can press the ESC key to make a menu go away. To move from menu to menu

1. Display any menu.

2. Press the LEFT or RIGHT DIRECTION key to open the next menu.



To choose commands with the mouse

1. Point to the menu name, then press and hold down the mouse button.
2. Drag the highlight down to the command you want.
3. Release the mouse button.

To cancel a command with the mouse, simply move the pointer off the menu and release the button.

3.4 Shortcut Command Keys

To the right of some menu commands you will see a key name, as shown in Figure 3.2:

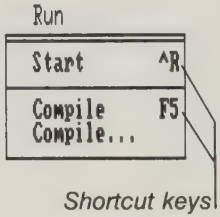


Figure 3.2 Shortcut Keys

These are *shortcut keys*. You can use the shortcut keys to carry out a command without displaying the menu first. For example, instead of pressing ALT-R to display the Run menu, then choosing Start to execute a program, you could just press CTRL-R. Table 3.1 lists the QuickBASIC shortcut keys.

Table 3.1

QuickBASIC Shortcut Keys

Command	Key(s)
Undo	SHIFT-ESC
Cut	DEL
Copy	F2
Paste	INS
Selected Text	CTRL-F
Repeat Last Find	F3
Next Error	F6
Start	CTRL-R
Compile	F5

3.5 Selecting Options from Dialog Boxes

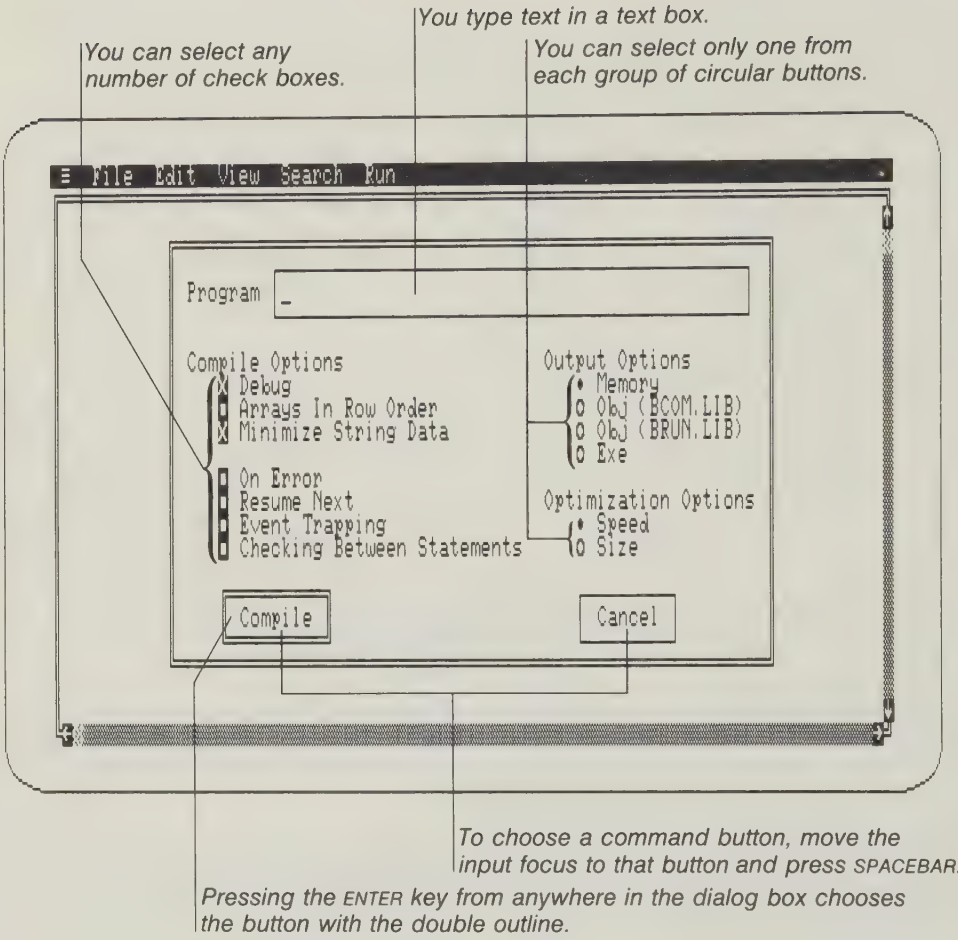
QuickBASIC displays a dialog box when it needs additional information or confirmation from the user to carry out an action. The dialog box contains areas where you enter information, select options to commands, or activate controls to execute or cancel a command.

While a dialog box is displayed, input from the keyboard affects the item in the dialog box that has the *input focus.* When a text box has the input focus, the cursor appears in the box. Other items that have the input focus are highlighted and the text appears in capital letters.

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



In this section, the Compile... dialog box is used to explain the parts of a dialog box. The Compile... dialog box looks like this:



Text boxes are used to enter information for commands, such as a file name, or text to be searched for. When the text box has the input focus, QuickBASIC is ready for you to enter or edit the text in the box. The flashing underscore indicates when the text box has the input focus. What you type appears to the left of the cursor.

You can press BACKSPACE to correct typing errors in a text box, or use the QuickBASIC editing keys described in Section 5.1. The TAB key moves the input focus from the text box to the other items in the dialog box.

Sometimes a text box appears with information already in it. For example, when the Compile... dialog box opens, the name in the text box is either the last file loaded or the last file compiled.

Square "check" boxes let you select several options for a command. Use the TAB and SHIFT-TAB keys to move forward and backward between check boxes. When the option has the input focus, press the SPACEBAR to select it. When a check box is selected, an "X" appears in the box.

Circular buttons indicate that only one option from the group can be selected. The button is filled when an option is selected. Use the DIRECTION keys to move among circular buttons.

"Command" buttons are labeled to indicate what choosing the button will do. Command buttons with double outlines are defaults—commands you will want to use most frequently. Use the TAB key to move between command buttons. Pressing SPACEBAR while the command button has the input focus executes the command with the options you chose. When the command executes, the dialog box disappears.

Often, a dialog box opens with information in it reflecting options chosen earlier. For example, the Compile... dialog box opens with the options set from the last time you compiled.

To select with a mouse



1. Point to the item in the text box.
2. Click the mouse button.

3.6 Making Selections

When you work with QuickBASIC, you need to *select* the object that the next command or action will affect. The selected object is usually highlighted, either in reverse video or high intensity.

In dialog boxes, you select items one at a time by pressing the SPACEBAR while the item has the input focus. To select text, either in the work area or in a text box, hold down the SHIFT key while you move the cursor over the text you want to select. You can use different key combinations to select more than one character of text at a time. These key combinations are listed in Table 5.1.

To select items in dialog boxes with the mouse, point at the item and click the button. To select text, point at the character to the left of the character you want to select and drag the mouse over it. The techniques for selecting text with the mouse are described in Section 5.1, "The QuickBASIC Editor."

3.7 Scrolling

Often, a program is too big to fit in the work area. To view information that won't fit in the work area, you can move the text up and down or right and left. This is called *scrolling.*

To scroll with the keyboard, once you have reached the last character on the screen, press the DIRECTION key for the direction you want to scroll. For example, to scroll right, go to the right-most character on the screen and keep pressing the RIGHT key. To scroll more than one character at a time, refer to Table 3.2.

Table 3.2
Scrolling

To scroll:	Press:
To the end of a line	END
To the beginning of a line	HOME
Down one screen	PGDN
Up one screen	PGUP
To the top of the file	CTRL-PGUP
To the end of the file	CTRL-PGDN



To scroll with the mouse, point at the "scroll box" (the dark box in the scroll bar) and drag it to a position in the scroll bar that corresponds to the general location you want (beginning, middle, or end of the file). To scroll one line or one character at a time, click on the arrows at either end of the scroll bar.

3.8 Practice Session

This practice session shows you how to load, compile, and execute a QuickBASIC program, and how to do simple editing. The practice session takes approximately 30 minutes to complete. To complete the practice session, you will need the sample program **DEMO.BAS** from QuickBASIC product Disk 1, and the file **QB.EXE** must be in the current directory or in your search path.

3.8.1 Compiling and Running a Program

This section shows you how to load and compile a program using the Microsoft QuickBASIC Compiler.

3.8.1.1 Starting QuickBASIC

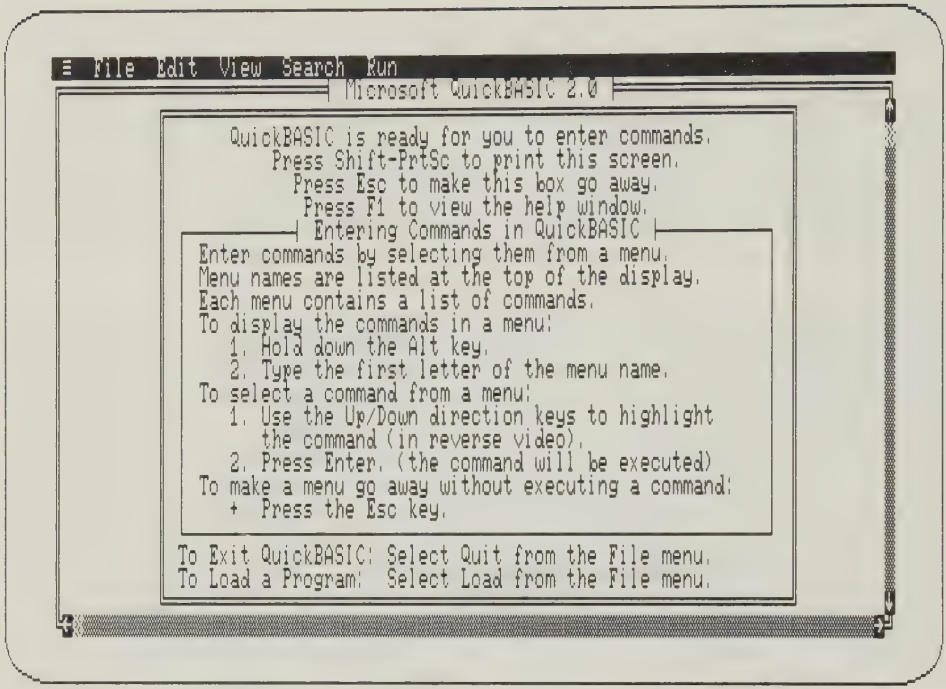
Before you begin, complete the following steps:

- Make sure **DEMO.BAS** is in the current directory.
- Make sure that QuickBASIC is installed as described in Section 2.3.

Start QuickBASIC by typing

```
qb
```

Press ENTER to carry out the command. Your screen should look like this:



This start-up screen reminds you how to choose commands, load a file, and exit QuickBASIC.

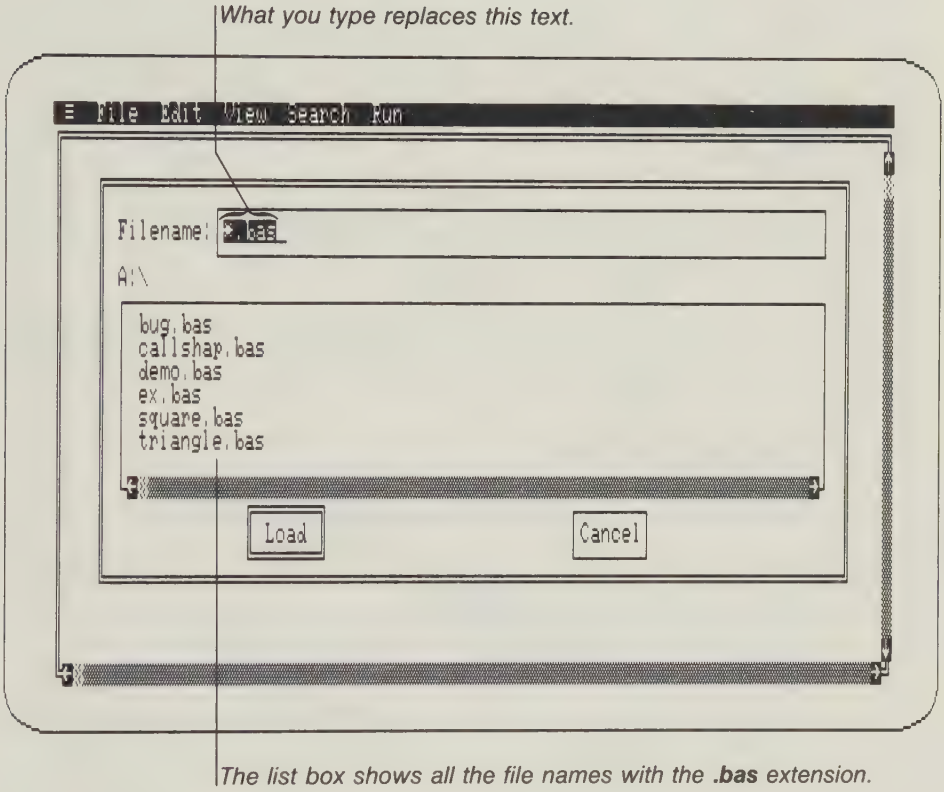
3.8.1.2 Loading DEMO.BAS

Press ESC to clear the start-up information from the screen. The QuickBASIC compiler is ready for you to start typing or to load a program from the disk. For this practice session, load the program **DEMO.BAS**. Follow these steps:

1. You execute commands in QuickBASIC by displaying a menu, then indicating which command you want to execute. Right now you want to execute the Load... command, which copies a file from a disk into memory. The Load... command is in the File menu, so press ALT-F to display the File menu. If you make a mistake and display a different menu, press ESC to close the menu and try again.

2. Press the DOWN key until the Load... command is highlighted. If you go past, keep pressing the key until Load... is highlighted again.

3. Press ENTER. The Load... dialog box opens:



When the Load... dialog box first opens, the specification `*.bas` is selected in the text box, and all the files in the current directory with the extension **.BAS** appear in the list box. Your screen may look different from the illustration. (You won't do anything with the list box in this practice session, but if you want to know more about it see Section 4.2.2, "Load...".)

4. You use the Load... dialog box to tell QuickBASIC which file you want to place in memory. To do this, type **DEMO.BAS**, the name of the file you want to load. What you type replaces the text in the text box. If you make a mistake, use the BACKSPACE key to reposition the cursor, then retype the necessary information to correct the error.

5. When you are finished typing the file name, press the TAB key twice to move the input focus to the Load command button. When Load has the input focus, the command button and the text appear highlighted in intense video. If you move the input focus past the Load command button, use the SHIFT-TAB keys to move the input focus back.

6. Press SPACEBAR to choose the command button. When a command button is chosen, it immediately carries out an action. In this case, the Load... command is executed.

   If the name in the caption bar changes to `demo.bas`, but no text appears in the work area, the **DEMO.BAS** program is not in the current directory. Repeat steps 1–6, using the correct drive and directory names.



To load a file with the mouse

1. Point to the word "File" on the menu bar and press the left button.

2. Holding down the left-most button, drag the mouse until the Load... command is highlighted.

3. Release the mouse button.

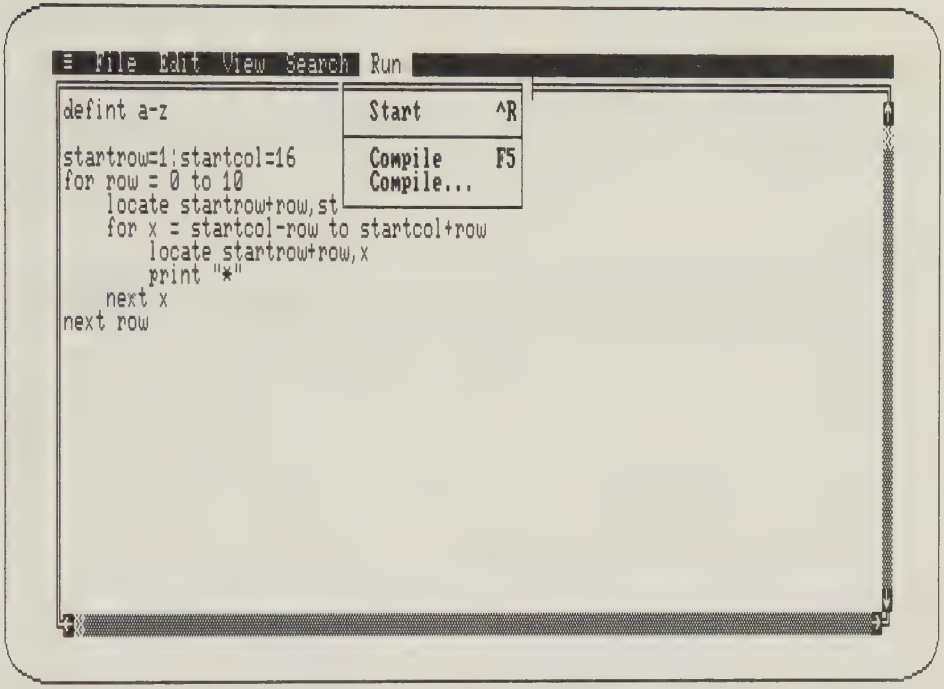
4. Point at the file name `demo.bas` in the list box and double click.

   The Load... command executes, and the text of **DEMO.BAS** appears in the work area.

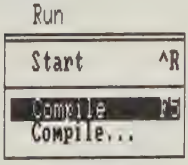
3.8.1.3 Compiling DEMO.BAS

When the file is loaded, the program text appears in the work area and you are ready to create an executable program. Follow these steps:

1. Press ALT-R to display the Run menu:



2. Use the DOWN key to choose Compile, which is highlighted as shown below:



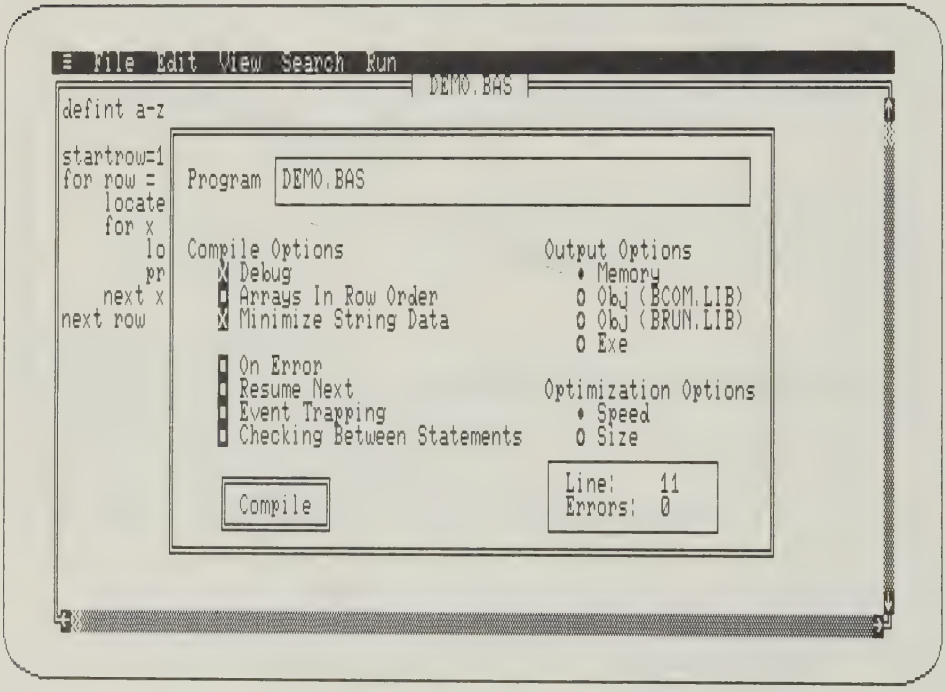
3. Press ENTER.

   The **DEMO.BAS** program will begin compiling.

To compile a program with the mouse

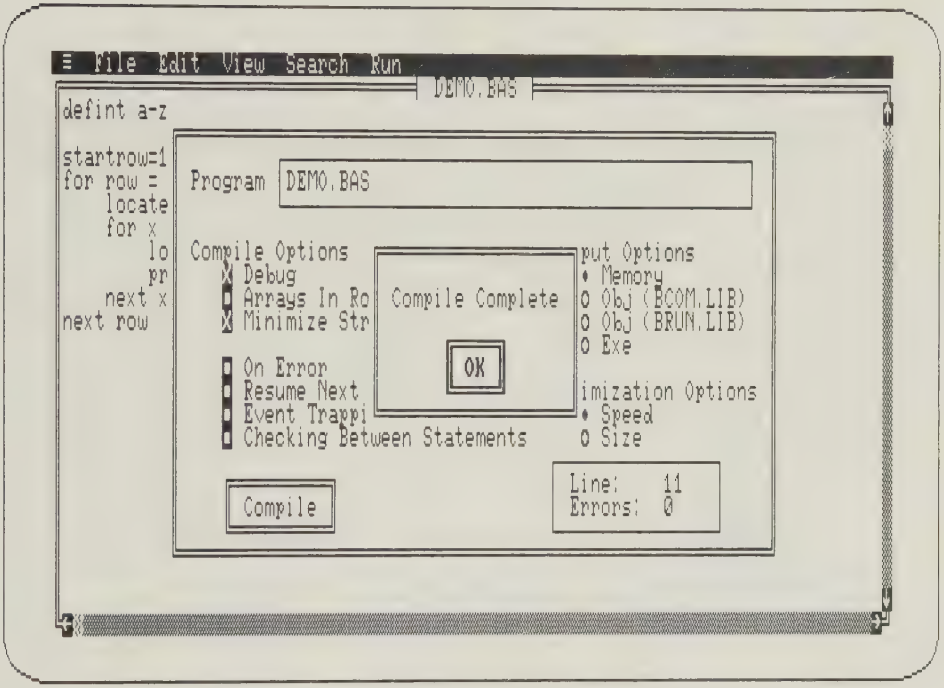
1. Point to the word "Run" in the menu bar and press the left button.
2. Holding down the button, drag the mouse until the Compile command is highlighted.
3. Release the mouse button.

While the program is compiling, your screen looks like this:



The display shows the option settings QuickBASIC is using to compile **DEMO.BAS**. Right now, QuickBASIC is using the default options: Debug, Minimize String Data, Memory, and Speed. You can tell the options are selected because the check boxes are marked with an "X" and the circular buttons are filled. (Using and selecting options is explained in Chapter 4, "Compiling and Running a Program.")

While the program is compiling, the line numbers in the lower-right corner increment rapidly. The number in the Errors line should remain at 0. When compilation is finished, the screen looks like this:



Press ENTER to return to the editor.

If you have a mouse, click on the OK command button to return to the editor.



3.8.1.4 Running DEMO.BAS

When the **DEMO.BAS** program is finished compiling, you are ready to execute it. Follow these steps:

1. Press ALT-R to display the Run menu.

2. Press the DOWN key to choose the Start command.

3. Press ENTER.

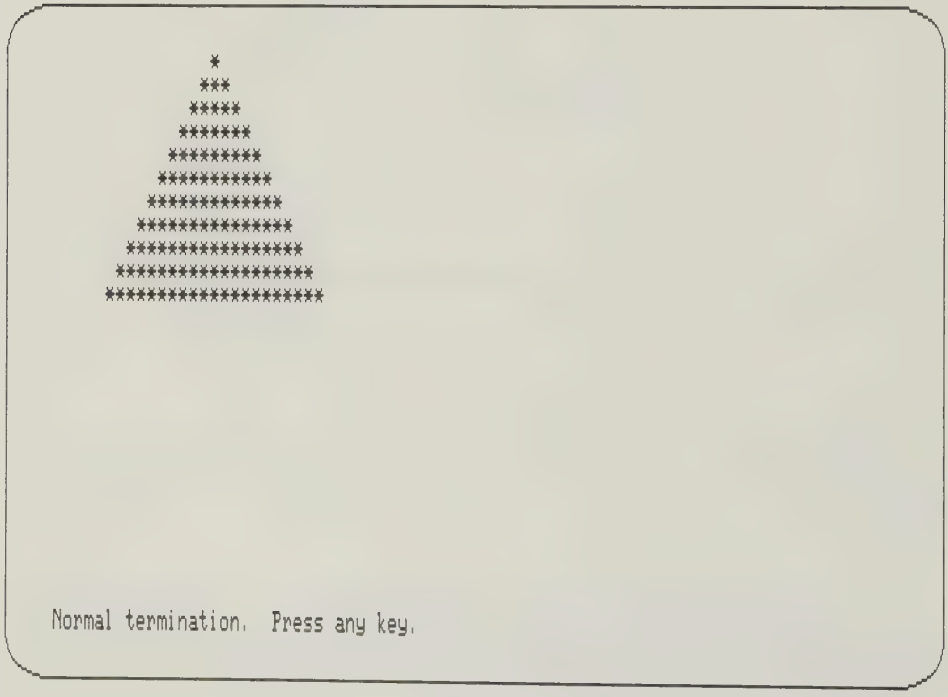
   The **DEMO.BAS** program will begin executing.



To run the program with the mouse

1. Point to "Run" in the menu bar and press the left button.
2. Holding down the button, drag the mouse until the Start command is highlighted.
3. Release the mouse button.

The **DEMO.BAS** program executes, and your screen should look like this:



When the program is finished executing, you'll see the message

```
Normal termination. Press any key.
```

Press ENTER to return to the QuickBASIC editor.

3.8.2 Using the QuickBASIC Editor

Although you can create programs using any text editor, QuickBASIC has a built-in editor that allows you to create, compile, run, and edit a program without leaving QuickBASIC. This section shows you how to use some of the functions of the built-in editor. A complete description of the editor appears in Chapter 5, "Editing and Debugging."

3.8.2.1 Copying and Pasting

In this section you'll edit **DEMO.BAS** so that it places a second triangle on the screen. The first step is to copy some lines of the program and place them at the end of the file.

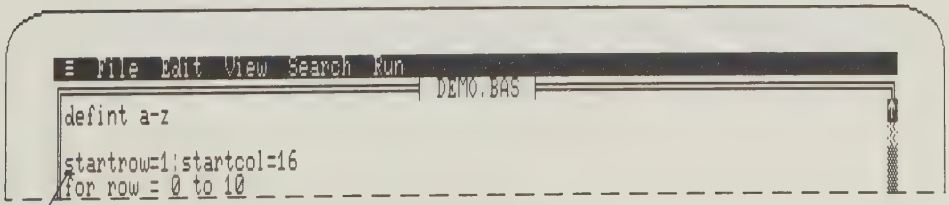
Step 1: Select the lines

Before you can copy the lines, you must select them. Selecting tells the Copy command which lines to copy. Follow these steps:

1. Use the DIRECTION keys to move the cursor to the beginning of the second line.

   If you are using a mouse, point to the beginning of the second line and click the left button.

   When the cursor is in the correct position, your screen looks like this:

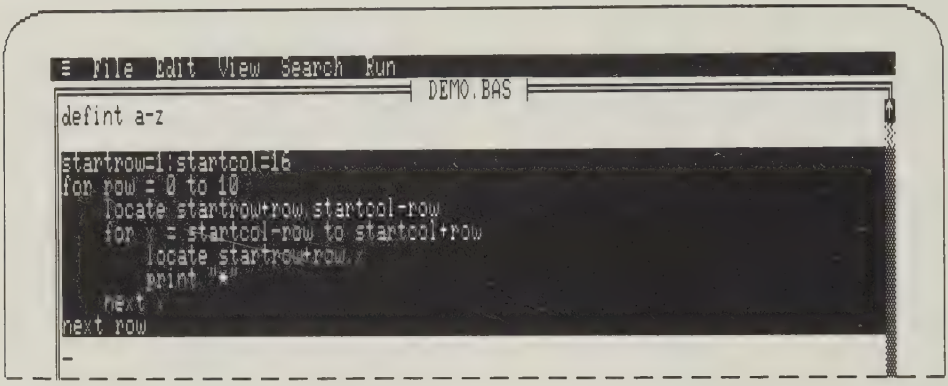


Position the cursor here before you start selecting text.

2. If the cursor is in the correct position, you're ready to select the text. Holding down the SHIFT key, press the DOWN key and select all the lines from here to the end of the program.

   If you are using a mouse, hold down the left button and drag the mouse down through the last line in the program.

Your screen should look like this:



Step 2: Copy the lines into the Clipboard

When the lines are selected, follow these steps to copy them into the Clipboard, which is a temporary holding place:

1. Press ALT-E to display the Edit menu.

2. Press the DOWN DIRECTION key until the Copy command is highlighted.

3. Press ENTER.

   The screen doesn't change, and although it looks like nothing is happening, the selected text is copied into the Clipboard.

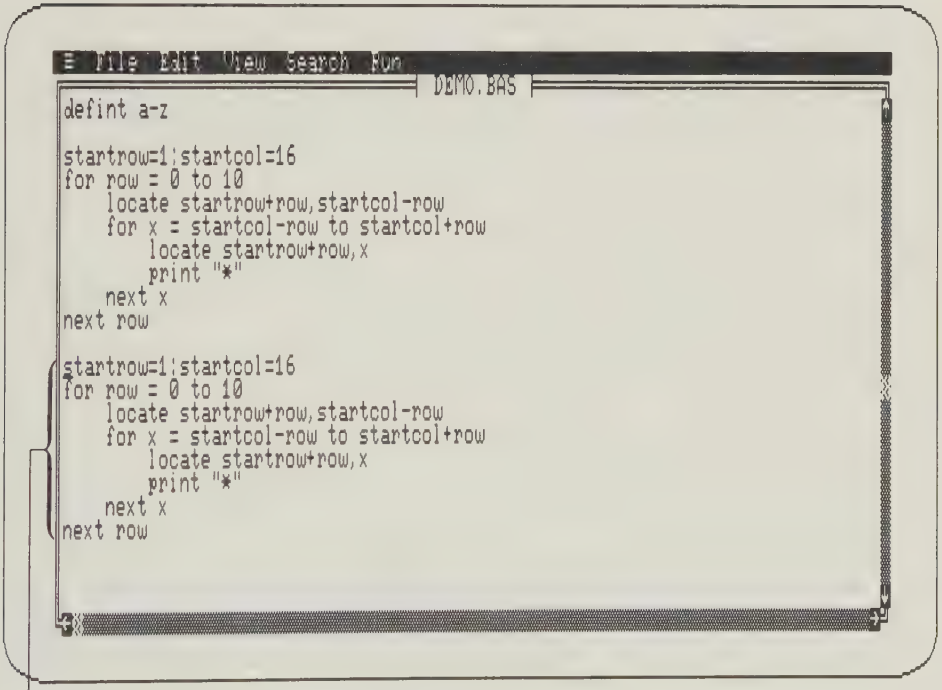
Step 3: Paste at the new location

The Paste command inserts the contents of Clipboard at the location of the cursor. Follow these steps to insert the text you just copied:

1. Press END to move the cursor to the end of the current line, then press ENTER twice. This creates an empty line between the original text and the new text, and will make the final program more readable.

   If you are using a mouse, point at the end of the current line and click. Press ENTER twice to create the empty line.

2. Choose Paste from the Edit menu to insert the text from the Clipboard. Your screen should look like this:



Paste inserts the text from the Clipboard.

3.8.2.2 Replacing Text

Now you need to set the location of the second triangle. You'll do this by changing the value of `startcol` in the copied text. This variable holds the number of the column where the triangle is printed. Follow these steps:

1. Move the cursor to the space after the `16` at the end of the first line of the new text.

```
    next x
next row

startrow=1:startcol=16_
for row = 0 to 10
    locate startrow+row,startcol-row
    for x = startcol-row to startcol+row
```

2. Press SHIFT-LEFT twice to select `16`.



   If you have a mouse, point to the space after the `6` and drag the mouse over the two characters to select them.

3. Type

   `64`

   What you type replaces the selected text. The second triangle will be printed on the screen at column 64.

When you are finished editing, follow these steps to recompile the program:

1. Press ALT-R to open the Run menu.
2. Press the DOWN key until Compile is highlighted.
3. Press ENTER.

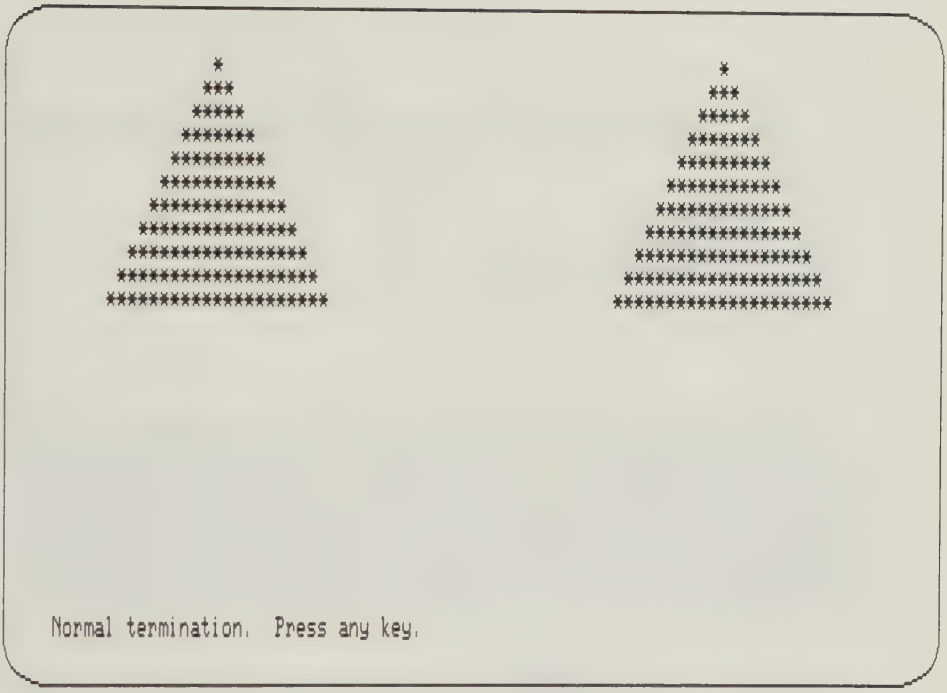
   The edited **DEMO.BAS** program begins compiling.



If you are using a mouse, point at the Run menu, drag the mouse until Compile is highlighted, then release the button.

If QuickBASIC encounters an error during compilation, see Section 3.8.3, "Finding and Correcting Errors," to find out how to correct it.

If the program compiles without an error, choose Start from the Run menu to execute it. When the program executes, there will be two triangles on the screen:



3.8.2.3 Deleting and Inserting

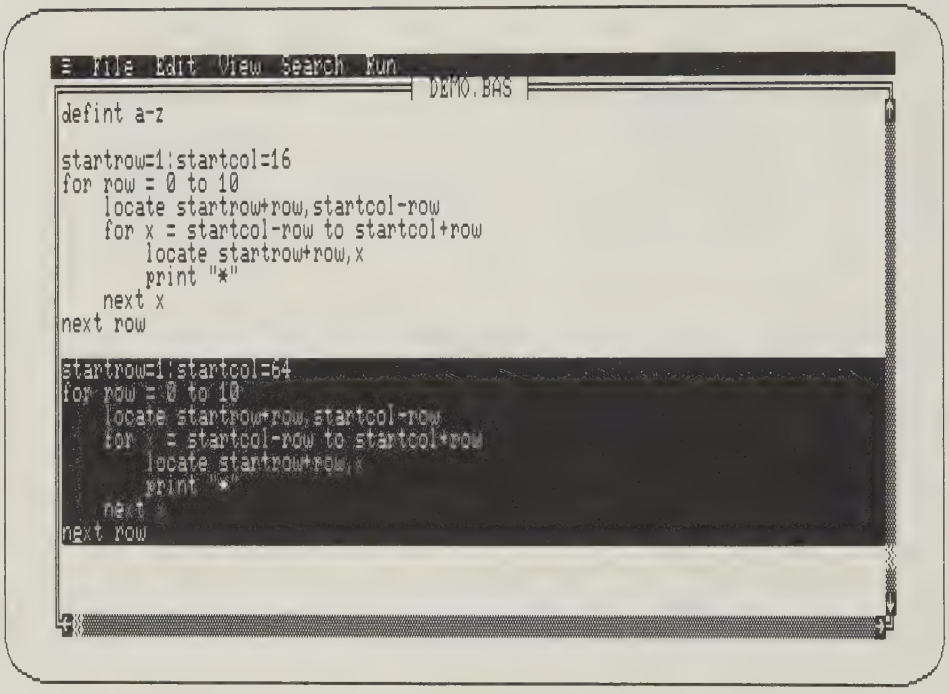
In this section, you'll delete a block of text and insert some new text in the middle of the program. As part of the example, you'll break the program into two parts, a "subprogram" that draws a triangle, and a "main" program that calls the subprogram, and gives the screen locations where the triangles will be drawn. Subprograms are described in Chapter 6, "Using Subprograms."

The first step is to delete the text we created in the last section. Follow these steps:

1. Use the SHIFT-DIRECTION keys to select the last eight lines of the program.

   If you have a mouse, select the lines with the mouse.

   When the lines are selected, the screen looks like this:



2. Press ALT-E to open the Edit menu.

   If you have a mouse, point at the Edit menu.

3. Use the DOWN key to highlight the Cut command.

   With the mouse, hold down the left button, and drag the highlight down to the Cut command.

4. Press ENTER to delete the selected lines.

   If you are using a mouse, release the button to delete the selected lines.

The next step is to insert the new "main" program text. First you'll need to open a new line, as follows:

1. Use the UP key to move to the second line of the program.

   If you have a mouse, move the pointer to the second line and click.



2. Press CTRL-N to insert a line above the second line.

   Notice that the first line of the program moves, or "scrolls," up and off the screen.

Now you need to enter the new, "main" part of the program. The variables `startrow` and `startcol` hold the row and column locations where the triangles are drawn. The subprogram `triangle` draws a triangle.

1. Type

```
for startrow = 1 to 12 step 12
    for startcol = 16 to 64 step 24
      call triangle (startrow,startcol)
    next startcol
next startrow
```

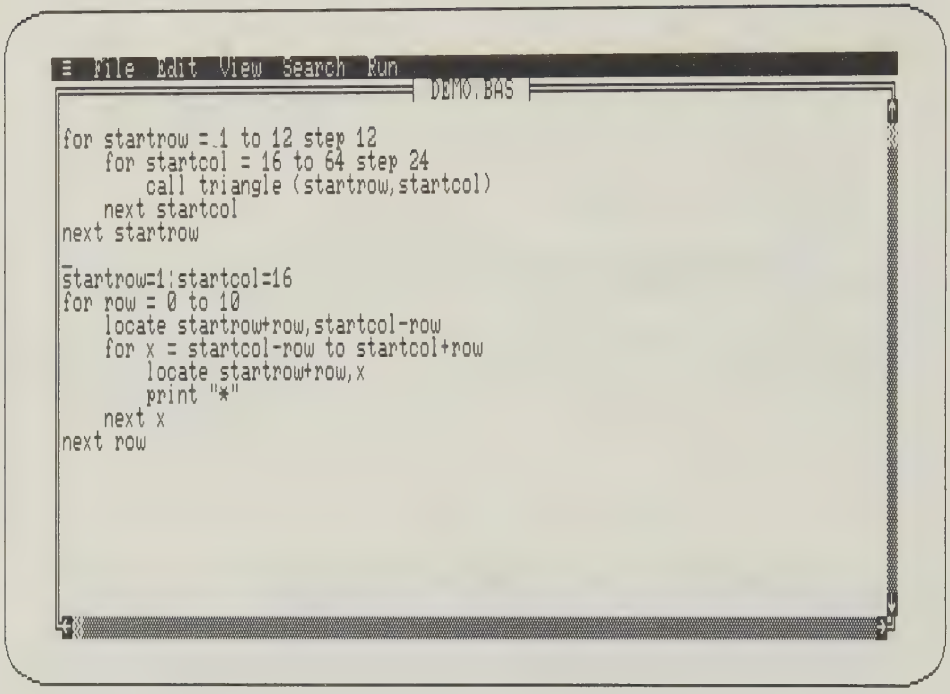
   When you start a new line after indenting with either the TAB key or the SPACEBAR, the cursor is positioned in the same column as the beginning of the previous line. Press END to move back to the left margin, or BACKSPACE to move back one space at a time.

   With the mouse, move the pointer to the location you want and click.



2. Press ENTER after you type the last line.

   At this point, your screen should look like this:



3. Now that you have created the main program, you are ready to create the subprogram. Move the cursor down to the line `startrow=1:startcol=16`. When the cursor is positioned correctly, the text looks like this:

```
next startrow

startrow=1:startcol=16
for row = 0 to 10
```

4. Press SHIFT-END to select the entire line.

   With a mouse, hold down the button and drag the mouse to the end of the line to select it.

5. Type

   ```
   sub triangle(vertical,horizontal) static
   ```

What you type replaces the selected text.

The variables `vertical` and `horizontal` hold the values from `startrow` and `startcol`, which are passed to the subprogram from the main program. In the next section you'll learn how to change the names `startrow` and `startcol` to `vertical` and `horizontal` where necessary.

3.8.2.4 Searching and Replacing

In this section you'll learn how to use the Change... command in the Search menu to find and replace parts of your program. You need to change the variable names `startrow` and `startcol` to `vertical` and `horizontal` in the last seven lines of the program. Follow these steps to use the Change... command in the Search menu to change the variable names:

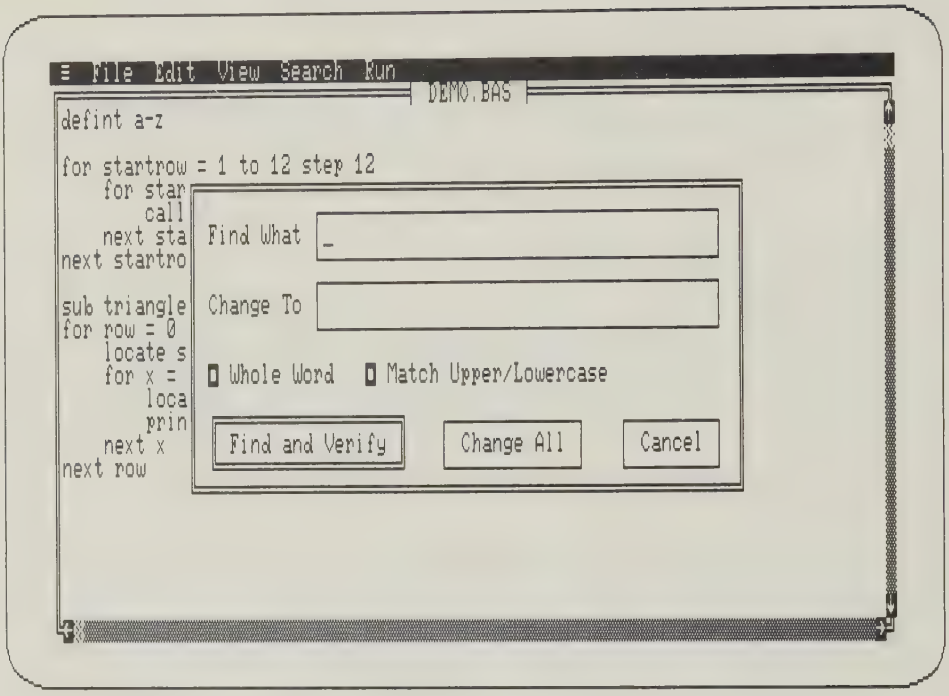
1. Press ALT-S to display the Search menu.

2. Use the DIRECTION keys to highlight the Change... command, then press ENTER.

   If you are using a mouse, point to the Search menu, drag the mouse down until the Change... command is highlighted, then release the button.



3. The Change... dialog box appears:



You will use the Change... dialog box to specify the text you want to find, and the replacement text.

4. The cursor is positioned correctly when the dialog box opens. Type

   ```
   startrow
   ```

   What you type appears in the Find What text box. When you are finished typing, press the TAB key to move the input focus (and the cursor) to the Change To text box.

   If you have a mouse, position the mouse pointer in the Change To text box and click.

5. Type

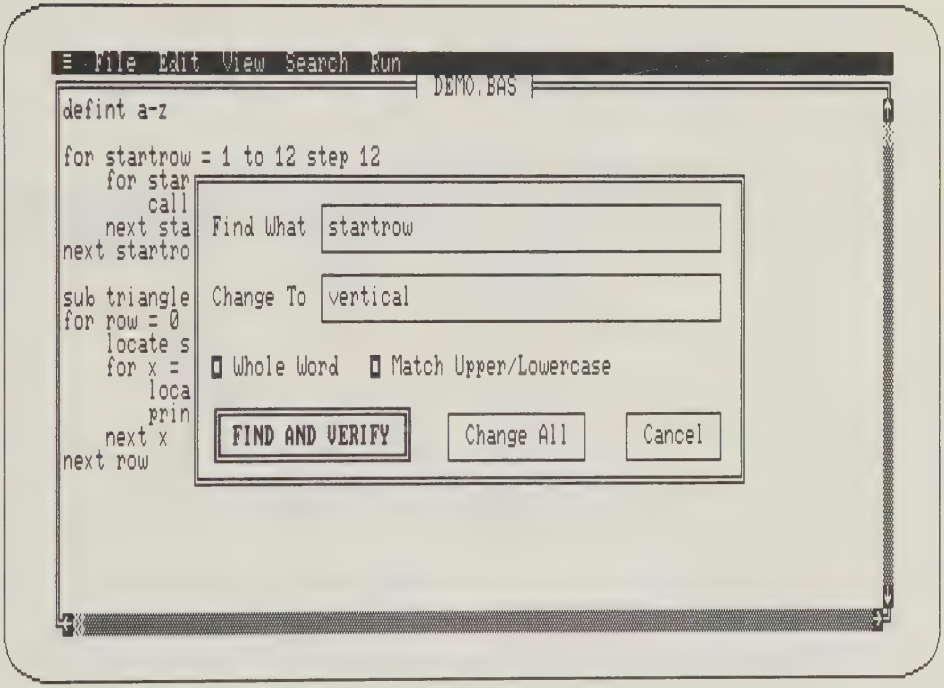
   ```
   vertical
   ```

   in the Change To text box.

6. Press the TAB key three times, until the Find and Verify box has the input focus. When an item has the input focus, the text changes to capital letters and the box is highlighted.

   With the mouse, move the pointer to the Find and Verify box and click.

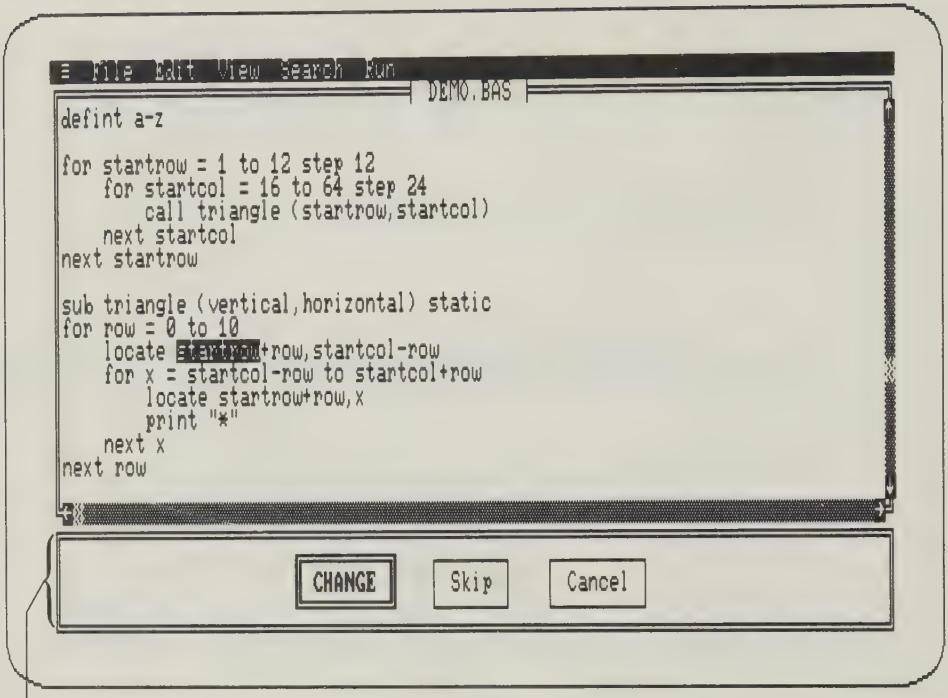
   When you are finished selecting items in the Change... dialog box your screen should look like this:



7. Press SPACEBAR.

   With the mouse, click on the Find and Verify command button.

   The first `startrow` after the **SUB** statement is highlighted, and the Verify dialog box appears:



Verify dialog box

8. Press ENTER to execute the change. The highlight moves to the next occurrence of `startrow`.

   With the mouse, click on the Change command button to execute the change.

9. Repeat step 8, until `startrow` is changed to `vertical` in the last seven lines and the search goes up to the first occurrence of `startrow`.

Important

Don't change any occurrences of `startrow` in the first six lines of the program. If you do, change them back before you compile the program.

---

10. Press TAB until the Cancel command button is highlighted, then press SPACEBAR to return to the editor.

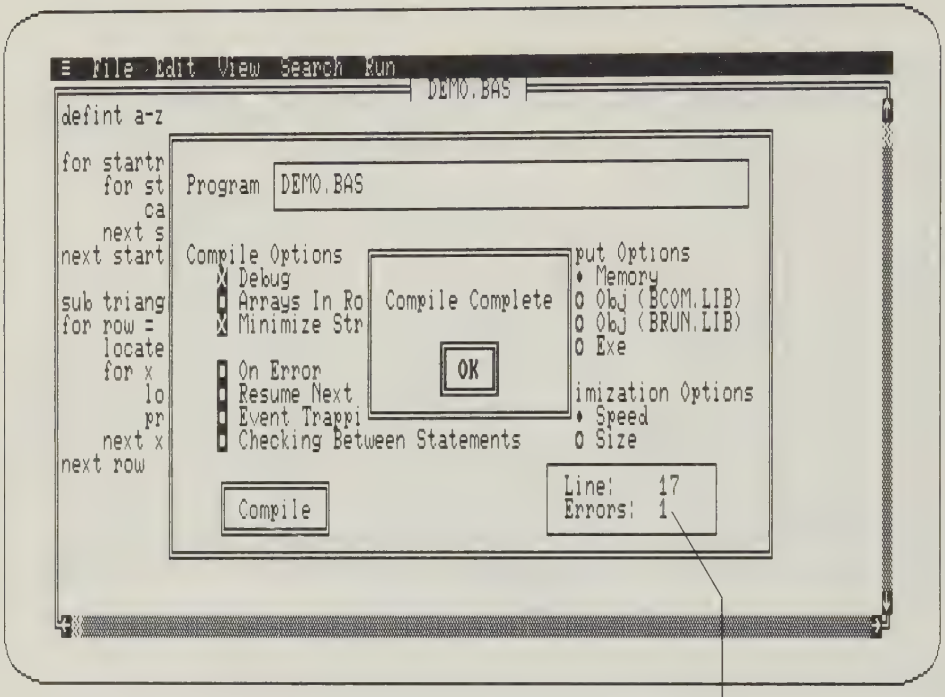
11. Now you want to change `startcol` to `horizontal` in the subprogram. Move the cursor down to the **SUB** statement, and repeat steps 1–10, typing `startcol` in the Find What text box and typing `horizontal` in the Change To text box. Remember, don't make any changes to the first six lines of the program.

Now try recompiling the program. You will get a compilation error. The next section explains how to find and correct it.

3.8.3 Finding and Correcting Errors

Whenever you edit a program, there is the possibility of introducing errors. When the QuickBASIC compiler encounters an error, it increments the "Errors" counter in the Compile display, saves the location of the error in an internal table, and continues compiling the program. When you return to the editor, the error message is displayed in the Error-Message window at the bottom of the screen.

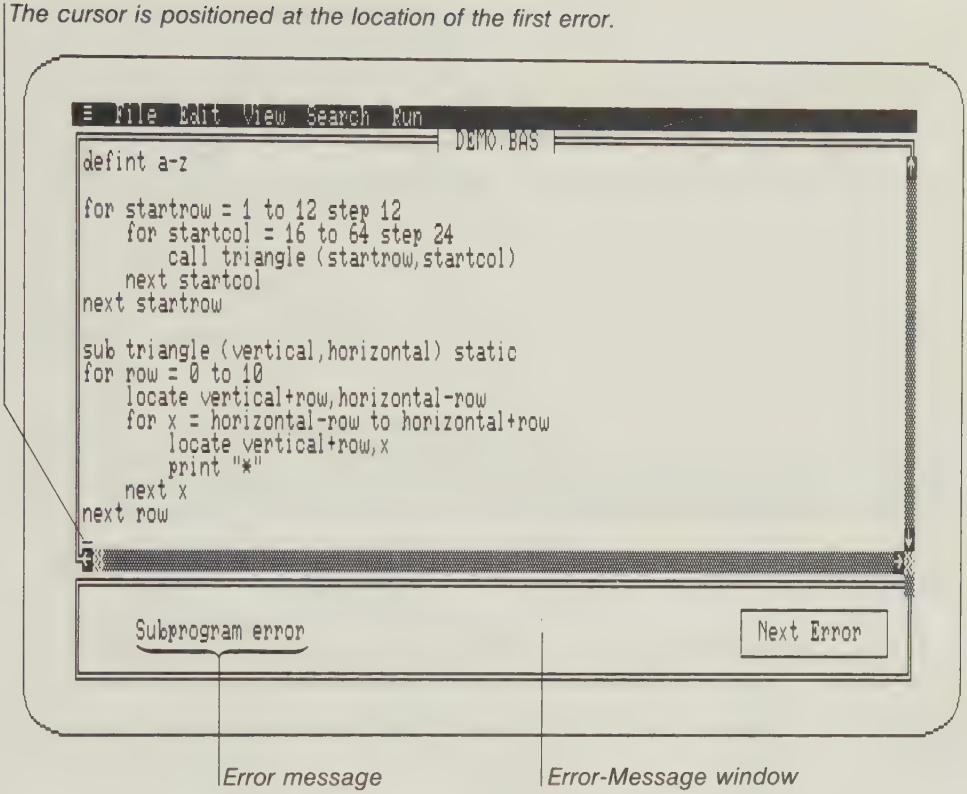
If you followed the above procedure exactly, when you compile the program your screen should look like this:



QuickBASIC found an error during compilation.

If you made a mistake in the practice session, you will see a different number in the box. When you return to the editor you will be at the location of the first error. The error message will be displayed in the Error-Message window at the bottom of the screen.

After compilation is finished, when you press ENTER the Error-Message window opens:



If you are at a different location than pictured above when you return to the editor, press F6 (the shortcut key for the Next Error command) until you reach the last line in the program and the Error-Message window looks like the one shown in the preceding illustration.

The program needs an **END SUB** statement to terminate the subprogram and compile correctly. To add the statement, follow these steps:

1. To create a new line below the current line, press END to move the cursor to the end of the line, then press ENTER.

2. Type

   ```
   end sub
   ```

If there were no other errors in your program, compile and run it with the Compile and Start commands from the Run menu.

If there were other errors in your program, follow these steps to find and correct them:

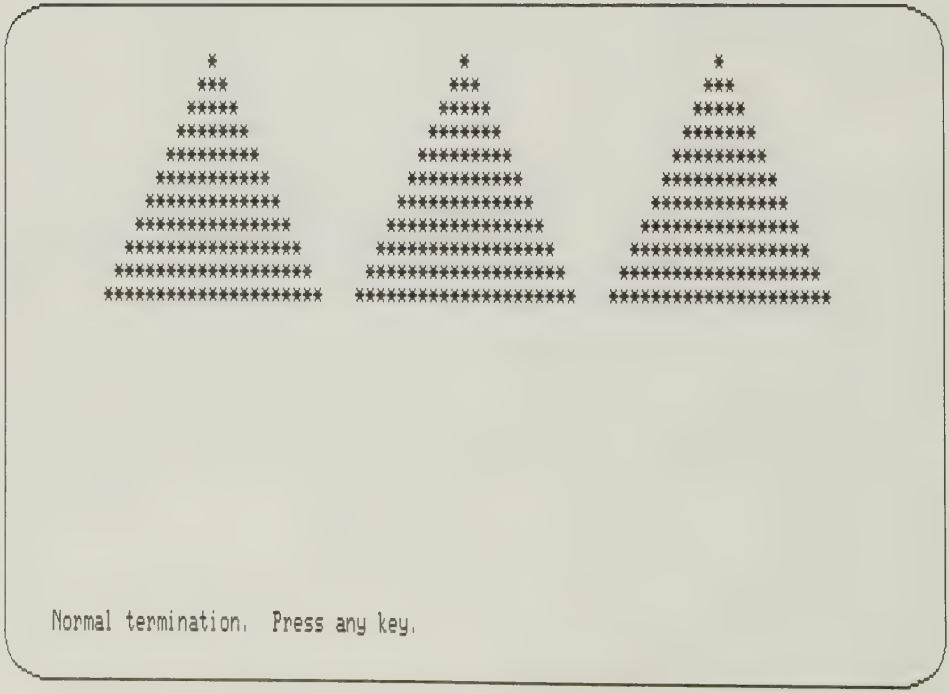
1. To find the next error, press ALT-S to display the Search menu.
2. Choose Next Error.
3. Use the editor to correct the error.



If you have a mouse, click on the Next Error command button to find the next error.

Continue choosing Next Error until the search moves back to the line above the `end sub` statement. Finding and correcting errors are explained in detail in Section 4.4.4, "Handling Compilation Errors," and Section 4.4.5, "Handling Run-Time Errors."

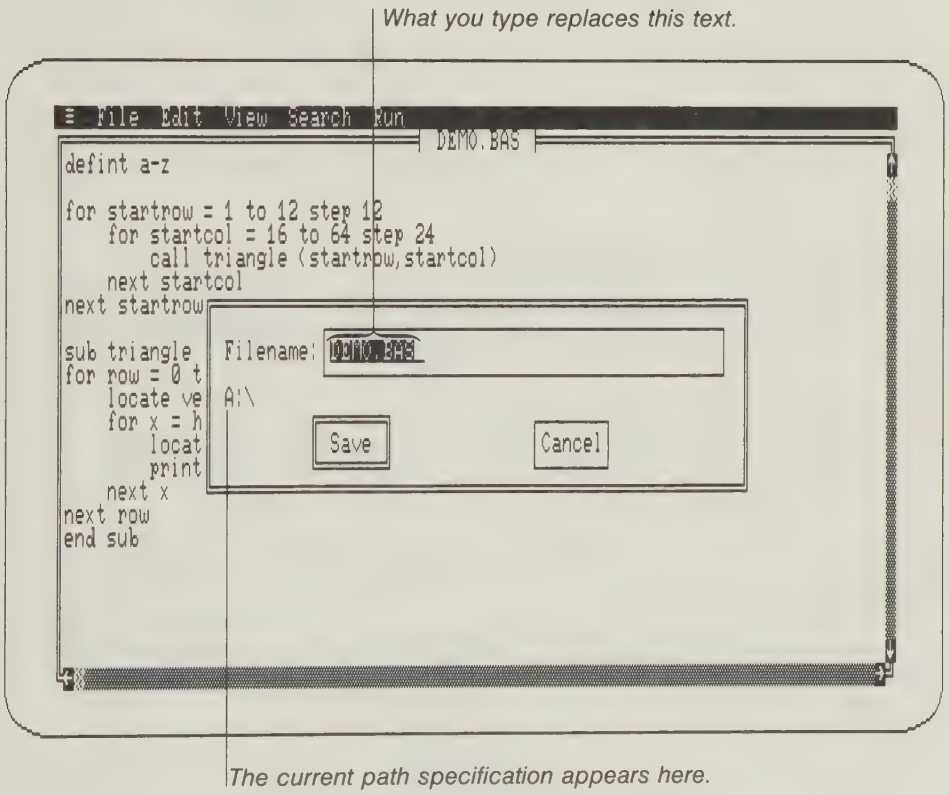
When the program executes, your screen should look like this:



3.8.4 Saving the Program

Use the Save... command to save the new version of the **DEMO.BAS** program as **DEMO2.BAS**. Follow these steps:

1. Press ALT-F to display the File menu.
2. Use the DOWN key to highlight Save... command.
3. Press ENTER.
4. The Save... dialog box opens:



Type

```
demo2.bas
```

to name the program. The cursor will be in the correct position when the dialog box opens.

5. Press ENTER to carry out the command.



To choose Save... with the mouse

1. Move the mouse pointer to the word "File" in the menu bar and hold down the left button.
2. Drag the mouse until the Save... command is highlighted, then release the button.
3. When the dialog box opens, type

   ```
   demo2.bas
   ```

4. Move the pointer to the Save command button and click to execute the command.

3.8.5 Leaving QuickBASIC

To leave QuickBASIC and return to the operating system, follow these steps:

1. Press ALT-F to display the File menu.
2. Use the DOWN DIRECTION key to highlight the Quit command.
3. Press ENTER.



To leave QuickBASIC with the mouse, choose the Quit command from the File menu.

You are now back at system level, and the DOS prompt should be in the upper-left corner of the screen.

3.8.6 Exploring Further

So far in this chapter you have learned about the QuickBASIC screen, and how to edit, compile, and run a source file with QuickBASIC. The practice session describes only a few of the things you can do with QuickBASIC. Chapter 4, "Compiling and Running a Program," and Chapter 5, "Editing and Debugging," explain these capabilities in detail. Both chapters are organized by menus. For instance, to find out how to use the Search commands, see Section 5.3, "The Search Menu."

The next two sections of this chapter are sample sessions for the advanced BASIC user. They will show you how to use the QuickBASIC debugging mode, and how to create a user library. It is recommended that you get some experience compiling and running programs with QuickBASIC, and that you learn more about subprograms before you read these sections.

3.9 Using Debugging Mode

Debugging mode allows you to control the rate of program execution and the amount of source and line number information displayed during execution. Debugging mode is activated when the **TRON** statement is executed. It is turned off with the **TROFF** statement. You can activate debugging mode for part or all of a source file.

To do the exercises in this section you will need the **BUG.BAS** program from product Disk 1.

3.9.1 The Debugging Mode Display

To begin the debugging practice session, start QuickBASIC and load the example program from the disk with the following command:

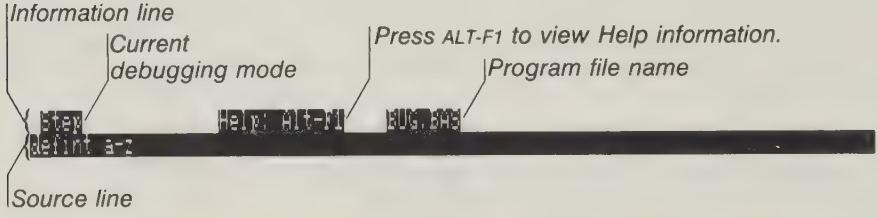
```
qb bug.bas
```

BUG.BAS is the same as the **DEMO2.BAS** program you created in the practice session, but there is a **TRON** statement at the beginning of the program.

Follow these steps to compile **BUG.BAS**:

1. Choose the Compile command from the Run menu. Programs that use debugging mode must be compiled with the Debug option, so make sure that Debug is set.

2. When compilation is finished, press ENTER to return to the editor, then choose Start from the Run menu. As the program begins executing, it enters debugging mode and your screen should look like this:



The two lines at the bottom of the screen are the *information line* and the *source line*. The information line contains the name of the program, the current debugging display mode, and some Help text.

The source line displays the line that is currently executing. If more than one source line is displayed, the line that is currently executing is highlighted.

The complete Help text, which lists the function keys that control debugging mode, can be displayed by pressing ALT-F1. The debugging-mode function keys are discussed in detail in Chapter 5, "Editing and Debugging."

3.9.2 Changing the Display

The ALT-F5 and ALT-F6 keys increase and decrease the number of source lines displayed on the screen. Press ALT-F5 several times to increase the display:

```
 Step          Help: Alt-F1    BUG.BAS
defint a-z

for startrow = 1 to 12 step 12
    for startcol = 16 to 64 step 24
        call triangle (startrow,startcol)
    next startcol
next startrow
```

3.9.3 Executing the Program

When debugging mode starts, it is in Step mode. Step mode executes the program one line at a time, waiting for you to press ALT-F9 before it executes the next line. Press ALT-F9 a few times and watch the highlight move down the screen. As lines are executed, the following output appears:

```
          *
         ***
        *****
       *******
      *********
     ***********
    *************
   ***************
  *****************
 *******************
*********************

 Step          Help: Alt-F1    BUG.BAS
defint a-z

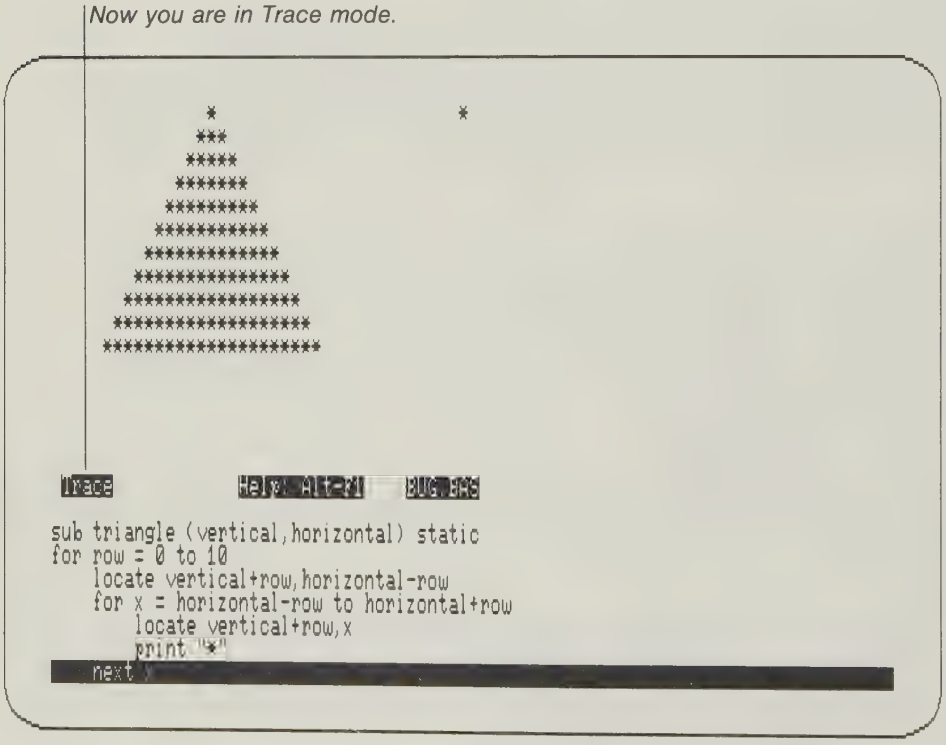
for startrow = 1 to 12 step 12
    for startcol = 16 to 64 step 24
        call triangle (startrow,startcol)
    next startcol
next startrow
```

The executing line is highlighted.

3.9.4 Subprogram Tracing

Like Step mode, Trace mode traces program execution one line at a time. However, Trace mode also traces program execution through user-defined functions and subprograms. The BUG.BAS program contains the subprogram `triangle` that draws the triangles at a location specified in the main program.

Press ALT-F8 to enter Trace mode, and keep pressing it. When the highlight moves into the subprogram, watch the correspondence between execution and the appearance of the output:



3.9.5 Continuous Execution

Animate mode executes the program continuously, but waits briefly after each line. The length of time between lines varies depending on the machine. Press ALT-F10 to enter Animate mode. The default Animate speed is Slow. Try pressing ALT-F10 to toggle the speed to Fast. To exit Animate mode, press ALT-F8 to re-enter Step mode.

3.9.6 Leaving Debugging Mode

To stop execution of the program and exit debugging mode, press CTRL-BREAK. When the program stops you will see the message:

```
Break key pressed. Press any key.
```

3.10 Creating a User Library

The user library contains routines that are called by the main program. User libraries usually contain QuickBASIC subprograms or assembly-language subroutines.

This section assumes you are familiar with QuickBASIC, that you've had some experience with subprograms or assembly-language subroutines, and that you've read Chapter 7, "Creating User Libraries."

In this section you will create a user library containing three subprograms, **triangle**, **square**, and **ex**, that print different shapes. First you will create object files from the BASIC source files, then use the **buildlib** command to place the subprograms in the user library **SHAPES.EXE**. To complete the exercise you will need the files **CALLSHAP.BAS**, **TRIANGLE.BAS**, **SQUARE.BAS**, and **EX.BAS** from product Disk 1.

3.10.1 Creating Object Files

This section describes how to create the object files you will need for the user library. Follow these steps to create the object file:

1. Start QuickBASIC.
2. Use the Load... command from the File menu to load **TRIANGLE.BAS**.
3. Choose Compile... from the Run menu.
4. Select the Obj(BRUN.LIB) option from the dialog box. This option creates an object file (a file that has the base name of the source file and the extension **.OBJ**) and writes it to the disk.
5. Press ENTER to start compiling.
6. When compilation is finished, load **SQUARE.BAS**.
7. Choose Compile from the Run menu. The option you selected in step 4 is still active, so you don't need to choose it again.
8. Repeat steps 6 and 7 for **EX.BAS**.
9. When all three programs are compiled, choose Quit from the File menu.

Use the DOS **DIR** command to display the list of files on the disk. You will see the names **TRIANGLE.OBJ**, **SQUARE.OBJ**, and **EX.OBJ**. These three files make up the user library you will create in the next section.

3.10.2 Building the Library

The **buildlib** command creates user library files and places the files into a single executable file. An internal table is also created that contains information about the names and locations of the contents.

To create a user library file named **SHAPES.EXE**, type

```
buildlib triangle.obj square.obj ex.obj, shapes.exe;
```

When processing is complete, there will be a user library file named **SHAPES.EXE** in the current directory.

3.10.3 Using the Library

When you start QuickBASIC with the /l option, it reads the user library **USERLIB.EXE** into memory. When /l has a *libraryfile* argument, the file named is loaded instead of **USERLIB.EXE**. Programs that require routines from the library will automatically search for them. You don't have to

quit and restart QuickBASIC if your program doesn't require the user library; it is ignored if not needed.

To run a program that uses the library you just created, follow these steps:

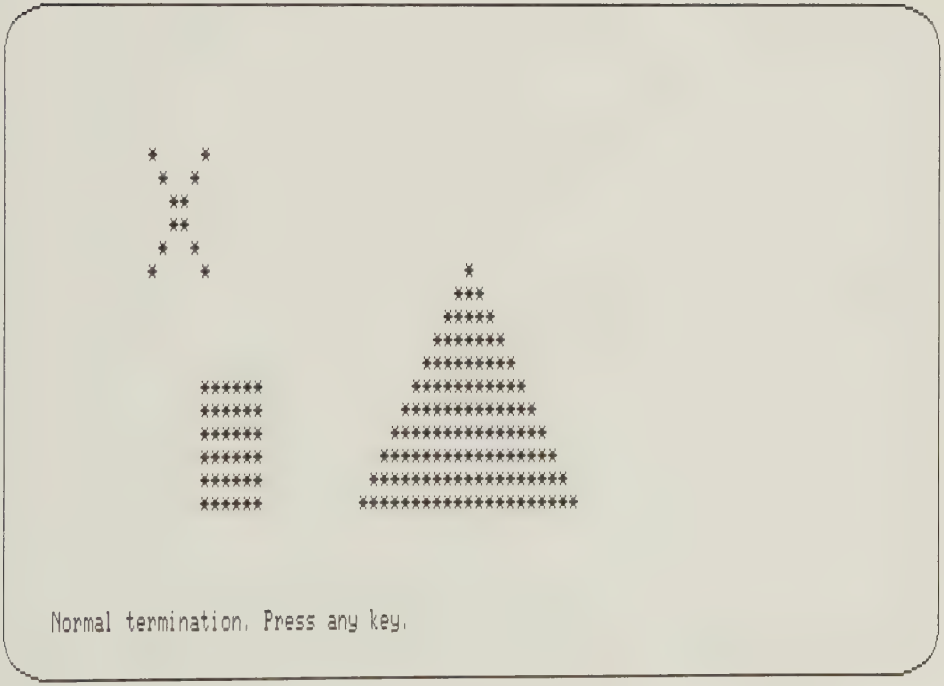
1. Type

   ```
   qb callshap.bas /l shapes.exe
   ```

   CALLSHAP.BAS is the program that calls the user library. It specifies the locations where the called figures will appear.

2. Choose Compile from the Run menu to compile **CALLSHAP.BAS**.

3. Choose Start from the Run menu to execute the program. Your screen should look like this:



Chapter 4
Compiling and Running a Program

4.1 Starting QuickBASIC 71
4.2 The File Menu 74
4.2.1 New 75
4.2.2 Load... 75
4.2.3 Save... 78
4.2.4 Auto Save 79
4.2.5 Print... 79
4.2.6 Shell 81
4.2.7 Quit 81
4.3 The View Menu 81
4.3.1 Options... 82
4.3.2 Errors 84
4.4 The Run Menu 84
4.4.1 Start 85
4.4.2 Compile 85
4.4.3 Compile... 88
4.4.3.1 Debug 90
4.4.3.2 Arrays in Row Order 91
4.4.3.3 Minimize String Data 91
4.4.3.4 On Error 91
4.4.3.5 Resume Next 92
4.4.3.6 Event Trapping 92

4.4.3.7 Checking Between Statements 92
4.4.3.8 Memory 92
4.4.3.9 Obj(BCOM.LIB) 93
4.4.3.10 Obj(BRUN.LIB) 93
4.4.3.11 Exe 93
4.4.3.12 Speed 93
4.4.3.13 Size 94
4.4.4 Handling Compilation Errors 94
4.4.5 Handling Run-Time Errors 95

This chapter describes

- Starting QuickBASIC, and the options available on the start-up command line
- Using the options in the View menu to change the appearance of the QuickBASIC display
- Using the File and Run menus to load, compile, and execute the QuickBASIC program
- Identifying and correcting compile and run-time errors

Menu commands and options are listed in the order they appear in the menu.

Before you begin this chapter you should be familiar with the QuickBASIC screen. You should also understand the techniques for choosing commands from menus, and options from dialog boxes, described in Chapter 3, "Exploring QuickBASIC."

4.1 Starting QuickBASIC

There are two ways to start QuickBASIC:

1. Start the QuickBASIC editor and choose commands from menus to edit, compile, and run your program. This method, described in this chapter, is most useful when you are creating and debugging a program and is recommended for most situations.
2. Start the QuickBASIC compiler from a single command line, bypassing the QuickBASIC editor. This method is most useful for creating executable programs that use the **BCOM20.LIB** library, or when you are compiling with a batch file created for use with QuickBASIC Version 1.0. Programs compiled this way must be explicitly linked. See Appendix D, "Using the Separate Compilation Method," for more information.

The **qb** command starts the QuickBASIC editor and has the following syntax:

qb [[*sourcefile*]] [[**/b**]] [[**/g**]] [[**/l** [*libraryfile*]]] [[**c:***buffersize*]] [[*compile-options*]][[**/cmd** *string*]]

The **qb** command takes the following arguments:

Argument	Description
sourcefile	The file to be loaded when QuickBASIC starts.
/b	Use this option when you are using a monochrome monitor with a color graphics card, or if you don't want to display the QuickBASIC screen in color. Note that you don't need to use **/b** if you don't have a color card; QuickBASIC can detect what type of card you have, but not what type of monitor.
/g	Try this option if you are using a color graphics adapter. QuickBASIC automatically adjusts the way it updates the screen to accommodate your hardware. On some machines it is possible to update the screen faster than the QuickBASIC default. The **/g** option causes QuickBASIC to update the screen as fast as possible. If you see "snow" (dots flickering on the screen) when you update large areas of the screen (for example, with PGUP), then your hardware cannot handle the fast speed and you should restart QuickBASIC without the **/g** option. This option has no effect on machines with enhanced graphics (EGA) or monochrome adapter cards.
/l [*libraryfile*]	Loads the user library file. The *libraryfile* argument is required only if the user library is *not* the default, **USERLIB.EXE**. The user library must be an executable (**.EXE**) file created using the **buildlib** command. See Chapter 7, "Creating User Libraries," for more information.
/c:*buffersize*	Sets the size of the buffer receiving remote data, using an asynchronous communications adapter. (The transmission buffer is allocated 128 bytes and cannot be changed on the **qb** command line.) This option has no effect if no asynchronous communications card is present. The default buffer size is 256 bytes; the maximum size is 32,767 bytes.

compile-options

The *compile-options* are listed in Table 4.1 and can be given on the **qb** command line. You will probably want to choose the *compile-options* from the Compile... dialog box in the QuickBASIC editor, unless you are using the separate compilation method described in Appendix D. Table 4.1 gives the command-line form of each *compile-option*, and its corresponding name from the Compile... dialog box. Choosing compile options from the Compile... dialog box is explained in Section 4.4.3, "Compile...".

/cmd *string*

Passes everything (*string*) from **/cmd** to the end of the line to the program at runtime. The program accesses *string* with the **COMMAND$** function. See Part 2, "Reference," for more information about **COMMAND$**.

If the **COMMAND$** function occurs in a standalone executable program, the function returns the command line used to invoke the program. Standalone executable programs are programs that are executed by entering the base name of the file.

Table 4.1

QuickBASIC Compile Options

Option	Information
/d	Debug, Section 4.4.3.1
/e	On Error, Section 4.4.3.4
/o*	Obj(BCOM.LIB), Section 4.4.3.9 and Section D.2
/q	Speed, Section 4.4.3.12
/r	Arrays in Row Order, Section 4.4.3.2
/s	Turns off the Minimize String Data option, Section 4.4.3.3
/v	Checking Between Statements, Section 4.4.3.7
/w	Event Trapping, Section 4.4.3.6
/x	Resume Next, Section 4.4.3.5

* Useful only with the separate compilation method described in Appendix D.

4.2 The File Menu

When you choose File from the menu bar, the File menu shown in Figure 4.1 opens:

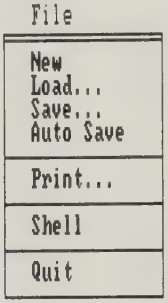


Figure 4.1 The File Menu

The commands in the File menu are described below:

New clears the current program from memory and the current program listing from your screen so you can begin a new program.

Load... copies a program from disk into memory.

Save... writes a program to disk from memory. You can also use this command to rename programs.

Auto Save automatically writes a modified file to disk before any operation that loads a new file or executes a program.

Print... prints all or part of the program in memory on the printer connected to LPT1.

Quit exits QuickBASIC and puts you back at the system prompt.

Shell lets you temporarily exit to DOS, then return to QuickBASIC.

The following sections explain how to use each command in the File menu.

4.2.1 New

The New command clears the program currently in memory so you can start entering a new program.

If you have modified the program currently in memory (or if the program is a new, unnamed program), when you choose New, a dialog box opens that asks if you want to save the file. Press ENTER to save the file with the name specified in the dialog box. If the program is untitled, the Save... dialog box opens when you press ENTER.

If you don't want to save the file, use the TAB key to move the input focus to the No command button, then press SPACEBAR.

To change the file name, use the the Cancel command button to cancel the New command, then use Save... to rename the file. After you've saved the file, choose New again.

4.2.2 Load...

The Load... command places a file from disk into memory. You can also use the Load... command to list the files and directories on your system.

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

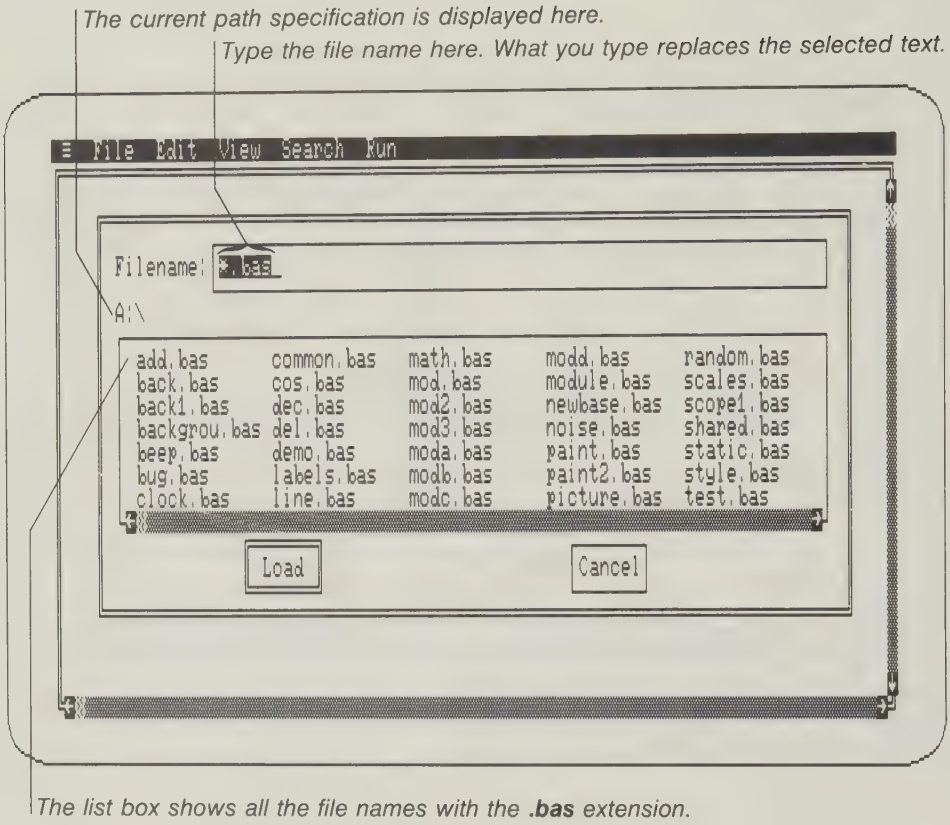


Figure 4.2 The Load... Dialog Box

When the Load... dialog box first appears, the file name `*.bas` appears highlighted in the text box. All the files in the current directory with the extension **.BAS** appear in the list box. If you are in a subdirectory, the entry ".." appears in the upper-right corner of the list box. It represents the directory immediately above the current directory.

Directory and file names are displayed in columns in the list box. When all the names don't fit in the box, use the DIRECTION keys to move the highlight to the edge of the box and keep pressing it to scroll the list. Note that you can only scroll the list box left and right.

If you are using a mouse, use the scroll bar at the bottom of the list box to scroll the list.



There are two ways to specify the file you want to load:

1. Type the name of the file you want to load, then press ENTER. The name appears in the text box.

   QuickBASIC assumes the **.BAS** extension for all file names without extensions. If you want to load a file that has no extension, type a "." immediately after the file name.

2. Press the TAB key to move the input focus to the list box, then use the DIRECTION keys to move through the list box until the file you want to load is highlighted. Press ENTER to load the file.

   The text of the file appears in the work area. Now you can edit or compile the program.

To load a file with the mouse, double click on the file name in the list box.



Listing Files with Load...

You can use the Load... dialog box to list the contents of any directory on your system.

In the list box, file names are displayed in lowercase letters, and directory names are displayed in capital letters. The directory immediately above the current directory is represented by the entry ".." in the upper-right corner of the list box. When a directory is entered in the text box, or selected in the list box, the files in that directory are listed. (When a file name is entered or selected, the file is loaded into memory.) You can use the DOS "wildcard" characters, ? and *, to list groups of files.

Example

To list all the files on the disk or in the current directory, type

```
*.*
```

in the text box, and press ENTER.

To list files in the directory immediately above the current directory, move the highlight to the entry

```
..
```

and press ENTER.

To list all the files on drive A, type

```
A:\*.*
```

in the text box and press ENTER.

Note

You can load any file on your system into QuickBASIC, but only ASCII files are meaningful to the QuickBASIC editor. If you accidentally load an executable or binary file, choose New from the File menu to clear your screen.



If you have a mouse, you can use the current path specification to move through the directory tree. For example, if the current path specification is `C:\QBAS\PROGS\GAMES` , double click on `PROGS` to list the files in that directory. Note that you should click on the path specification displayed *below* the text box, not inside the text box.

4.2.3 Save...

The Save... command writes the program currently in memory to the disk. You can also use the Save... command to change file names.

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

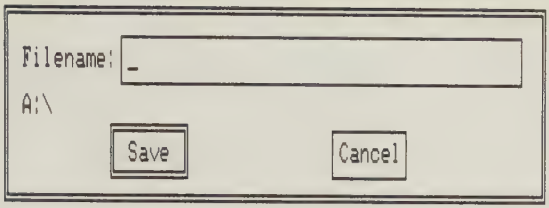


Figure 4.3 The Save... Dialog Box

If you are saving a new, unnamed file, type in the file name. What you type appears in the text box.

If you are saving a file that already has a name, the file name appears selected in the text box. If you want to change the file name, just begin typing. What you type replaces the old name.

4.2.4 Auto Save

The Auto Save command toggles (turns on and off) automatic file saving. A check appears to the left of the command when Auto Save is turned on.

Whenever you attempt an operation that can result in the loss of changes made to a file (running a program, for example), QuickBASIC opens a dialog box that asks if you want to save the program on disk. When Auto Save is turned on, modified files are saved on disk automatically, and the dialog box does not appear.

4.2.5 Print...

The Print... command prints all or part of the program currently in memory.

When you choose the Print... command, the dialog box shown in Figure 4.4 opens:

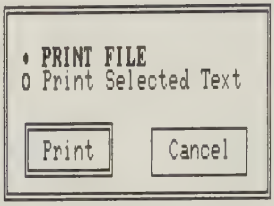


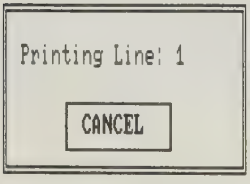
Figure 4.4 The Print... Dialog Box

The dialog box shows the active option. Print File prints the entire file. If no text is selected, this is the default option.

Print Selected Text prints only the selected text in the file. When text is selected, this is the default option.

If you want to print only part of the program, select the text you wish to print before you choose the Print... command. Otherwise the entire file is printed.

While the file is printing, the following message box appears:



Press ENTER to stop printing at any line.



If you have a mouse, click the Cancel command button to stop printing.

QuickBASIC sends Print... command output to LPT1. You can direct the output to another device with the DOS **MODE** command. See your DOS manual for details.

4.2.6 Shell

The Shell command lets you temporarily return to DOS command level, where you can execute other programs and DOS commands. QuickBASIC remains in memory.

To return to QuickBASIC, type

```
exit
```

The QuickBASIC screen appears in the same state it was in when you left.

4.2.7 Quit

The Quit command removes QuickBASIC from memory and returns you to the system prompt.

When the Auto Save command in the file menu is not set, and you have modified an existing program, or the program is a new, unnamed program, choosing the Quit command causes a dialog box to open that asks if you want to save the file. Press ENTER to save the file with the name specified in the dialog box. If the program is untitled, the Save... dialog box will appear.

To change the file name, use the the Cancel command button to cancel the Quit command, then use Save... to rename the file. Choose Quit again to exit QuickBASIC.

With the mouse, click on the No command button to return to system level without saving the file.



If you don't want to save the file, use the TAB key to move the input focus to the No command button, then press SPACEBAR to return to system level.

4.3 The View Menu

The commands in the View Menu let you change the appearance of your display, and control the presence of the Error-Message window. Figure 4.5 shows the View menu.

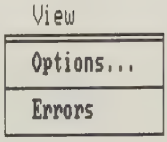


Figure 4.5 The View Menu

The commands in the View menu are described below:

Options... alters the QuickBASIC display to show more program text, changes the foreground and background colors (color monitors only), and sets tab stops.

Errors turns off and on the Error-Message window.

The following sections explain these commands in detail.

4.3.1 Options...

The Options... command lets you alter the QuickBASIC display. When you choose Options... from the View menu, the dialog box shown in Figure 4.6 opens:

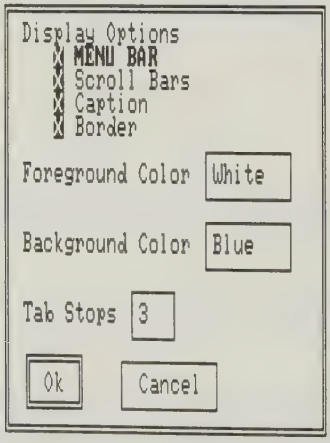


Figure 4.6 The Options... Dialog Box

The Display Options in the Options... dialog box are described below:

Menu Bar toggles the menu bar. "Toggling" reverses the status of the menu bar, turning it off, if it was on, and on, if it was off.

Scroll Bars toggles the scroll bars on the right and bottom edge of the work area.

Caption toggles the caption bar.

Border toggles the border around the work area.

Foreground Color and **Background Color** control the display colors.

Tab Stops specifies the number of columns between tab positions.

All the Display Options are on by default.

Choosing Colors

There are two ways to choose foreground and background colors:

1. While the option has the input focus, type the first letter of the color you want. If you want black or blue, you may have to press B twice.

2. While the option has the input focus, use the UP and DOWN keys to scroll through the color list until the name of the color you want appears in the box. Although you can only see one color name at a time, they are listed in the following order:

   Black
   Blue
   Green
   Cyan
   Red
   Magenta
   Yellow
   White

   For example, if "White" is displayed in the Foreground Color box, pressing the UP key once changes the name in the box to "Yellow."

Color choices are useful only on a color monitor. On a monochrome monitor, if any color except "Black" is selected as the background color the display changes to reverse video. Foreground color selections have no effect on a monochrome monitor.

4.3.2 Errors

The Errors command controls the presence of the Error-Message window. Figure 4.7 shows the Error-Message window.

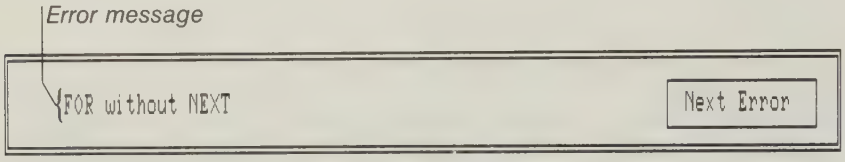


Figure 4.7 The Error-Message Window

If errors are found at compile or run time, the Error-Message window appears when you return to the editor and the cursor is positioned at the location of the first error. If QuickBASIC found more than one error, choose Next Error from the Search menu (or use the shortcut key, F6) to view the next error message.



If you have a mouse, click on the Next Error command button to move to the next error.

To remove the Error-Message window from the display, choose the Errors command. This is useful for displaying more program text. Even if the window is turned off, executing the Next Error command moves the cursor to the next error location. You can view the error message by turning the Error-Message window back on.

When the Error-Message window is turned on, a check appears to the left of the word "Errors" in the View menu.

4.4 The Run Menu

You compile and execute programs with commands from the Run menu, shown in Figure 4.8.

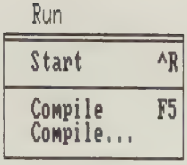


Figure 4.8 The Run Menu

The commands in the Run menu are described as follows:

Start executes the compiled program currently in memory.

Compile compiles a program with the current option settings.

Compile... lets you select options, then compile the program.

The shortcut keys let you choose the command without pulling down the menu. For example, to start compiling simply press F5.

The following sections describe the commands in the Run menu in detail.

4.4.1 Start

The Start command executes a compiled program. If the program contains a **TRON** statement, and was compiled with the Debug option, the program enters QuickBASIC debugging mode when **TRON** is executed. Debugging mode is described in Chapter 5, "Editing and Debugging."

You can also use the Start command to both compile and execute a program. When you choose Start, it checks to see if the program has been modified. If it has, QuickBASIC begins compiling using the current option settings. Execution begins automatically when you press ENTER in response to the Compile Complete message box.

4.4.2 Compile

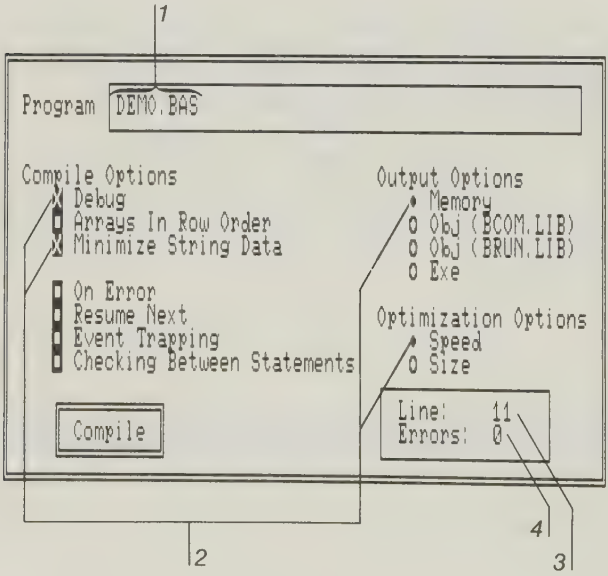
The Compile command compiles a program using the default compiler settings, the options set in a previous compilation, or the options set on the **qb** command line.

When the Auto Save command is not set, and you have modified the program since the last compilation, a dialog box appears asking if you want to save the file. The compiler processes the file as it exists in memory, so unless you are editing an **$INCLUDE** file, you probably won't want to save the file until it compiles successfully.

Important

If you are editing a file that is called from another program by the **$INCLUDE** metacommand, you should always save it on the disk before you compile the program that calls it, or you will lose the changes you made. When the program specified in the Compile display is not the program currently in memory, a dialog box appears asking if you want to save the current program. When compilation begins, the current program is erased from memory and replaced with the program named in the Program text box.

During compilation, your screen will look like this:



The elements of the display are explained below:

1. The name of the file being compiled. It is either the name of the last file loaded or of the last file compiled.
2. The current option settings. Debug, Minimize String Data, Memory, and Speed are on by default. (The options are explained in Section 4.4.3, "Compile...".) If you previously compiled with different options, the option settings from the previous compilation are still set.
3. The current line that is being compiled. This number increments rapidly.
4. The number of errors detected.

When compilation is complete the following message box opens:



To return to the editor, press ENTER.

If you have a mouse, click OK to return to the editor.



If no errors were found, QuickBASIC will return to the location you were at when you compiled the program. If errors were detected during compilation, when you return to the editor, the Error-Message window appears and the cursor is positioned at the location of the first error.

Section 4.4.4, "Handling Compilation Errors," describes the procedure for finding and correcting compilation errors.

4.4.3 Compile...

The Compile... command compiles the program with the specified options. The individual options are described in Sections 4.4.3.1–4.4.3.13. You need to use the compile options if you want to

- Trap events
- Use error-handling statements
- Store arrays in row order
- Write string data to the symbol table instead of storing it in memory
- Produce an object file
- Produce an executable file
- Optimize the executable program to make it smaller or faster
- Use debugging mode (**TRON**)

When you select Compile... from the Run menu, the dialog box shown in Figure 4.9 opens:

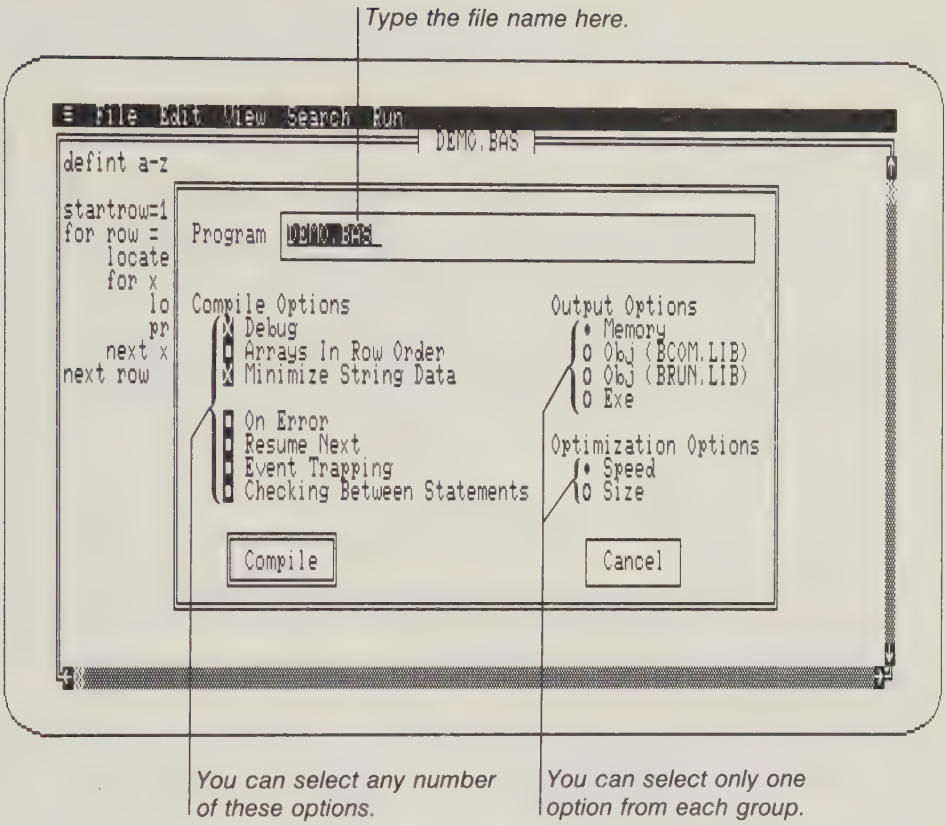


Figure 4.9 The Compile... Dialog Box

The name of the last file compiled, or the last file loaded, appears in the text box. If you are compiling a new, unnamed file, no name will appear in the text box; press TAB to move to the Compile Options.

Use the TAB key to move between items in the dialog box, and the UP and DOWN keys to move among the Output and Optimization Options. Press SPACEBAR to select an item.

You can choose more than one Compile Option, but only one Output Option and one Optimization Option. Some options imply other options. For example, programs that contain **RESUME** statements also contain **ON ERROR** statements, so QuickBASIC automatically turns on the

On Error option when Resume Next is chosen, and turns off Resume Next if On Error is turned off. The same is true for the Event Trapping and Checking Between Statements options.

4.4.3.1 Debug

The Debug option causes the compiler to include code in the executable program that generates error-handling messages and reports error locations at run time. Debug also supports the use of **TRON** and **TROFF**. If the Debug option is not used, **TRON** and **TROFF** are ignored.

The Debug option is on by default. When Debug is on, the QuickBASIC compiler generates larger and slower code that checks the following:

- Arithmetic overflow

  All arithmetic operations, both integer and floating-point, are checked for overflow and underflow.

- Array bounds

  All array references are checked to see if subscripts are within the bounds specified in **DIM** statements.

- Line locations

  The generated binary code includes additional tables so that run-time error messages can indicate the lines on which errors occur.

- **RETURN** statements

  Each **RETURN** statement is checked for a prior **GOSUB** statement.

- CTRL-BREAK

  After executing each line, the compiler checks to see if you have pressed CTRL-BREAK. If CTRL-BREAK is pressed, program execution stops. When you return to the editor, the cursor is positioned on the line where the break occurred.

If you don't use the Debug option, then array bound errors, **RETURN** without **GOSUB** errors, and arithmetic overflow errors do not generate error messages at compile or run time. Program results may therefore be unpredictable.

Note that if a program is not compiled with the Debug option, a user cannot exit the program by pressing CTRL-BREAK, except when entering data in response to an **INPUT** statement prompt. To exit the program the user must restart the computer.

4.4.3.2 Arrays in Row Order

The compiler normally stores arrays in column order. For example, the element `ARRAY(2,1)` is followed by `ARRAY(3,1)`. The Arrays in Row Order option instructs the compiler to store arrays in row order, where the element `ARRAY(2,1)` would be followed by `ARRAY(2,2)`. Since the BASIC interpreter also stores and accesses arrays in column order, you won't need this option for compiling interpreted BASIC programs. This option is useful if you are using assembly-language routines that store arrays in row order.

4.4.3.3 Minimize String Data

In order to minimize the amount of space that string literals (strings of characters used in **PRINT** and **INPUT** statements, for example) occupy in memory, when the same string literal appears twice in a program, the compiler references the same memory location. This means the compiler must retain all string literals in its data area throughout compilation. Normally this is not a problem, but in programs with many string literals, this optimization can cause a "Program memory overflow" error. If you get such an error, try turning off the Minimize String Data option.

4.4.3.4 On Error

Use the On Error option when the program contains an **ON ERROR GOTO...RESUME** *linenumber* construction. This option causes the compiler to generate the location information required by the **GOTO** and **RETURN** statements. When the On Error option is set, the compiler also creates an internal line-number-address table so each run-time error message can include the line number where the error occurs. Because the location information and line number data created by this option result in a larger executable file, do not use On Error unless the program contains an **ON ERROR GOTO** statement.

If the program contains both **ON ERROR GOTO** and **RESUME NEXT** or **RESUME 0** statements, use the Resume Next option.

4.4.3.5 Resume Next

Use the Resume Next option when the program contains one or more **RESUME**, **RESUME NEXT**, or **RESUME 0** statements.

Because the compiler must generate more location information for these statements than for **RESUME** *linenumber*, programs compiled with the Resume Next option take longer to compile and produce larger executable programs than those compiled with the On Error option only. If the program contains only **RESUME** *linenumber* statements, use the On Error option instead of the Resume Next option.

Note, however, that it is considered better programming practice to use the generalized error processing provided by **RESUME**, **RESUME NEXT**, and **RESUME 0**. Programs that use these statements instead of **RESUME** *linenumber* are often smaller to begin with, which can offset the size disadvantage of the Resume Next option.

4.4.3.6 Event Trapping

The Event Trapping option enables event trapping for communications (**COM**), lightpen (**PEN**), joystick (**STRIG**), timer (**TIMER**), and function keys (**KEY**). It causes checking after each line in the program to see whether an event has occurred for any of these devices.

4.4.3.7 Checking Between Statements

The Checking Between Statements option enables event trapping, like the Event Trapping option, but it causes checking after each statement instead of each line.

4.4.3.8 Memory

The Memory option is the default. With it, the program compiles directly into memory, without creating an executable program on the disk.

4.4.3.9 Obj(BCOM.LIB)

The Obj(BCOM.LIB) option produces an object file and writes it to the disk; it does not create an executable program. It is useful only when you are creating object files for use in standalone executable programs, as described in Appendix D, "Using the Separate Compilation Method." If you want to produce object files for the user library, use the Obj(BRUN.LIB) option described in the following section.

In programs created with the Obj(BCOM.LIB) option, run-time routines are invoked through intersegment calls. See Appendix D, "Using the Separate Compilation Method," for more information about this option.

4.4.3.10 Obj(BRUN.LIB)

The Obj(BRUN.LIB) option produces an object file and writes it to the disk; it does not create an executable program. Use this option to create object files for inclusion in the user library.

In programs created with the Obj(BRUN.LIB) option, run-time routines are invoked through software interrupts. If a user library was loaded when the object file was created, this library must be present at run time. If you are creating standalone executable programs from object files produced by the Obj(BRUN.LIB) option, the presence of the **BRUN20.EXE** run-time module is required at run time. See Appendix D, "Using the Separate Compilation Method," for more information about this option.

4.4.3.11 Exe

The Exe option creates an executable file with the base name of the source file and the extension **.EXE**. Use this option to create files that are the object of the **CHAIN** statement, or to create programs that are executed by entering the base name of the source file.

4.4.3.12 Speed

Use the Speed option to produce a program that runs as quickly as possible. When the Speed option is used, calls to run-time support are optimized to be fast, resulting in a larger executable program.

4.4.3.13 Size

Use the Size option to produce a program that is as small as possible. When the Size option is used, calls to run-time support are optimized to occupy less space in memory, possibly resulting in somewhat slower execution time.

4.4.4 Handling Compilation Errors

If errors are found during compilation, when you return to the editor the cursor is positioned at the location of the first error. Use the following procedure to correct compilation errors:

- Correct the error.
- Choose Next Error from the Search menu (or press the shortcut key F6) to locate the next error.



  If you are using a mouse, click on the Next Error command button in the Error-Message window to locate the next error.

  When all errors have been located, the message "No more errors" appears in the Error-Message window.

- When you have corrected all the errors, compile the program again. If no errors are found during compilation, the Error-Message window will disappear.

QuickBASIC maintains the errors found during compilation in a special table, up to a maximum of 25 errors. It keeps track of modifications to the source file, so QuickBASIC points to the correct location regardless of changes to the source file. Correcting errors does not remove them from the table. If you return to the beginning of the file after the "No more errors" message, then choose Next Error, the cursor is positioned at the location of the first error even though the error has been corrected. An exception to this is when a line is deleted; deleting a line removes its entry from the error table. Otherwise, the table is not cleared until another compile is executed.

The Errors command in the View menu toggles the Error-Message window. The Error-Message window is on by default. To remove the window from the screen and display more text, choose the Errors command. Choose Errors again to turn the window back on. You can move to the next error without turning the window back on.

During compilation, system errors can occur that are unrelated to the contents of your program. For example, the disk may fill up. When these errors occur, compilation stops and the Compile Complete box appears. When you press ENTER, a message box opens containing the words "CRITICAL ERROR" and the error message. The error messages are described in Appendix E, "Error Messages." Press ENTER to return to the editor, and recompile the program.

4.4.5 Handling Run-Time Errors

When errors are detected at run time in programs compiled and executed in the QuickBASIC environment, the following message appears at the bottom of the screen:

```
Runtime error. Press any key.
```

Pressing any key returns you to the QuickBASIC editor. If the program was compiled with the Debug option, the cursor is positioned at the location of the error, and the Error-Message window displays the error message. When errors are found in programs compiled without the Debug option, a dialog box appears reminding you to compile with the Debug option to obtain error-location information.

Chapter 5
Editing and Debugging

5.1 The QuickBASIC Editor 99
5.2 The Edit Menu 101
5.2.1 Undo 102
5.2.2 Cut and Copy 103
5.2.3 Paste 103
5.3 The Search Menu 104
5.3.1 Find... 105
5.3.2 Selected Text 106
5.3.3 Repeat Last Find 106
5.3.4 Change... 107
5.3.5 Label 108
5.3.6 Next Error 109
5.4 Debugging Mode 109
5.4.1 The Debugging Mode Display 110
5.4.2 Step, Trace, and Animate 111
5.4.3 Leaving Debugging Mode 111
5.4.4 Debugging Mode Function Key Summary 112

This chapter describes how to enter and edit program text, using the QuickBASIC editor, and the Edit and Search menus. It also describes the use of QuickBASIC's debugging mode to find and correct program errors.

Before you begin this chapter you should be familiar with the QuickBASIC screen, and the techniques for choosing commands and selecting text described in Chapter 3, "Exploring QuickBASIC."

5.1 The QuickBASIC Editor

When you start QuickBASIC you are in the editor. While you are in the editor you can choose commands from menus; enter new program text; move the cursor around on the screen; and insert, delete, and replace text.

This section describes the keyboard editing functions. Section 5.2 explains how to use the commands in the Edit menu to delete, move, insert, and replace text. Each QuickBASIC editing task requires three steps:

1. Moving the cursor
2. Selecting text
3. Inserting, deleting, or replacing text

Table 5.1 describes the cursor movement, text selection, and editing functions you can perform with the keyboard.

Table 5.1
The QuickBASIC Editor

To Move the Cursor:	Press:
Up/down one line	UP/DOWN* or CTRL-K/J
Right/left one character	RIGHT/LEFT* or CTRL-L/H
Right/left one word	CTRL-RIGHT/LEFT
To the beginning/end of the line	HOME/END
To the top/bottom of the screen	CTRL-HOME/END
To the beginning/end of the file	CTRL-PGUP/PGDN
To Scroll:	**Press:**
Up/down one line	CTRL-UP/DOWN
Up/down one screen	PGUP/PGDN
To Select Text:	**Press:**
A character	SHIFT-LEFT/RIGHT
A line	SHIFT-UP or SHIFT-DOWN
A word	SHIFT-CTRL-LEFT/RIGHT
To Insert:	**Press:**
A line below the current line	CTRL-B
A line above the current line	CTRL-N
To Delete:	**Press:**
Delete the current line	CTRL-Y
Delete to end of line	CTRL-D
Delete character left of cursor	BACKSPACE
Delete character under cursor	SHIFT-BACKSPACE
To Start a New Line:	**Press:**
At the current location	ENTER

* RIGHT, LEFT, UP, and DOWN refer to the DIRECTION keys.



Using the mouse, you can move the cursor and select text quickly, then use the keyboard to perform the desired editing function.

To move the cursor with the mouse, position the mouse pointer where you want the cursor and click the left button. Table 5.2 describes ways to select text with the mouse.

Table 5.2

Selecting Text with a Mouse

To Select:	Do This:
A character	Point to the character to the right of the character you want to select, press the left button, and drag the mouse left, over the character or characters you want to select. Dragging right selects the character the cursor is on.
A word	Double click on the word.
A line	Point to the first column of the line, press the left button, drag the mouse down one line and release the button. Dragging up selects the line above the cursor.
Several lines	Point to the first line you want to select, then press the button and drag the mouse up or down through the lines you want to select.

If you make a mistake while selecting text, click on the selected text or press ESC to cancel the selection.

5.2 The Edit Menu

Figure 5.1 shows the Edit menu. The commands in the Edit menu let you delete, move, or copy text from your program, and undo modifications to the current line.

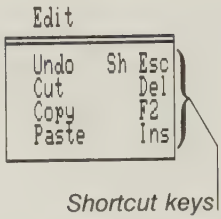


Figure 5.1 The Edit Menu

The commands in the Edit menu are described below:

Undo reverses any modifications to the current line.

Cut deletes text and places it in the "Clipboard." The Clipboard is a temporary storage area QuickBASIC uses to hold text that is copied or deleted from the work area.

Copy makes a copy of the text and places it on the Clipboard, leaving the copied text in its original location.

Paste inserts text from the Clipboard into your program.

The shortcut keys let you choose a command without pulling down the menu. For example, to copy text into the Clipboard simply press F2.

Undo is disabled (appears grey) until the current line is modified. The Cut and Copy commands are disabled until text is selected, and Paste is disabled until text is placed in the Clipboard. If you choose an Edit command while it is disabled, nothing happens.

5.2.1 Undo

The Undo command restores the current line to its original state, reversing all modifications. You can restore a line as long as the cursor is on it. When you move the cursor off the line, the changes are fixed.

Because CTRL-Y removes the entire line, you cannot restore lines deleted by using CTRL-Y with Undo. Text deleted with CTRL-Y is placed in the Clipboard, however, so you can restore it with the Paste command.

5.2.2 Cut and Copy

The Cut and Copy commands put text in the Clipboard. The Cut command deletes the selected text and places it in the Clipboard. The Copy command makes a copy of the selected text and places it in the Clipboard, leaving the selected text in its original location.

To use the Cut or Copy commands, follow these steps:

1. Select the text you wish to place in the Clipboard.
2. Choose Cut or Copy from the Edit menu.

Important

Text does not accumulate in the Clipboard. What you place in the Clipboard replaces whatever was there.

5.2.3 Paste

The Paste command inserts text from the Clipboard. If text in the work area is selected, the contents of the Clipboard replaces the selected text. If no text is selected, the Paste command inserts the contents of the Clipboard to the left of the cursor. You can paste the same information from the Clipboard as many times as you want.

To insert text from the Clipboard, follow these steps:

1. Move the cursor to the place you want the text inserted. It will be inserted to the left of the cursor. If you want to replace text with the contents of the Clipboard, select the text to be replaced.
2. Choose Paste from the Edit menu.

Note

Text remains in the Clipboard until you end your QuickBASIC session. (The New and Load... commands don't erase the Clipboard.) Cutting or copying a large block of text consumes a great deal of memory. When you use the Clipboard for large blocks of text, it is a good idea to erase the Clipboard afterward by copying something small into it, such as a single character.

5.3 The Search Menu

Figure 5.2 shows the Search menu. You can find specific parts of your program using commands from the Search menu.

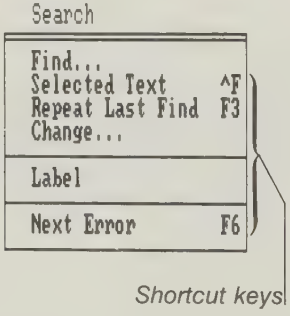


Figure 5.2 The Search Menu

The commands in the Search menu are described below:

Find... finds the first occurrence of the text you specify.

Selected Text finds text you selected from the work area.

Repeat Last Find finds the next occurrence of the text you specified in a previous search command.

Change... finds specified text and replaces it with different text.

Label appends a colon to the text you specify and finds a label with that name.

Next Error locates the next line containing a program error found when you last compiled or ran the program.

The shortcut keys let you execute a Search command without displaying the menu. For example, pressing F6 executes Next Error.

Searches begin at the cursor and continue until the specified text is found or the starting point of the search is reached. All the commands in the Search menu use the last text searched for as the object of the search, unless you specify otherwise.

The following sections describe the Search commands in detail.

5.3.1 Find...

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

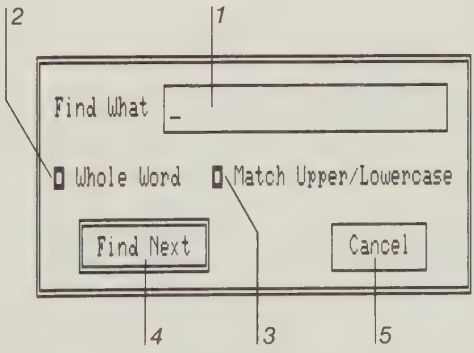


Figure 5.3 The Find... Dialog Box

The elements in the Find... dialog box are explained below:

1. You type the text you want to search for in the Find What text box. The text you last searched for appears highlighted in the text box.
2. The Whole Word option finds the specified text only when it is surrounded by spaces, punctuation marks, or other characters not considered parts of a word. The characters A–Z, a–z, 0–9, and !, #, $, %, and & (commonly used in variable declarations) are considered parts of a word. For example, if CHR is the specified text, the Whole Word option will locate CHR: but not CHR$.
3. The Match Upper/Lowercase option finds only instances of the specified text that match exactly. For example, if CHR is the specified text, the Match Upper/Lowercase option will find CHR but not Chr.
4. Find Next searches for the next occurrence of the specified text.
5. The Cancel command button cancels the selections in the Find... dialog box and returns to the work area.

If the text isn't found, a box appears that says "Text Not Found." Press ENTER to remove the box and continue.



If you are using a mouse, clicking OK lets you continue.

5.3.2 Selected Text

You can also search for text in a program with the Selected Text command. Selected Text looks for text you highlight in the work area. To use the Selected Text command, follow these steps:

1. Select the text you wish to search for. Note that the selected text must be on a single line.
2. Choose the Selected Text command from the Search menu.

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

5.3.3 Repeat Last Find

Repeat Last Find searches for the text specified in the previous Find... command.

5.3.4 Change...

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

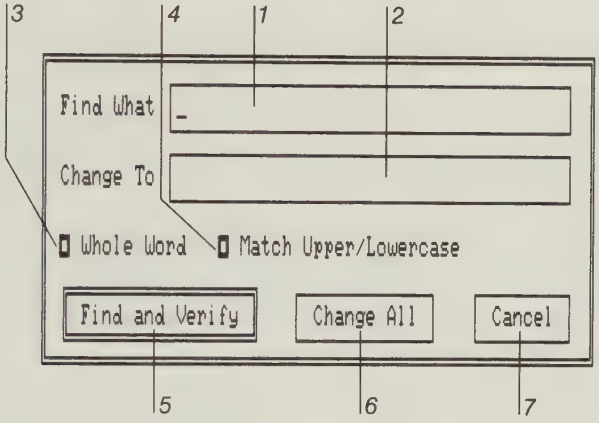


Figure 5.4 The Change... Dialog Box

The elements of the Change... dialog box are explained below:

1. Type the text you want to change in the Find What text box. When you choose the Change... command, whatever text you last searched for appears highlighted in the text box.

2. Type the replacement text in the Change To text box.

3. The Whole Word option finds and replaces the specified text only when it is surrounded by spaces, punctuation marks, or other characters not considered parts of a word. The characters A–Z, a–z, 0–9, and !, #, $, %, and & (commonly used in variable declarations) are considered parts of a word. For example, if CHR is the specified text, the Whole Word option will locate CHR: but not CHR$.

4. The Match Upper/Lowercase option finds and replaces the specified text only when the case of the letters matches exactly.

5. The Find and Verify command button finds the text, then displays three more command buttons; Change, Skip, and Cancel are shown below:



   The Change command button executes the change, Skip searches for the next occurrence of the specified text (without changing anything), and Cancel stops the search and returns to the work area.

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

7. The Cancel command button cancels Search and places you back in the work area.

If you make a mistake, use Load... from the File menu to reload the file, then start the search and replace sequence again.

5.3.5 Label

The Label command finds a line label by appending a colon to the text you specified in the previous Find... command.

For example, if `start` is the text you named in a previous Find... command, choosing Label causes QuickBASIC to search for the `start:` line label.

5.3.6 Next Error

When you compile a program, QuickBASIC "remembers" where the errors are, up to the maximum allowed (25) before compilation stops. The Next Error command finds each line containing an error, starting at the beginning of the program. To locate the lines containing errors, follow these steps:

1. Choose Next Error from the Search menu. The cursor moves to the line that contains the first error.
2. Correct the error.
3. Choose Next Error again.

If the Error-Message window is present, repeat the above steps until the window contains the message "No More Errors." If the Error-Message window is not visible, choosing Next Error after the last error is located will position the cursor back at the location of the first error.

5.4 Debugging Mode

This section describes QuickBASIC's debugging mode. You use debugging mode to find and correct errors that occur during program execution. There is a debugging mode practice session in Section 3.9, "Using Debugging Mode."

QuickBASIC's debugging mode displays line number and source code information during program execution, and allows you to control the rate at which the program executes. Debugging mode is activated at run time by the execution of the **TRON** statement in the source file, and turned off by the **TROFF** statement. You can activate debugging mode for all or part of a BASIC source file by bracketing selected areas with **TRON** and **TROFF** statements. Note that programs that contain **TRON** and **TROFF** statements must be compiled with the Debug option.

5.4.1 The Debugging Mode Display

The debugging mode display begins at the first program line after the **TRON** statement. When the program enters debugging mode the screen looks like Figure 5.5:

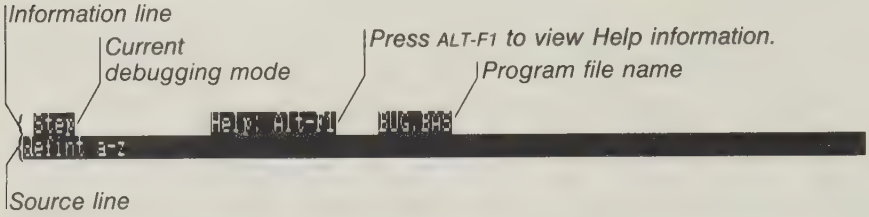


Figure 5.5 The Debugging Mode Display

The *source line* in the debugging mode display shows the line that is currently executing. You can increase the number of source lines displayed to 24 or decrease it to 1 with the ALT-F5 and ALT-F6 keys. If you decrease the display so the information line is the only line on the screen, then press ALT-F6 again, the current program line is displayed. The debugging mode display uses the program's current screen mode. This means that if the program is executing in 40-column text mode, only the first 40 columns of program text are displayed.

The *information line* displays the name of the program and information about the current debugging mode. The last item on the information line is the program name.

The ALT and function keys control debugging mode. You can suspend execution and display a list of the function keys and their uses by pressing ALT-F1. To reactivate debugging mode, press ALT-F1 again. Section 5.4.4, "Debugging Mode Function Key Summary," summarizes the key sequences and their functions.

Important

Debugging mode prevents programs from altering the debugging mode display. Programs that contain graphics statements such as **GET** and **PUT** may return an error or behave unpredictably if the program's screen output intersects the debugging mode display.

5.4.2 Step, Trace, and Animate

Debugging mode has three "submodes" that control the rate and scope of program execution:

1. "Step" executes the program one line at a time. This is the default when debugging mode starts. Pressing ALT-F9 executes the next line.
2. Like Step, "Trace" executes the program one line at a time, but follows program execution through user-defined functions, subprograms, and subroutines. Trace is toggled by ALT-F8.
3. "Animate" executes the program continuously. Animate has two speeds: Slow and Fast. Slow speed (the default) waits a few seconds between each line before executing the next line. Fast executes each line with no delay. Pressing ALT-F10 starts Animate, and toggles the speed. To exit Animate, press the ALT key sequence of the mode you want to enter.

5.4.3 Leaving Debugging Mode

To get out of debugging mode, press CTRL-BREAK. Program execution stops and you are placed back in the editor at the line where the break occurred.

5.4.4 Debugging Mode Function Key Summary

Table 5.3 summarizes the key sequences that control debugging mode.

Table 5.3
Debugging Mode Function Keys

Key	Function
ALT-F1	Toggles the Help line. The Help line lists each function key and its use.
ALT-F5	Increases the number of source lines shown on the screen. Up to 24 source lines can be displayed. Each time you press ALT-F5 one line is added to the display.
ALT-F6	Decreases the number of source lines shown on the screen. Each time you press ALT-F6 one line is removed from the display.
ALT-F7	Clears the debugging display and executes the program continuously. This key is a toggle, and pressing F7 again redraws the screen and returns to the previous mode.
ALT-F8	Activates trace mode, or executes the next line.
ALT-F9	Activates Step mode, or executes the next line.
ALT-F10	Toggles Animate mode and the Animate speed settings.

Chapter 6
Using Subprograms

6.1 Declaring Subprograms 115
6.2 Calling Subprograms 117
6.3 Passing Variables with CALL 118
6.4 Passing Arrays with CALL 119
6.5 Accessing Parameters with SHARED 121
6.5.1 Sharing from the Main Program: the SHARED Attribute 122
6.5.2 Overriding a SHARED Variable with a STATIC Statement 123
6.5.3 Sharing from the Subprogram: the SHARED Statement 124
6.6 Subprograms in User Libraries 125
6.6.1 Naming Restrictions 125
6.6.2 Passing Parameters 125
6.6.3 Using Named COMMON 126
6.7 Common Errors 126
6.7.1 Argument and Parameter List Errors 126
6.7.2 Variable Aliasing 127
6.7.3 Using GOSUB...RETURN and GOTO Statements in Subprograms 128
6.7.4 Error- and Event-Handling Routines 128

Programs, especially large programs, are easier to maintain and debug if they are divided into smaller, more manageable parts. Creating your own functions with the **DEF FN** statement is one way to divide your programs into separate pieces. QuickBASIC provides another way to segment programs, the *subprogram*. Subprograms are different from functions because they do not return a value that is associated with the subprogram name (and thus cannot be used in expressions), and they can be called before they are defined.

To distinguish between subroutines written in BASIC, subroutines written in assembly language, and subroutines that are the object of **GOSUB** statements, the word subprogram is used in this chapter to define a segment of code delimited by **SUB** and **END SUB** statements.

6.1 Declaring Subprograms

Subprograms are delimited by **SUB** and **END SUB** statements. The **SUB** statement has the following syntax:

SUB *subprogram-name* [[(*parameterlist*)]] **STATIC**

The **SUB** statement takes the following arguments:

Argument	Description
subprogram-name	Is a name up to 31 characters long. The name cannot appear in any other **SUB** statement used in the program.
parameterlist	Contains variables and arrays that represent corresponding variables and arrays passed from the main program. Parameters are separated by commas.
STATIC	Indicates that the subprogram is nonrecursive; that is, it does not call itself, nor does it call another subprogram that calls this subprogram. Only nonrecursive subprograms are currently supported, and an error occurs if **STATIC** is omitted.

The variables declared in the parameter list must correspond in type and number to the variables passed from the main program. Arrays are declared in the parameter list in the following form:

arrayname (*dim-num*)

Argument	Description
arrayname	Is the name of the array.
dim-num	Is an integer representing the number of dimensions in the array. This argument tells the subroutine the size of the array being passed. One dimension is the default.

All QuickBASIC expressions are allowed within a subprogram, except the following:

- User-defined functions and their definition statements (**DEF FN**).
- A **SUB...END SUB** block. Subprogram definitions cannot be nested. However, a subprogram can call another subprogram.

Example

The following subprogram multiplies the values passed to it in the array VALUES. The index value of the array is passed to the subprogram in the variable S.

```
SUB MULT(VALUES(1),S) STATIC
    FOR INDEX=1 TO S
        PRINT VALUES(INDEX)*16
    NEXT INDEX
END SUB
```

6.2 Calling Subprograms

Execution is transferred from the main program to the subprogram with the **CALL** statement. The **CALL** statement can invoke QuickBASIC subprograms or assembly-language subroutines. This section and the rest of this chapter apply only to **CALL** when it is used with subprograms. Calling assembly-language subroutines is described in Chapter 8, "Interfacing with Assembly Language."

The **CALL** statement has the following syntax:

CALL *name* [[(*argumentlist*)]]

Argument	Description
name	Is the name of the called subprogram. The name is limited to 31 characters.
argumentlist	Is a list of variables, arrays, and array elements passed to the called subprogram.

The number and type of the arguments in the **CALL** statement must match the number and type of parameters in the **SUB** statement, or stack space will not be allocated correctly. Arguments and parameters must be given in the same order. The compiler does not check to see if **CALL** statement arguments and **SUB** statement parameters match. No error message is generated if they do not match.

The following example illustrates the form of a complete subprogram call:

```
(main program)
.
.
.
CALL subprogram-name(argumentlist)
.
.
.
(end of main program)

SUB subprogram-name(parameterlist) STATIC

    (body of subprogram)

END SUB
```

6.3 Passing Variables with CALL

Variables can be passed from the main program to subprograms through the **CALL** statement argument list. The following example shows how a **CALL** statement invokes a subprogram and passes variables A and B to it:

```
A=5 : B=0
CALL SQUARE(A,B)
PRINT "Main program variable values are " A "and " B
END

SUB SQUARE(X,Y) STATIC
    Y=X*X
    PRINT "In subprogram, Y =" Y
END SUB
```

The above example prints the following results:

```
In subprogram, Y = 25
Main program variable values are 5 and 25
```

The subprogram changes the value of variable B by assigning a new value to the **SUB** statement's corresponding Y parameter. When the subprogram SQUARE begins executing, the initial value of Y is 0. The subprogram computes a new value for Y (in this case 25) and stores it in the passed parameter.

Variables passed using the above method are passed *by reference*; i.e., their location in memory is passed to the subprogram. The subprogram acts on the value in that location, and when control returns to the calling program, the value in that location may be changed. When you don't want the subprogram to change the value of a variable, you must pass the actual value of the variable. When an actual value is passed, the subprogram copies the value into a temporary location for its own use, and removes the temporary location before control is returned to the calling program. The original value is unchanged.

To pass a variable by value, enclose it in parentheses. For example, if you change the CALL SQUARE statement in the above example to

```
CALL SQUARE(A,(B))
```

B is passed by value, which is 0. In the example, subprogram SQUARE cannot change the value of B. The subprogram can change the value of Y, but this value is local to the subprogram and cannot be passed back to the main program through the B variable. The revised example prints:

```
In subprogram Y=25
Main program variable values are 5 and 0
```

Note that you can also pass expressions as parameters, as in the following:

```
CALL PROG2(X,(Y+1)) STATIC
```

Expressions are passed by value.

6.4 Passing Arrays with CALL

Arrays can be passed to subprograms with the **CALL** statement. The **CALL** statement with an array argument has the following form:

CALL *subprogram* (*arrayname*( ))

Argument	Description
subprogram	Is the name of the subprogram being called
arrayname	Is the name of the array being passed

Arrays use the same type-matching rules as variables.

Example

The following example creates an array, then passes the array to a subprogram that multiplies each array element by 16:

```
DEFINT I-L
OPTION BASE 1
LIMIT=4
DIM ARRAY(LIMIT)
FOR INDEX=1 TO LIMIT
   ARRAY(INDEX)=INDEX
NEXT INDEX
CALL MULT(ARRAY(),LIMIT)
END

SUB MULT(VALUES(1),SUBSCRIPT) STATIC
   FOR ELEMENT=1 TO SUBSCRIPT
       PRINT VALUES(ELEMENT) "times 16 is " VALUES(ELEMENT)*16
   NEXT ELEMENT
END SUB
```

Passing Individual Array Elements

If a subprogram does not require an entire array, you can pass the values of individual array elements instead. To pass separate elements of an array, enclose their subscripts in parentheses. The statement in the following example passes the value of the element in row 3, column 9 of ARRAY1 to the PROGNAME subprogram:

```
CALL PROGNAME(ARRAY1(3,9))
```

Using Array-Bound Functions

The **LBOUND** and **UBOUND** functions, which are described in Part 2, "Reference," are useful for determining the size of an array passed to a subprogram. The **LBOUND** function finds the smallest index value of an array subscript; **UBOUND** finds the largest index value. They are especially useful for passing dynamic arrays, because the calling program does not have to pass the size of each dimension to the subprogram.

For instance, the subprogram in the following example uses **LBOUND** and **UBOUND** instead of passing the upper bounds of the array explicitly. The **LBOUND** function initializes the variable ROW to the lowest subscript value of ARRAY, and the **UBOUND** function limits the number of times the FOR loop executes to the number of elements in the array.

```
SUB PROG2(ARRAY(2)) STATIC
    FOR ROW=LBOUND(ARRAY,2) TO UBOUND(ARRAY,2)
.
.
.
```

The following example uses **LBOUND** and **UBOUND** to compute the limits of the array, instead of a variable value as in the previous example:

```
OPTION BASE 1
DIM ARRAY(4)
FOR INDEX=1 TO UBOUND(ARRAY)
   ARRAY(INDEX)=INDEX
NEXT INDEX
CALL MULT(ARRAY())
END

SUB MULT(VALUES(1)) STATIC
   FOR INDEX=LBOUND(VALUES) TO UBOUND(VALUES)
       PRINT VALUES(INDEX) "times 16 is " VALUES(INDEX)*16
   NEXT INDEX
END SUB
```

6.5 Accessing Parameters with SHARED

In addition to passing variables via arguments to **CALL** and **SUB**, QuickBASIC provides two other ways for variable values to be accessed across subprogram calls:

1. The **SHARED** attribute to main program **COMMON**, **DIM**, and **REDIM** statements permits use of the specified main program variables by all subprograms. This method is most useful if all variables will be shared by all subprograms.
2. The **SHARED** statement permits a subprogram to use variables declared without the **SHARED** attribute in the main program. This method is most useful when different subprograms use different combinations of main program variables.

Sections 6.5.1, 6.5.2, and 6.5.3 discuss these two ways to pass parameters.

6.5.1 Sharing from the Main Program: the SHARED Attribute

Subprograms can use all main program variables declared in **COMMON**, **DIM**, and **REDIM** statements with the **SHARED** attribute. If all the variables in the main program are also used by all the subprograms, this is the easiest and most efficient way to share variables. To use the **SHARED** attribute, place the word **SHARED** directly after the keyword in a **COMMON**, **DIM**, or **REDIM** statement.

Note

The **SHARED** attribute and the **SHARED** statement, described in Sections 6.5.1, 6.5.2, and 6.5.3, only share variables within a single compiled module. They do not share variables with subprograms in the user library, or with subprograms compiled separately and run outside of the QuickBASIC integrated environment.

Example

The following statements declare variables shared across all subprograms:

```
COMMON SHARED A,B,C
COMMON SHARED /LENGTH/ SHORT, MEDIUM, LONG
SHARED X,Y
DIM SHARED ARRAY(10,10)
REDIM SHARED ALPHA(N%)
```

In the next example, the main program passes the value of A to the PROG2 subprogram. When the executable program is run, the number 10 is printed.

```
' MAIN PROGRAM
COMMON SHARED A 'Variable A can be used by any
A=6+4           'subprogram
CALL PROG2

' SUBPROGRAM
SUB PROG2 STATIC
    PRINT A     'prints 10
END SUB
```

Without the **COMMON SHARED** statement, the above program would look like this:

```
'MAIN PROGRAM
A=6+4
CALL PROG2 (A)

' SUBPROGRAM
SUB PROG2(X) STATIC
    PRINT X
END SUB
```

The significance of the **SHARED** attribute becomes apparent when large numbers of variables are being passed from the main program.

6.5.2 Overriding a SHARED Variable with a STATIC Statement

Variables declared in the main program with the **SHARED** attribute are global. Variables declared in subprograms are local to the subprogram. In a subprogam you may occasionally need to use a local variable name that has been previously declared a global variable. You can use the **STATIC** statement to override global variables at the subprogram level.

Note that when you use the **STATIC** statement with an array variable, you must immediately dimension the array, as in following example:

```
SUB PROG2 (X) STATIC
    STATIC ARRAY%(1)
    DIM ARRAY%(25)
.
.
.
END SUB
```

Example

The output of the following program illustrates the effect of the **STATIC** statement on a global variable:

```
FOR I = 1 TO 10
  X=X+2          'x is a global variable
  CALL PROG2
  PRINT "main program X=" X
NEXT I

SUB PROG2 STATIC
  STATIC X       'declares x local to this subprogram
  PRINT "subprogram X=" X
END SUB
```

The output of the program is:

```
subprogram X = 0
main program X = 2
subprogram X = 0
main program X = 4
subprogram X = 0
.
.
.
```

6.5.3 Sharing from the Subprogram: the SHARED Statement

In a subprogram, declaring variables with the **SHARED** statement allows the subprogram to use main program variables that do not have the **SHARED** attribute. The **SHARED** statement has the form

SHARED *variable* [[,*variable*. . .]]

The argument *variable* is the name of any variable declared in the main program. A **SHARED** statement, with array arguments, has the form:

SHARED *arrayname*( ) [[,*arrayname*( )]]

Variables and arrays can both be used as arguments to a single **SHARED** statement.

Example

The following example illustrates the use of two variables (A and D) already declared in the main program. The subprogram can declare them in a **SHARED** statement before they are used:

```
A=4:D=6
CALL PROG2

SUB PROG2 STATIC
    SHARED A,D
    X=A+D
    PRINT X      'prints 10
END SUB
```

6.6 Subprograms in User Libraries

You can place often-used subprograms in the user library **USERLIB.EXE**, where they can be easily linked to any program processed with QuickBASIC. The following sections apply to calling subprograms in libraries.

6.6.1 Naming Restrictions

Subprograms compiled with the the Obj(BCOM.LIB) or Obj(BRUN.LIB) options must have unique names. If a subprogram in the main program has the same name as a subprogram in the user library, a "Duplicate definition error in user library" occurs.

6.6.2 Passing Parameters

If the main program uses subprograms that are stored in the user library, you can pass parameters from the main program to the subprogram. Parameters can be passed either as arguments to the **CALL** and **SUB** statements, or with the **COMMON** statement. The **COMMON** statement is described in Part 2, "Reference."

COMMON [[**SHARED**]] */blockname/ itemlist*

The **COMMON** statement is useful when there are many variables to be shared, or when not all variables are shared by all subprograms.

6.6.3 Using Named COMMON

Named **COMMON** separates shared variables into groups that are identified by their *blockname*. It is useful when subprograms need to share specific groups of variables from the main program. See Part 2, "Reference," for an example of using named **COMMON**.

6.7 Common Errors

Several errors are commonly made when calling subprograms:

- Mismatched argument and parameter lists
- Variable aliasing
- Entering and exiting subprograms improperly
- Misplaced error-handling routines

Sections 6.7.1–6.7.4 describe these errors and how to correct them.

6.7.1 Argument and Parameter List Errors

Mismatched argument and parameter list errors are caused when the order, type, or number of arguments passed to a BASIC subprogram do not correspond exactly to the parameters in the subprogram. The QuickBASIC compiler does not check for this discrepancy, and no error message is generated. However, noticeable side effects will probably occur.

Example

In the following program, the main program passes a string to a subprogram that is expecting an integer. Although the program compiles and links without errors, when the program is executed, it will eventually cause a "String space corrupt" error.

```
A$="This is a string"
CALL PROG2(A$)

SUB PROG2(X%) STATIC
    X%=X%*X%
END SUB
```

6.7.2 Variable Aliasing

A common mistake in long programs containing many variables is variable aliasing. Variable aliasing occurs when more than one name refers to the same location in memory; for example, when an argument passed to a subprogram can be referenced in the subprogram in more than one way. This can happen when the same variable is used twice as a parameter to a **CALL** statement, or when a variable passed as a parameter is also accessed by means of the **SHARED** statement or the **SHARED** attribute.

To avoid aliasing problems, pass arguments by value, make minimal use of **SHARED** statements and attributes, and use the **STATIC** statement to ensure variable values in subprograms.

Example

The following example illustrates how unexpected variable aliasing can occur.

```
COMMON SHARED A
A=4
CALL PROG2 (A)

SUB PROG2 (X) STATIC
    PRINT "Half of"; X; "plus half of"; A; "is";
    X=X/2
    A=A/2
    PRINT (A+X)
END SUB
```

When run, the preceding program displays the following message:

```
Half of 4 plus half of 4 is 2
```

In the above example, A (passed in the COMMON SHARED statement) is the variable the subprogram references from the main program, and it is considered global data. When the subprogram modifies X, it is in effect modifying A, so when further operations are performed on A with the assumption A=4, the results are false.

6.7.3 Using GOSUB...RETURN and GOTO Statements in Subprograms

Subprograms are intended to be entirely independent of the main program. To preserve this independence, it is recommended that subprograms be entered only with the **CALL** statement, and exited only with the **END SUB** statement. This ensures that the stack is maintained properly, and prevents unexpected errors at run time. Entering and exiting subprograms with **GOTO**, **GOSUB**, or **RETURN** *linenumber* statements is not recommended. Within a subprogram, the objects of **GOTO** and **GOSUB** statements should be located inside the subprogram.

Note

Note that the objects of **ON ERROR** and **ON** *event* **GOSUB** statements must be located outside the subprogram and within the same compiled module, as stated in Section 6.7.4.

Improperly placed **GOTO**, **GOSUB**, or **RETURN** *linenumber* statements will cause a "Subroutine error" message at run time.

6.7.4 Error- and Event-Handling Routines

Error- and event-handling routines must be located in the main program, outside the subprogram but within the same compiled module. Error- and event-handling routines, located inside **SUB...END SUB** blocks, will cause "Subprogram error" messages, and may not return correctly. The recommended structure for error handling with a subprogram is:

```
        SUB TEST STATIC
        .
        .
        .
        ON ERROR GOTO 5000
        .
        .
        .
        END SUB

5000:   'error handling routine begins here

          (body of routine)

        RESUME
```

Note

Error- and event-handling routines are local to each compiled module. Error and event handlers cannot be used by subprograms in the user library, or by other programs compiled separately and run outside the QuickBASIC integrated environment.

Chapter 7
Creating User Libraries

7.1 Building a User Library 133
7.2 Specifying a User Library 134
7.3 QuickBASIC User Library Routines 135

The **buildlib** program allows you to create libraries of routines to be used by QuickBASIC programs. You can use **buildlib** to create libraries of programs written in BASIC or assembly language.

To create a library

- Compile the routines that will compose the library using the Obj(BRUN20.LIB) option from the Compile... command in the Run menu. The user library is created from object files.
- Use the **buildlib** command to create the library, which by default is called **USERLIB.EXE**. The use of **buildlib** is described below.

See Section 3.10, "Creating a User Library," for a hands-on practice session. The practice session starts with BASIC source files and takes you step-by-step through the process of compiling, building, and loading a user library.

7.1 Building a User Library

The **buildlib** command links the given object files and creates the library file. It has the following syntax:

buildlib *files*[,*libraryname*];

Argument	Description
files	Are the object files of routines that will be in the library.
libraryname	Is the name of the resulting library. If *libraryname* is omitted, the library is named **USERLIB.EXE** and placed in the current directory.

To add, delete, or rename a routine in the library, you must run **buildlib** again with the entire list of object files.

Example

The following command links the files **GRAPH.OBJ**, **CIRCLE.OBJ**, and **CHART.OBJ** to create a user library **USERLIB.EXE**:

```
buildlib graph.obj circle.obj chart.obj ;
```

7.2 Specifying a User Library

The **qb** command's /l option loads the **USERLIB.EXE** default user library. QuickBASIC loads the user library when it is invoked and automatically links the routines to the compiled program. If your library has a name other than **USERLIB.EXE**, you will need to give the /l option a file-name argument. The /l option with a file-name argument has the following form:

/l *libraryfile*

User library file names must have the extension **.EXE**. Only one user library can be loaded at a time. QuickBASIC searches for the user library in the following locations:

1. The place specified in the command line argument, if any.
2. The current directory.
3. The path specified by the **lib** environment variable. (See your DOS documentation for information about environment variables.)

Example

The following command uses the routines in the library **FIGS.EXE** when **REPORT.BAS** is processed:

```
qb report.bas /l figs.exe
```

7.3 QuickBASIC User Library Routines

QuickBASIC supplies an unbuilt user library, **USERLIB.OBJ**, that contains three assembly-language subroutines, **INT86**, **INT86X**, and **PTR86**. These routines provide software interrupt support for system service calls.

To compile and run QuickBASIC programs that use the interrupt support routines, you must load a user library that contains them when you start QuickBASIC. This section describes how to place the interrupt routines in the user library. Using **INT86**, **INT86X**, and **PTR86** is described in Chapter 8, "Interfacing With Assembly Language."

The file **USERLIB.OBJ** is on QuickBASIC Disk 2. The following example creates such a library:

```
buildlib userlib.obj;
```

The above command line produces the file **USERLIB.EXE**. The interrupt support routines can be located in any user library as long as the library is loaded when QuickBASIC is started.

Chapter 8

Interfacing with Assembly Language

8.1 Calling Assembly-Language Subroutines with CALL and CALLS 139

8.2 Coding Subroutines for the Compiler 143

8.3 Passing Arrays to Assembly-Language Subroutines 147

8.4 Calling System Services: Interrupt Support Routines 147

8.4.1 INT86 and INT86X 147

8.4.2 PTR86 150

8.5 The Run-Time Memory Map 152

This chapter explains how to call assembly-language subroutines from Microsoft QuickBASIC programs, and the rules for coding assembly-language subroutines called by compiled programs. It also describes the calling interface used by the interrupt routines **INT86**, **INT86X**, and **PTR86**.

This chapter is intended for experienced programmers who are familiar with BASIC and 8086 assembly language. If you need help learning to program in assembly language, or you need more information about DOS, see Section 1.6, "Resources for Learning."

Microsoft QuickBASIC interfaces with assembly-language subroutines through the **CALL** statement. To use assembly-language subroutines with QuickBASIC, call them with the correct number and type of parameters, then place the assembled object file in the user library. (If you are using the separate compilation method described in Appendix D, link the assembled object file with the BASIC object file.) Since the operating system decides where in memory the assembly-language program should go, this virtually eliminates memory-allocation problems.

Note

When using assembly-language subroutines, it is advisable, but not mandatory, to include **PREFIX.OBJ** in the user library or at link time. The **PREFIX.OBJ** file predeclares all segments and classes in the correct order for use with QuickBASIC.

8.1 Calling Assembly-Language Subroutines with CALL and CALLS

The **CALL** statement calls an assembly-language subroutine and passes the unsegmented (16-bit offset) addresses of the statement arguments to the routine. The **CALLS** statement has the same syntax as the **CALL** statement, but passes the segmented (32-bit segment and offset) addresses of its arguments. See Part 2, "Reference," for a description of the **CALL** statement syntax.

Invoking the **CALL** statement causes the following to occur:

1. Arrays are passed by pushing an array descriptor on the stack.

2. Array elements are passed by pushing a 2-byte offset to the value relative to the data segment (**DS**) onto the stack. Dynamic arrays are allocated **FAR** (outside **DS**). The specified array element is first copied to a temporary **NEAR** location (in **DS**) and the offset of the temporary location is passed. Note that the results of expressions are also copied to temporary locations.

   Because dynamic arrays are passed in temporaries, the offset supplied will *not* point into the array itself, and adjacent elements are not addressable. To get the address of the dynamic array data, use **VARPTR** to obtain the address of the first array element, then use the **PTR86** routine to obtain the segment and offset values. See Section 8.4.2 for more information about **PTR86**.

3. BASIC pushes the 2-byte offset of each nonarray argument's location in **DS** onto the stack.

4. The BASIC return address code-segment (**CS**) and offset (**IP**) are pushed onto the stack.

5. Control is transferred to the assembly-language subroutine with a **FAR** call. The routine must also execute a **FAR** return.

Figure 8.1 shows what the stack looks like when the **CALL** statement is executed.

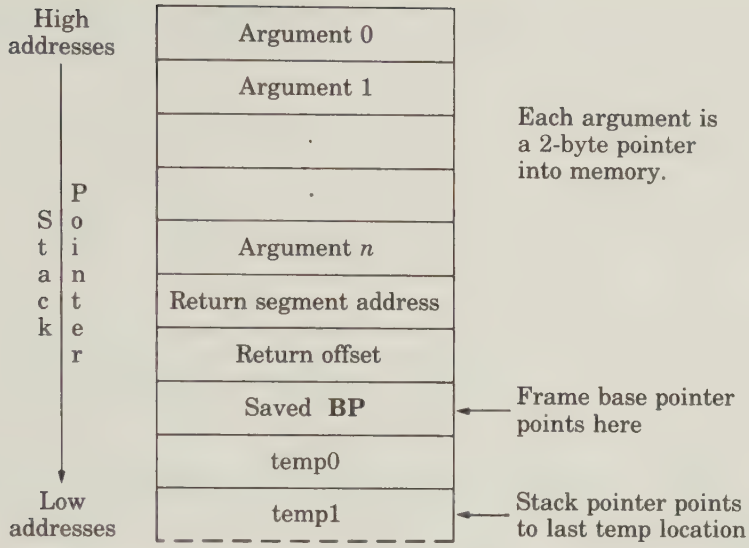


Figure 8.1 The Stack after CALL Executes

The assembly-language subroutine now has control. Arguments are referenced relative to the stack. To reference an argument, establish a base-pointer value in `bp` by saving `bp`, then set `bp` to the value of `sp`, as illustrated in the following example:

```
push   bp
mov    bp,sp
.
.
.
```

You can calculate the offset of argument n, the last argument, as **BP** + 6. Argument n–1 is at **BP** + 8. You can locate any argument, for example, k, by using the following formula, where k is the argument and n is the total number of arguments made by the program:

Location k = **BP** + 6 + (2*(n–k))

The following formula gives you the location of argument 0:

Location of argument 0 = **BP** + 6 + 2n

If you have local variables (variables that use the stack instead of the data segment), begin the assembly-language program with code similar to the following:

```
push    bp
mov     bp,sp
sub     sp, space
.
.
.
mov     sp, bp
pop     bp
ret
```

The *space* argument is equal to the amount of space used by local variables in the program.

If all the local variables are 2 bytes long, you can calculate the location of local variable 0 with the following formula:

Location of local variable 0 = **BP** – 2

If all the local variables are the same 2-byte length, you can calculate the locations of subsequent local variables. For each variable, subtract an additional 2 bytes from **BP**. Use the following formula to calculate the location of the *k* local variable:

Location of local variable = **BP** – 2(*k* + 1)

Important

Your program must clean up the stack before exiting the called subroutine. Do this by popping **BP**, as in the following example:

```
push  bp
mov   bp, sp
sub   sp, space
.
.
.
mov   sp,bp
pop   bp
ret
```

8.2 Coding Subroutines for the Compiler

You must observe the following rules when coding an assembly-language subroutine for the Microsoft QuickBASIC Compiler:

1. On exit, the called routine must restore the contents of the **BP**, **SS**, and **DS** registers to their initial values.

2. All data declared and referenced in the routine should be in the segment **DATA**, and in the group **DGROUP**. First create **DGROUP** with the **GROUP** directive, then tell the assembler that **DGROUP** is addressed off the **DS** register with the **ASSUME DS:DGROUP** assembler directive.

3. The called routine must know the number and type of the arguments passed. References to arguments are positive offsets added to **BP** (assuming the called routine moved the current stack pointer into **BP**).

4. The called routine must do a far **RET** *n* to adjust the stack to the start of the calling sequence. Upon exit, the following registers may be modified from their entry values: **BX**, **CX**, **DX**, **SI**, **DI**, and **ES**.

   In programs called by the **CALL** statement, *n* is two times the number of parameters in the argument list. In programs called by the **CALLS** statement, *n* is four times the number of parameters in the argument list.

5. To return values to BASIC, include the variable name that will receive the result in the argument list.

6. The routine *must not* change the length of any strings. Microsoft QuickBASIC cannot correctly manipulate strings if their lengths are modified by external routines.

If an argument is a string, the argument's offset points to 4 bytes called the "string descriptor." Bytes 0 and 1 of the string descriptor contain the length of the string (0 to 32,767). Bytes 2 and 3, respectively, are the lower and upper 8 bits of the string's starting address in string space. Changing bytes past the length of the string, or modifying the string descriptor, will cause unpredictable results, probably a "String space corrupt" error message.

Use the **SADD** and **LEN** functions to access bytes 2 and 3 directly. The **LEN** function returns the length of the string, and the **SADD** function returns the location of the start of the string within the internal string heap used by QuickBASIC.

See Part 2, "Reference," for information about these functions. See the example in Section 8.4.1, "INT86 and INT86X."

Examples

Passing an Integer

The following program demonstrates how to call an assembly-language subroutine, and how to pass an integer argument. The program, when run, asks for input in the form of an integer, adds 16 to it, then prints the result on the screen.

```
INPUT "Enter an integer: ",A%
CALL INCR (A%)
PRINT A%
```

The following subroutine, `incr`, adds `16` to the integer it is given:

```
data     segment  word     public   'data'

storage dw        16

data     ends

dgroup   group    data

code     segment  byte     public   'code'
         assume   cs:code, ds:dgroup

         public   incr
incr     proc     far           ;routine is a FAR procedure
         push     bp            ;save BP
         mov      bp,sp         ;set up to address of BP
         mov      bx,[bp]+6     ;fetch address of parm 0
         mov      ax,storage
         add      [bx],ax       ;add value to parm 0
         pop      bp            ;restore BP
         ret      2             ;return and remove 1 parm from stack
incr     endp

code     ends

         end
```

Passing a Single-Precision Number

The following example demonstrates how to pass a single-precision argument to an assembly-language subroutine. The following statement calls the subroutine:

```
CALL ENTRY(A!)
```

The assembly-language subroutine can appear as follows:

```
entry    proc    far
         push    bp
         mov     bp,sp

         mov     bx,6[bp]        ;get address of A!
         mov     ax,[bx]
         mov     dx,[bx+2]       ;single precision is 4 bytes
         .
         .
         .
         pop     bp
         ret     2
```

Passing a Double-Precision Number

The following example demonstrates how to pass a double-precision argument to an assembly-language subroutine. The following statement calls the subroutine:

```
CALL ENTRY(A#)
```

The assembly-language subroutine can appear as follows:

```
entry    proc    far
         push    bp,sp
         mov     bp,sp

         mov     bx,6[bp]
         mov     ax,[bx]
         mov     dx,[bx+2]
         mov     si,[bx+4]
         mov     di,[bx+6]
         .
         .
         .
         pop     bp
         ret     2
```

Passing a String

The following example demonstrates how to access string arguments, and pass and store a return result in a variable. Using the string descriptor, the following statement calls the subroutine:

```
CALL ENTRY (B$)
```

The assembly-language subroutine can appear as follows:

```
entry proc  far
      push  bp
      mov   bp,sp          ;get current stack position bp
      mov   bx,6[bp]       ;get B$ descriptor address
      mov   cx,[bx]        ;get length of B$ in cx
      mov   dx,2[bx]       ;get address of B$ text in dx
.
.
.
      pop   bp
      ret   2              ;restore stack,return
entry endp
```

The following command calls the subroutine using the **SADD** function to access the string data:

```
CALL ENTRY(SADD(B$))
```

The assembly-language subroutine would look like this:

```
entry     proc      far           ;routine is a FAR procedure
          push      bp            ;save BP
          mov       bp,sp         ;set up to address of BP
          mov       bx,[bp]+6     ;fetch address of parm 0
            .
            .
            .
          pop       bp            ;restore BP
          ret       2             ;return and remove 1 parm from stack
entry     endp
```

8.3 Passing Arrays to Assembly-Language Subroutines

Use the **VARPTR** function, not the array descriptor, to pass arrays to assembly-language subroutines, then use the **PTR86** routine to compute the segment-offset value of the first array element. See Section 8.4.2 for an example, and for more information about **PTR86**.

8.4 Calling System Services: Interrupt Support Routines

The interrupt routines **INT86**, **INT86X**, and **PTR86** provide support for calling DOS, BIOS, or other system services. Before QuickBASIC programs that make system service calls can be processed, a user library containing the routines must be loaded into memory. (User libraries are loaded when QuickBASIC is invoked. Creating such a library is described in Chapter 7, "Creating User Libraries.") This section describes the **INT86**, **INT86X**, and **PTR86** calling interfaces.

8.4.1 INT86 and INT86X

The **INT86** and **INT86X** interrupt routines execute 8086 software interrupts. Before executing the interrupt, the routines set the registers with values specified in an integer array. **INT86** sets the nonsegment registers only; **INT86X** sets all the registers.

Warning

INT86X should be used only when the system call *requires* segment register values. Altering segment registers incorrectly can cause serious problems.

The **INT86** and **INT86X** calling interface has the following syntax:

CALL INT86[X] (*int_ no*,**VARPTR**(*in_ array*(*x*)),**VARPTR**(*out_ array*(*y*)))

Argument	Description
int_ no	Is the interrupt being executed. It is an integer between 0 and 255. See your DOS documentation for the interrupt routine numbers.
in_ array(*x*)	Is an integer array that specifies the register values at interrupt time. The array index, *x*, is an integer or an integer expression. **INT86** uses an 8-element array; **INT86X** uses a 10-element array. The last two array elements in **INT86X** represent the segment register values. Table 8.1 lists the array input values and their corresponding registers.

Table 8.1

INT86 and INT86X Register Values

Array Element	Register
in_ array(*x*)	**AX**
in_ array(*x*+1)	**BX**
in_ array(*x*+2)	**CX**
in_ array(*x*+3)	**DX**
in_ array(*x*+4)	**BP**
in_ array(*x*+5)	**SI**
in_ array(*x*+6)	**DI**
in_ array(*x*+7)	**FLAGS**
in_ array(*x*+8)*	**DS**
in_ array(*x*+9)*	**ES**

*These array elements are used only by INT86X. To use the current run-time values of DS and ES, assign the value –1 to array elements 8 and 9.

out_ array(*y*)	Contains the post-interrupt register values. It has the same structure as *in_ array*.

If an error occurs, *int_no* = -1 and *out_array* are unchanged. Errors are caused when *int_no* is not in the range 0–255.

The **INT86** and **INT86X** routines alter all registers except **BP** and **DS**.

Example

The following example uses **INT86** to open a file and place some text in it:

```
DIM INARY%(7),OUTARY%(7)          'define input and output arrays
                                  'for INT86
'
AXREG% = 0                        'define register array indices to
BXREG% = 1                        'make program easier to understand
CXREG% = 2
DXREG% = 3
BPREG% = 4
SIREG% = 5
DIREG% = 6
FLREG% = 7
'
INARY%(AXREG%) = &H3C00           'DOS function to create a file
INARY%(CXREG%) = 0                'DOS attribute for created file
INARY%(DXREG%) = SADD("FOO.TXT"+CHR$(0))
                                  'pointer to filename string
                                  'with zero byte termination

CALL INT86(&H21,VARPTR(INARY%(0)),VARPTR(OUTARY%(0)))
                                  'perform the creation
'
INARY%(BXREG%) = OUTARY%(AXREG%)   'move created file handle for write
INARY%(AXREG%) = &H4000           'DOS function to write to file
TEXT$ = "hello, world"+CHR$(13)+CHR$(10)
                                  'define text to write to file

INARY%(CXREG%) = LEN(TEXT$)       'get length of text string
INARY%(DXREG%) = SADD(TEXT$)      'get address of text string
CALL INT86(&H21,VARPTR(INARY%(0)),VARPTR(OUTARY%(0)))
                                  'perform the write
'
INARY%(AXREG%) = &H3E00           'DOS function to close a file
CALL INT86(&H21,VARPTR(INARY%(0)),VARPTR(OUTARY%(0)))
                                  'perform the close
```

8.4.2 PTR86

The **PTR86** interrupt routine converts floating-point values returned by the **VARPTR** function to segment-offset values. It is used to convert the address of a large numeric array variable to a segment-offset value for use with **INT86X**.

The **PTR86** calling interface has the following syntax:

CALL PTR86 (*varseg*,*varoff*,**VARPTR**(*var*))

Argument	Description
varseg	Is the segment address returned by **PTR86**
varoff	Is the offset address returned by **PTR86**
var	Is the variable whose address is being computed

If an error occurs, **PTR86** returns a value of –1 in *varseg*.

Example

The following example opens a file and places characters typed at the keyboard into it:

```
DIM INARY%(9),OUTARY%(9)        'define INT86 input and output arrays
BUFSIZ% = 2000                  'size of buffer for output
DIM BUF%((BUFSIZ%-1)/2)         'define integer LNA to serve as the buffer
'
AXREG% = 0                      'variables to make array references
BXREG% = 1                      'easier to understand
CXREG% = 2
DXREG% = 3
BPREG% = 4
SIREG% = 5
DIREG% = 6
FGREG% = 7
DSREG% = 8
ESREG% = 9
'
INARY%(AXREG%) = &H3D00         'open file for reading only
INARY%(DXREG%) = SADD("READ.BAS"+CHR$(0))
                                'get offset of filename string

CALL INT86(&H21,VARPTR(INARY%(0)),VARPTR(OUTARY%(0)))
                                'perform open
'
INARY%(BXREG%) = OUTARY%(AXREG%)
                               'move file handle for read operation
INARY%(AXREG%) = &H3F00        'operation to read file into buffer
INARY%(CXREG%) = BUFSIZ%       'size of buffer to read into
CALL PTR86(INARY%(DSREG%),INARY%(DXREG%),VARPTR(BUF%(0)))
                                'buffer at DS:DX

CALL INT86X(&H21,VARPTR(INARY%(0)),VARPTR(OUTARY%(0)))
                                'perform read

BUFSIZ% = OUTARY%(AXREG%)       'get actual bytes read
'
INARY%(0) = &H3E00              'operation to close file
CALL INT86(&H21,VARPTR(INARY%(0)),VARPTR(OUTARY%(0)))
                                'perform close
'
ADR! = VARPTR(BUF%(0))          'pointer to start of buffer
INARY%(AXREG%) = &H600          'operation for direct console I/O
FOR IND% = 1 TO BUFSIZ%         'for each character in buffer...
    INARY%(DXREG%) = PEEK(ADR!-1+IND%)
                                'get value of IND%'th character
    CALL INT86(&H21,VARPTR(INARY%(0)),VARPTR(OUTARY%(0)))
                                'perform direct console input

NEXT IND%                       'repeat until done with buffer
```

8.5 The Run-Time Memory Map

Figure 8.2 shows the memory map during in-memory program execution.

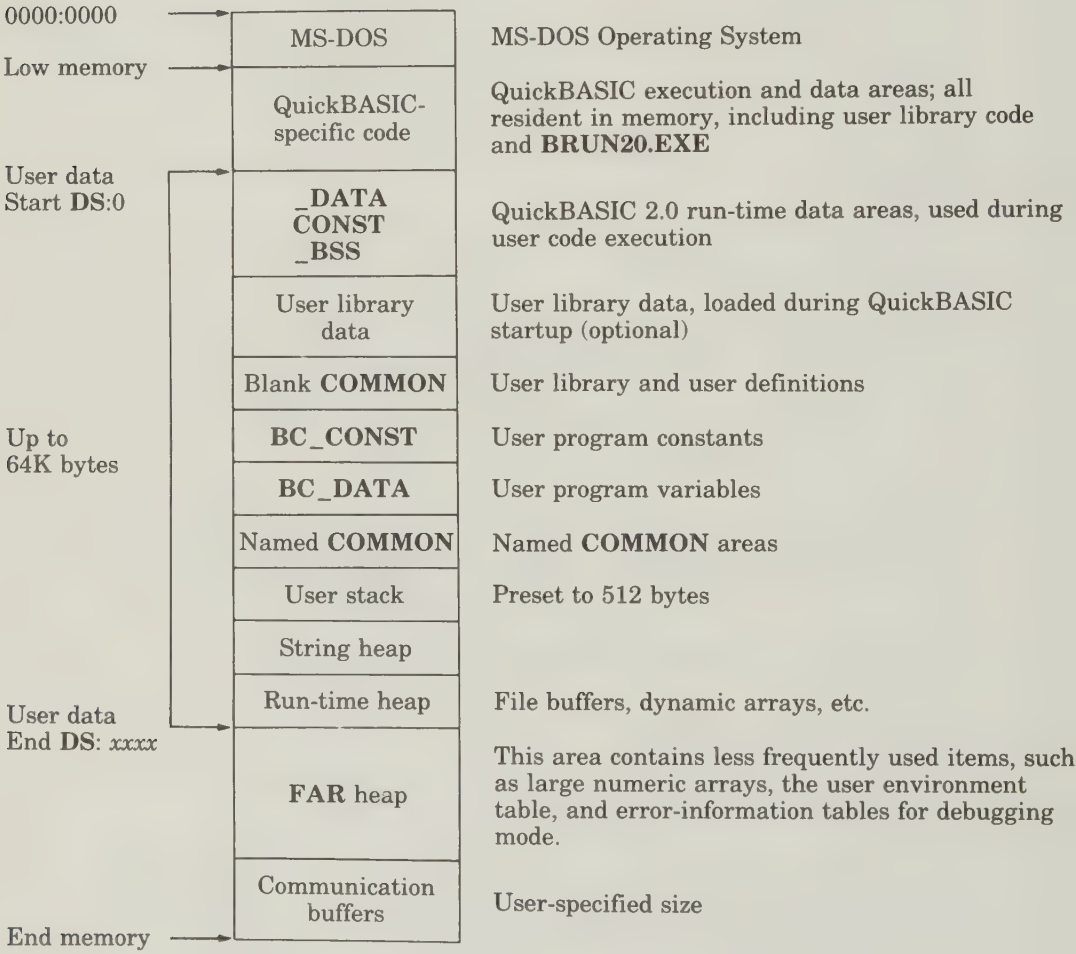


Figure 8.2 Memory Map during Program Execution

Part 2
Reference

9 Language Elements 155

10 Compiler-Interpreter Language Differences 181

11 Statement and Function Reference 195

Chapter 9
Language Elements

9.1 Character Set 157
9.2 The QuickBASIC Program Line 158
9.2.1 Using Line Labels 160
9.2.2 Creating Lines Longer than 255 Characters 161
9.3 Data Types 161
9.4 Constants 162
9.5 Variables 165
9.5.1 Variable Names 166
9.5.2 Declaring Variable Types 166
9.5.3 Array Variables 168
9.6 Expressions and Operators 169
9.6.1 Hierarchy of Operations 169
9.6.2 Arithmetic Operators 170
9.6.2.1 Integer Division 171
9.6.2.2 Modulo Arithmetic 172
9.6.2.3 Overflow and Division by Zero 172
9.6.3 Relational Operators 172
9.6.4 Logical Operators 173
9.6.5 Functional Operators 176
9.6.5.1 Intrinsic Functions 176
9.6.5.2 User-Defined Functions 177
9.6.6 String Operators 177
9.7 Type Conversion 178

This chapter presents the character set and the rules for the constants, variables, expressions, and operators used by the Microsoft QuickBASIC language.

9.1 Character Set

The Microsoft QuickBASIC character set consists of alphabetic characters, numeric characters, and special characters.

The alphabetic characters in BASIC are the uppercase letters (A–Z) and lowercase letters (a–z) of the English alphabet.

The BASIC numeric characters are the digits 0–9. The letters a–f and A–F can be used as parts of hexadecimal numbers. The following special characters are recognized by BASIC:

Character	Name or Function
RETURN	Terminates input of a line
	Blank (or space)
!	Exclamation point (suffix for single-precision data type)
#	Number (or pound) sign (suffix for double-precision data type)
$	Dollar sign (suffix for string data type)
%	Percent (suffix for integer data type)
&	Ampersand
'	Single quotation mark (apostrophe)
(	Left parenthesis
)	Right parenthesis
*	Asterisk or multiplication symbol
+	Plus sign
,	Comma
-	Minus sign
.	Period or decimal point

/	Slash or division symbol
:	Colon
;	Semicolon
<	Less than
=	Equal sign or assignment symbol
>	Greater than
?	Question mark
@	At symbol
[	Left bracket
\	Backslash or integer division symbol
]	Right bracket
^	Up arrow or exponentiation symbol
_	Underscore

9.2 The QuickBASIC Program Line

QuickBASIC program lines have the following syntax:

[[*line-identifier*]] *statement* [[: *statement...*]] [[*comment*]]

Program lines can contain a maximum of 255 characters, and must end with a carriage return. (You can use the underscore character (_) to create "logical" lines longer than 255 characters. Refer to Section 9.2.2 for more information.)

The QuickBASIC Compiler supports two types of *line-identifiers*: line numbers and alphanumeric line labels:

1. A line number can be any combination of digits, from 0 to 65,529. QuickBASIC accepts lines with no space between the line number and the keyword. Spaces are allowed in line numbers, but the compiler uses only the digits preceding the space as the line number. The following are valid line numbers:

```
1
200
300PRINT "hello"     '300 is the line number
65000
25 10                'The compiler considers this line 25
```

Important

Using line number 0 is not recommended. Although line number 0 is allowed, error-trapping statements (such as **ON ERROR GOTO** and **ON** *event* **GOTO**) interpret the presence of line number 0 to mean that error trapping is disabled. For example, the following statement will not branch to line 0 if an error occurs:

```
ON ERROR GOTO 0
```

Note that if **RESUME 0** is present in a program that also contains line number 0, execution will *not* resume on line number 0, but on the line where the error occurred.

---

2. An alphanumeric line label can be any combination of from 1 to 40 letters and digits that ends with a colon. QuickBASIC keywords are not permitted. The following are valid alphanumeric line labels:

   ```
   alpha:
   a16:
   SCREENsub:
   123:
   ```

   Case is not significant in line labels. The following line labels are equivalent:

   ```
   alpha:
   Alpha:
   ALPHA:
   ```

Line numbers and labels can begin in any column, as long as they are the first nonblank characters on the line. Spaces are allowed between an alphanumeric label and the colon following it. There cannot be more than one line label on a line.

A BASIC *statement* can be either executable or nonexecutable. An executable statement tells your program what to do next (such as read input, write output, open a file, or branch to another part of the program). All statements are executable, except the following:

COMMON
DATA
DEF*type*
DIM (static arrays only)
OPTION BASE
REM (includes metacommands)

A *comment* is a nonexecutable statement used to clarify a program's operation and purpose. A comment is introduced by a single quote character ('), or the **REM** statement.

More than one BASIC statement can be placed on a line, but each statement must be separated from the last by a colon (:).

9.2.1 Using Line Labels

QuickBASIC does not require each line in a source program to have a line number or label. You can mix alphanumeric labels and line numbers in the same program, and you can use alphanumeric labels as objects of any BASIC statement where line numbers are permitted, except as the object of an **IF...THEN** statement. In **IF...THEN** statements, BASIC permits only a line number, unless you explicitly use a **GOTO** statement. For example, the following statement will compile and execute correctly:

```
IF A = 10 THEN 500
```

However, if the object of the **IF...THEN** statement is a line label, a **GOTO** statement is required, as follows:

```
IF A = 10 THEN GOTO INCOMEDATA
```

If you are trapping errors, the **ERL** function will return only the last line *number* encountered before the error. Line labels or numbers are not required with the **RESUME** and **RESUME NEXT** statements.

9.2.2 Creating Lines Longer than 255 Characters

You can use the underscore character (_) to create "logical" lines of more than 255 characters. Using this feature, you can improve program structure and readability.

The underscore (_) feature can be used to break lines for improved readability, and is particularly effective with **FIELD** statements. Use of the underscore is shown in the following:

```
FIELD #1,10 AS A$,_    'name
         10 AS B$,_    'address
         10 AS C$,_    'zip code
         10 AS D$_     'license number
```

9.3 Data Types

Four data types exist in BASIC. They are described in detail as follows:

1. String

   A string is a sequence of up to 32,767 characters. The codes for these characters range from 0–127 for ASCII (American Standard Code for Information Interchange) characters, and from 128–255 for non-ASCII characters (see Appendix A, "ASCII Character Codes.").

2. Integer numeric

   Integers are stored internally as signed (two's complement) 16-bit binary numbers ranging in value from –32,768 to 32,767.

3. Single-precision floating-point numeric

   Single-precision numbers are accurate to 7 decimal places, and have approximate ranges of –1.7E+38 to –2.9E–39 for negative values, true zero, and 2.9E–39 to 1.7E+38 for positive values.

   Single-precision numbers are stored internally as 32-bit (or 4-byte) binary numbers in sign-magnitude format. The number is negative if the most significant bit (or MSB) of the third byte is set (that is, if it is equal to 1). The number is positive if the MSB is clear (0).

The unsigned magnitude of the number is contained in 2 parts: a 24-bit binary fraction (the mantissa), and an 8-bit exponent (the power of 2 by which the mantissa must be multiplied to get the single-precision value).

The mantissa is a normalized, unsigned fraction whose binary point is located to the left of its MSB (since this fraction is normalized, the MSB is implied). The remaining 23 bits of the mantissa comprise the seven right-most bits of the third byte, plus the 16 bits of the second and first bytes, respectively. The value of the mantissa ranges from 0.5 (decimal) up to, but not including, 1.

The exponent is the value contained in the last (fourth) byte, minus 256 (80H). This is known as an "excess 256" representation. A zero value is defined as a binary representation whose fourth byte is 0, regardless of the contents of the other 3 bytes.

4. Double-precision floating-point numeric

   Double-precision numbers have the same allowable ranges as single-precision numbers, but are accurate to 15 decimal places. The binary representation of a double-precision number has 58 bits to accommodate this extra accuracy. The preceding discussion of single-precision numbers applies to double-precision numbers, with the following differences:

   The sign bit is the MSB of the seventh byte.

   The mantissa consists of an implied 1, the 7 right-most bits of byte 7, and bytes 6–1, respectively.

   The exponent is the eighth byte. A value of 0 in this byte implies a value of 0 for the entire number.

9.4 Constants

Constants are predefined values that do not change during program execution. There are two types of constants: string and numeric.

A string constant is a sequence of up to 32,767 alphanumeric characters enclosed in double quotation marks. These alphanumeric characters can be any of the characters whose ASCII codes fall within the range 32 to 127, except the double quote character ("). These are all valid string constants:

```
"HELLO"
"$25,000.000"
"Number of Employees"
```

Numeric constants are positive or negative numbers. There are three types of numeric constants: integer, fixed-point, and floating-point. These types of constants are described as follows:

1. Integer constants

   a. Decimal

      One or more decimal digits (0–9), with an optional sign prefix (+ or –). The range for decimal constants is –32,768 to +32,767.

   b. Hexadecimal

      One or more hexadecimal digits (0–9, a–f, or A–F) with the prefix &H or &h. The range for hexadecimal constants is &H0 to &HFFFF.

      Examples:

      ```
      &H76
      &H32F
      ```

   c. Octal

      One or more octal digits (0–7) with the prefix &O, &o, or &. The range for octal constants is &O0 to &O17777.

      Examples:

      ```
      &O347
      &1234
      ```

   Integer constants do not contain decimal points.

2. Fixed-point constants

   Positive or negative real numbers; that is, numbers that contain decimal points.

   Example:

   ```
   9.0846
   ```

3. Floating-point constants

   Positive or negative numbers represented in exponential form (similar to scientific notation). A single-precision floating-point constant consists of an optionally signed integer or fixed-point number (the

mantissa) followed by the letter E and an optionally signed integer (the exponent). The exponent is a power of ten, by which the mantissa is multiplied to get the value of the floating-point number. Double-precision floating-point constants are denoted by the letter D instead of E.

Examples:

```
235.988E-7 = .0000235988
2359E6 = 2359000000
4.35D-10 = .000000000435
```

Fixed-point and floating-point numeric constants can be either single-precision or double-precision numbers. Single-precision numeric constants are stored with 6 digits of precision (plus the exponent) and printed with up to 6 digits of precision. Double-precision numbers are stored with 14 digits of precision (plus the exponent) and printed with up to 14 digits of precision. A single-precision constant is any numeric constant that has one of the following properties:

- Six or fewer digits with no "D" exponential character or trailing "#"
- Exponential form denoted by "E"
- A trailing exclamation point ("!")

A double-precision constant is any numeric constant that has one of the following properties:

- Seven or more digits with no "E" exponential character or trailing "!"
- Exponential form denoted by "D"
- A trailing number sign ("#")

The following are examples of numeric constants:

Single Precision	Double Precision
46.8	345692811
-1.09E-6	-1.09432D-06
3489.0	3489.0#
22.5!	7654321.1234

Numeric constants in BASIC cannot contain commas.

9.5 Variables

Variables are named values that can change during the execution of a program, or from one execution of a program to the next. As with constants, there are two types of variables: numeric and string. A numeric variable can be assigned only a numeric value (either integer, single-precision, or double-precision); a string variable can be assigned only a character-string value. You can assign the following values to a variable:

1. A constant value, as in the following example:

   ```
   A = 4.5
   ```

2. The value of another variable, as in the following example:

   ```
   B$ = "ship of fools"
   A$ = B$
   ```

3. The value obtained by combining other variables and/or constants with operators, as in the following example:

   ```
   PI = 3.141593
   CONVERSION = 180/PI
   ```

   For more information on combining variables and constants, see Section 9.6, "Expressions and Operators."

In any case, the variable must always match the type of data (numeric or string) assigned to it.

Note

Before a variable is assigned a value, its value is assumed to be zero (for numeric variables) or null (for string variables).

9.5.1 Variable Names

A BASIC variable name can contain as many as 40 characters. The characters allowed in a variable name are letters, numbers, the decimal point, and the type-declaration characters %, !, #, and $ (see Section 9.5.2, "Declaring Variable Types"). The first character in a variable name must be a letter. If a variable begins with **FN**, it is assumed to be a call to a user-defined function that has been defined with the **DEF FN** statement. A variable name cannot be a reserved word, but embedded reserved words are allowed.

For example, the following statement is illegal because **LOG** is a reserved word:

```
LOG = 8
```

Reserved words include all BASIC commands, statements, function names, and operator names (see Appendix B, "QuickBASIC Reserved Words," for a complete list of reserved words).

9.5.2 Declaring Variable Types

Variable names can be declared as either numeric values or string values. String-variable names are written with a dollar sign ($) as the last character, noted in the following example:

```
A$ = "SALES REPORT"
```

The dollar sign is the type-declaration character for string variables; that is, it "declares" that the variable will represent a string. Numeric-variable names can declare integer values (denoted by a "%" suffix), single-precision values (denoted by either a "!" suffix, or no suffix), or double-precision values (denoted by a "#" suffix). Computations with integer- and single-precision variables are less accurate than those with double-precision

variables. However, you may want to declare a variable to have a lower precision type for one of the following reasons:

- Variables of higher precision use more memory.
- Arithmetic computation times are longer for higher precision numbers; a program with repeated calculations runs faster with integer variables.

The default type for a numeric variable is single precision.

The type-declaration characters for variables and the memory requirements (in bytes) for storing each variable type are listed in Table 9.1.

Table 9.1

Variable-Type Memory Requirements

Declaration Character	Variable Type	Bytes Required
%	Integer	2
!	Single precision	4
#	Double precision	8
$	String	4 bytes for descriptor 2 bytes for string back pointer 1 byte for each character in string

The following are examples of BASIC variable names:

Variable Name	Variable Type
PI#	Double precision
MINIMUM!	Single precision
LIMIT%	Integer
FIRSTNAME$	String
ABC	Single precision (default type)

The BASIC statements **DEFINT**, **DEFSTR**, **DEFSNG**, and **DEFDBL** can be included in a program to declare the types for certain variable

names. By using one of these **DEF***type* statements, you can specify that all variables starting with a given letter or range of letters will be of a certain variable type, without having to use the trailing declaration character. For more information on the **DEF***type* statements, see **DEF***type* statement reference page in Chapter 11, "Statement and Function Reference."

9.5.3 Array Variables

An array is a group or table of values referred to by the same variable name. The individual values in an array are called elements. Array elements are also variables, and can be used in any BASIC statement or function that uses variables. Declaring the name and type of an array, and setting the number of elements in the array, is known as *dimensioning* the array.

Each element in an array is referred to by an array variable that is subscripted with an integer or an integer expression (single- and double-precision expressions can be used as array subscripts; however, they are truncated to integer values). An array-variable name has as many subscripts as there are dimensions in the array. For example, V(10) refers to a value in a one-dimensional array, while T$(1,4) refers to a value in a two-dimensional string array. The default maximum subscript value for any array dimension is 10; this maximum value can be changed by using the **DIM** statement. (See the reference pages for **DIM** in Chapter 11, "Statement and Function Reference," for more information). The maximum number of dimensions for an array is 63. The maximum number of elements per dimension is 32,768. The maximum amount of memory that can be taken for an array is 64K.

Note

The array-variable T and the simple variable T in the following example refer to two distinct variables:

```
DIM T(11)
T = 2 : T(0) = 1              'T is simple variable
FOR I% = 0 TO 10              'T(0) is element of array
    T(I% + 1) = T * T(I%)
NEXT
```

Array elements, like nonarray variables, require a certain amount of memory, depending on the variable type. See Table 9.1 for information on the memory requirements for storing arrays.

9.6 Expressions and Operators

An expression can be a string or numeric constant, a variable, or a single value obtained by combining constants, variables, and other expressions with operators. Operators perform mathematical or logical operations on values. The operators provided by BASIC can be divided into four categories, as follows:

1. Arithmetic
2. Relational
3. Logical
4. Functional

9.6.1 Hierarchy of Operations

The BASIC operators have an order of precedence; that is, when several operations take place within the same program statement, certain kinds of operations will be executed before others. Operations are executed in the following order:

A. Arithmetic operators
   1. Exponentiation
   2. Negation
   3. Multiplication and division
   4. Integer division
   5. Modulo arithmetic
   6. Addition and subtraction

B. Relational operators (=, >, <, <>, <=, and >=)

C. Logical operators
   1. **NOT**
   2. **AND**
   3. **OR** and **XOR**
   4. **EQV**
   5. **IMP**

An exception to the order of operations listed above occurs when an expression has adjacent exponentiation and negation operators; in this case, the negation is executed first. For example, the following statement prints .0625 (4^{-2}), not –16 (-4^2):

```
print 4 ^ - 2
```

If the operations are of the same level, the left-most one is executed first, the right-most last, as in the following example:

```
A = 3 + 6 / 12 * 3 - 2  'A = 2.5
```

The order of operations in the preceding example is as follows:

1. 6 / 12
2. 0.5 * 3
3. 3 + 1.5
4. 4.5 – 2

9.6.2 Arithmetic Operators

Use parentheses to change the order in which arithmetic operations are performed. Operations within parentheses are performed first. Inside parentheses, the usual order of operation is maintained. Here are some sample algebraic expressions and their BASIC counterparts:

Algebraic Expression	BASIC Expression
$\frac{X-Y}{Z}$	(X-Y)/Z
$\frac{XY}{Z}$	X*Y/Z
$\frac{X+Y}{Z}$	(X+Y)/Z
$(X^{2})^{Y}$	(X^2)^Y
$X^{Y^{Z}}$	X^(Y^Z)
X(-Y)	X*(-Y)

Generally, two consecutive operators must be separated by parentheses. Exceptions to this rule are * -, * +, ^ -, and ^ +; therefore, the last expression in the right-hand column above could also be written X*-Y.

9.6.2.1 Integer Division

Integer division is denoted by the backslash (\) instead of the forward slash (/), which indicates floating-point division. Before integer division is performed, operands are rounded to integers, and hence must be greater than –32,768.5 and less than +32,767.5. The quotient of an integer division is truncated to an integer.

Example

```
PRINT 10\4, 10/4, -32768.499\10, -32768.499/10
```

Output:

```
 2             2.5          -3276          -3276.8499
```

9.6.2.2 Modulo Arithmetic

Modulo arithmetic is denoted by the modulus operator **MOD**. Modulo arithmetic provides the integer value that is the remainder of an integer division.

Example

```
X% = 10.4\4
REMAINDER% = INT(10.4) - 4*X%    '10\4 = 2, with remainder 2
PRINT REMAINDER%, 10.4 MOD 4
```

Output:

```
 2             2
```

9.6.2.3 Overflow and Division by Zero

If the evaluation of a numeric expression results in division by 0 (case 1), 0 being raised to a negative power (case 2), or overflow (case 3), then one of the following things will happen:

1. If the program was compiled with the Debug (/**d**) option, it will display a "Division by Zero" (case 1 or case 2) or "Overflow" (case 3) error message.
   a. If there is no error-trapping routine, the program ends.
   b. If there is an error-trapping routine to handle the error, control branches to the target label (see **ON ERROR GOTO** in Chapter 11, "Statement and Function Reference," for more information on error trapping).
2. If the program was compiled without the Debug option, the error is not detected.

9.6.3 Relational Operators

Relational operators are used to compare two values, as shown in Table 9.2. The result of the comparison is either "true" (nonzero) or "false" (0). This result can then be used to make a decision regarding program flow.

Table 9.2

Relational Operators and Their Functions

Operator	Relation Tested	Expression
=	Equality*	X = Y
< >	Inequality	X < > Y
<	Less than	X < Y
>	Greater than	X > Y
<=	Less than or equal to	X <= Y
>=	Greater than or equal to	X >= Y

*The equal sign is also used to assign a value to a variable.

When arithmetic and relational operators are combined in one expression, the arithmetic operation is always performed first. For example, the following expression is true if the value of X plus Y is less than the value of T - 1 divided by Z:

```
X + Y < (T - 1)/Z
```

9.6.4 Logical Operators

Logical operators perform tests on multiple relations, bit manipulations, or Boolean operations, and return a true (nonzero) or false (zero) value to be used in making a decision.

Examples

```
IF D < 200 AND F < 4 THEN 80

WHILE I > 10 OR K < 0
.
.
.
WEND

IF NOT P THEN PRINT "Name not found"
```

There are six logical operators in BASIC; they are listed below in order of precedence:

Operator	Operation
NOT	Logical complement
AND	Conjunction
OR	Disjunction (inclusive or)
XOR	Exclusive or (same precedence as **OR**)
EQV	Equivalence
IMP	Implication

Each operator returns results as indicated in Table 9.3. A "T" indicates a true value and an "F" indicates a false value. Operators are listed in order of operator precedence.

Table 9.3

Values Returned by Logical Operations

X	Y	NOT X	X AND Y	X OR Y	X XOR Y	X IMP Y	X EQV Y
T	T	F	T	T	F	T	T
T	F	F	F	T	T	F	F
F	T	T	F	T	T	T	F
F	F	T	F	F	F	T	T

In an expression, logical operations (also known as Boolean operations) are performed after arithmetic and relational operations. Logical operators convert their operands to 16-bit, signed, two's complement integers in the range –32,768 to +32,767. (If the operands are not in this range, an error results.) If both operands are supplied as 0 or –1, logical operators return 0 or –1, respectively, as in the following example. (Note the similarity of the output from this program to Table 9.3: "T" becomes –1, and "F" becomes 0.)

Example

```
PRINT " X       Y       NOT     AND     OR      XOR     IMP     EQV"
PRINT
I = 10 : J = 15
X = (I = 10) : Y = (J = 15) 'X is true (-1); Y is true (-1)
GOSUB TRUTHTABLE
X = (I > 9) : Y = (J > 15)  'X is true (-1); Y is false (0)
GOSUB TRUTHTABLE
X = (I <> 10) : Y = (J < 16)          'X is false (0); Y is true (-1)
GOSUB TRUTHTABLE
X = (I < 10) : Y = (J < 15) 'X is false (0); Y is false (0)
GOSUB TRUTHTABLE
END

TRUTHTABLE:
   PRINT X "     " Y "     ";NOT X "     " X AND Y "     " X OR Y;
   PRINT "     " X XOR Y "     " X IMP Y "     " X EQV Y
   PRINT
RETURN
```

Output:

```
 X       Y       NOT     AND     OR      XOR     IMP     EQV

-1      -1       0      -1      -1       0      -1      -1

-1       0       0       0      -1      -1       0       0

 0      -1      -1       0      -1      -1      -1       0

 0       0      -1       0       0       0      -1      -1
```

Logical operators compare each bit of the first operand with the corresponding bit in the second operand to compute the bit in the result; in these "bit-wise" comparisons, a 0 bit is equivalent to a "false" value (F) in Table 9.3, while a 1 bit is equivalent to a "true" value (T).

It is possible to use logical operators to test bytes for a particular bit pattern. For instance, the **AND** operator can be used to mask all but one of the bits of a status byte, while the **OR** operator can be used to merge two bytes to create a particular binary value.

Example

```
PRINT 63 AND 16          '    63 (binary 111111)
                         'AND 16 (binary  10000)
                         '-----------------
                         '    16 (binary  10000)

PRINT -1 AND 8           '    -1 (binary  1111111111111111)
                         'AND  8 (binary              1000)
                         '--------------------------------
                         '     8 (binary              1000)

PRINT 10 OR 9            '    10 (binary 1010)
                         'OR   9 (binary 1001)
                         '--------------------
                         '    11 (binary 1011)

PRINT 10 XOR 10,         'Always 0

PRINT NOT 10, NOT 11, NOT 0      'NOT X = -(X + 1)
```

Output:

```
 16
 8
 11
 0          -11           -12           -1
```

9.6.5 Functional Operators

A function is used in an expression to call a predetermined operation to be performed on an operand. For example, **SQR** is a functional operator used twice in the following assignment statement:

```
A = SQR (20.25) + SQR (37)
```

BASIC incorporates two kinds of function: intrinsic and user-defined.

9.6.5.1 Intrinsic Functions

Functions perform operations on their operands and return values. BASIC has many predefined functions built into the language. Examples are the **SQR** (square root) and **SIN** (sine) functions.

9.6.5.2 User-Defined Functions

Functions can be defined by the user with the **DEF FN** statement. These functions are defined only for the life of a given program, and are not part of the BASIC language. In addition to **DEF FN**, the Microsoft QuickBASIC Compiler allows you to define subprograms. (For more information on defining your own functions and subprograms, see the entries for **DEF FN** and **SUB...END SUB** in Chapter 11, "Statement and Function Reference.")

9.6.6 String Operators

A string expression consists of string constants, string variables, and other string expressions combined by string operators. There are two classes of string operations: concatenation and string function.

The act of combining two strings is called concatenation. The plus symbol (+) is the concatenation operator for strings. For example, the following program fragment combines the string variables A$ and B$ to produce the value FILENAME.

```
A$ = "FILE": B$ = "NAME"
PRINT A$ + B$
PRINT "NEW " + A$ + B$
```

Output:

```
FILENAME
NEW FILENAME
```

Strings can be compared using the following relational operators:

< > = < > <= >=

Note that these are the same relational operators used with numbers.

String functions are similar to numeric functions, except that the operands are strings rather than numeric values. String comparisons are made by taking corresponding characters from each string and comparing their ASCII codes. If the ASCII codes are the same for all the characters in both strings, the strings are equal. If the ASCII codes differ, the lower code number precedes the higher. If the end of one string is reached during string comparison, the shorter string is said to be smaller if they are equal up to that point. Leading and trailing blanks are significant. The following are examples of true statements:

```
"AA" < "AB"
"FILENAME" = "FILENAME"
"X&" > "X#"
"CL " > "CL"
"kg" > "KG"
"SMYTH" < "SMYTHE"
B$ < "9/12/78"                    'where B$ = "8/12/85"
```

Thus string comparisons can be used to test string values or to alphabetize strings. All string constants used in comparison expressions must be enclosed in quotation marks. See Appendix A, "ASCII Character Codes," for more information on the ASCII collating sequence.

9.7 Type Conversion

When necessary, BASIC will convert a numeric constant from one type to another, according to the following rules:

- If a numeric constant of one type is set equal to a numeric variable of a different type, the numeric constant will be stored as the type declared in the variable name, as in the following example:

  ```
  A% = 23.42
  PRINT A%
  ```

  Output:

  ```
   23
  ```

  However, if a string variable is set equal to a numeric value, or vice versa, a "Type Mismatch" error message is generated.

- During expression evaluation, all of the operands in an arithmetic or relational operation are converted to the same degree of precision: that of the most precise operand. Also, the result of an arithmetic operation is returned to this degree of precision, as in the following example:

  ```
  X% = 2 : Y! = 1.5 : Z# = 100
  PRINT X% / Y! * Z#
  ```

  Output:

  ```
   133.3333373069763
  ```

  Although the preceding result is displayed in double precision,

(because of the double-precision variable Z#), it has only single-precision accuracy, since the first operation (X% / Y!) is calculated with single-precision accuracy. This explains the nonsignificant digits (73069763) after the fifth decimal place. Contrast this with the output from the following example:

```
X% = 2 : Y# = 1.5 : Z# = 100
PRINT X% / Y# * Z#
```

Output:

```
 133.3333333333333
```

- Logical operators convert their operands to integers and return an integer result. Operands must be in the range –32,768 to +32,767 or an "Overflow" error message is generated.

- When a floating-point value is converted to an integer, the fractional portion is rounded, as in this example:

```
TOTAL% = 55.88
PRINT TOTAL%
```

Output:

```
 56
```

Chapter 10
Compiler-Interpreter Language Differences

10.1 New Programming Features 183
10.1.1 Dynamic and Static Arrays 184
10.1.2 Large Numeric Arrays 186
10.2 New BASIC Statements and Functions 186
10.3 Using Metacommands 187
10.3.1 Metacommand Syntax 187
10.3.2 Processing Additional Source Files: $INCLUDE 188
10.3.3 Dimensioned Array Allocation: $STATIC and $DYNAMIC 188
10.4 Compiler-Interpreter Differences 189
10.4.1 A Comparison of Compilation and Interpretation 189
10.4.2 Source-File Format 190
10.4.3 Cassette BASIC 190
10.4.4 Operational Differences 191
10.4.5 Statements and Functions Prohibited by the Compiler 192
10.4.6 Statements Requiring Modification 193

This chapter explains how to use some of the features added to QuickBASIC to create more efficient and useful programs. It also describes the difference between interpretation and compilation, and lists the differences between interpreted BASIC and QuickBASIC. Even if you are an experienced BASIC programmer, you should read this chapter to learn about the new features, and to find out if you need to change any source files created with the interpreter.

Table 10.1 lists the functions and statements covered in this chapter. See Chapter 11, "Statement and Function Reference," for a complete reference to all the QuickBASIC statements and functions.

Table 10.1

Functions and Statements Described in This Chapter

Enhanced in Compiler	New in Compiler	May Require Modifying of Interpreter Programs
COMMON	COMMAND$	BLOAD/BSAVE
DEF FN	LBOUND	CALL
END	LOCK	CALLS
ERASE	PALETTE	CHAIN
FOR...NEXT	PCOPY	COMMON
FRE	REDIM	DEF*type*
GOSUB	SADD	DIM
GOTO	SHARED	DRAW
IF...THEN...ELSE	STATIC	PLAY
OPEN	SUB...END SUB	RESUME
RETURN	UBOUND	RUN
WIDTH	UNLOCK	

10.1 New Programming Features

Microsoft QuickBASIC has several new features that allow you to

- Write more readable programs (line extension and line labels: see Section 9.2)
- Use memory more efficiently (dynamic and static arrays: see Section 10.1.1)
- Use large amounts of data (large numeric arrays: see Section 10.1.2)

The following sections describe these features.

10.1.1 Dynamic and Static Arrays

The QuickBASIC compiler supports two types of arrays: static arrays and dynamic arrays. An array is static if space to hold its elements is allocated at compile time; it is dynamic if allocation occurs at run time. Static arrays occupy slightly less space in memory than dynamic arrays, and static array elements can be accessed faster. However, dynamic arrays provide more efficient use of memory because space for the array is not allocated until it is needed.

Arrays dimensioned with constant subscripts are considered static by default; arrays dimensioned with variable subscripts are considered dynamic. You can use the **$STATIC** and **$DYNAMIC** metacommands, which are explained in Section 10.3.3, to override these defaults. Both static and dynamic arrays are dimensioned with the **DIM** statement, which has the same syntax in the compiler as in the interpreter.

Dynamic arrays can be redimensioned with the **REDIM** statement. The **REDIM** statement changes the amount of space allocated to a dynamic array. When a **REDIM** statement is executed, the array it refers to is deallocated, then reallocated with the new dimensions. The old array-element values are lost. (Note that **REDIM** does not require a previous **DIM** statement, and thus can also allocate new arrays.) Static arrays cannot be reallocated with **REDIM**.

The **ERASE** statement operates differently on static and dynamic arrays. On a static array, **ERASE** sets all array elements to zero, or to null strings. On a dynamic array, **ERASE** deallocates the array. The dynamic array must be redimensioned with a **DIM** statement before it can be referenced again.

The syntax of the **ERASE** and **REDIM** statements is described in Chapter 11, "Statement and Function Reference."

The following three rules apply when dimensioning arrays with **DIM**:

1. Static arrays can be dimensioned only once. Dynamic arrays can be redimensioned using the **REDIM** statement, or with an **ERASE** *array* **DIM** *array* sequence.
2. **DIM** statements that allocate dynamic arrays are considered executable statements, and must appear after all **COMMON** statements in the program.

3. When a **DIM** statement that allocates a dynamic array is executed, the array must be currently unallocated, otherwise a "Redimensioned array" error message appears.

Warning

Bounds checking is performed on arrays only when the program is compiled with the Debug (**/d**) option. Severe run-time errors may result when the bounds of an array are exceeded in programs compiled without the Debug option. The upper bound of an implicitly dimensioned static array defaults to 10.

Examples

The following example allocates space for a static two-dimensional array:

```
REM $STATIC
DIM A(10,20)
```

The next example allocates space for dynamic arrays C and D:

```
10 INPUT "how many?"; N
20 REM $DYNAMIC
30 DIM C(2,3,4)
40 DIM D(N)
```

Line 50 sets all elements to 0 in static array A; line 60 erases dynamic arrays C and D:

```
50 ERASE A
60 ERASE C,D
```

Line 70 redimensions array C:

```
70 REDIM C(4,5,6)
```

10.1.2 Large Numeric Arrays

Microsoft QuickBASIC permits the use of large numeric arrays. Although the size of each array is limited to 64K, you can have as many numeric arrays as will fit in memory. Large numeric arrays must be dynamic.

10.2 New BASIC Statements and Functions

The statements and functions in Table 10.2 are new to QuickBASIC, and are not available in most other Microsoft BASIC products.

Table 10.2
New BASIC Statements and Functions

Statement/Function	Description
COMMAND$	Returns the string specified after the /cmd option on the invocation command line
LBOUND/UBOUND	Returns the lower and upper bounds for a specified array dimension
LOCK...UNLOCK	Controls access by other processes to all or part of an opened file in a networking environment
PALETTE	Changes the way color numbers are mapped to display colors
PCOPY	Copies one screen page to another
REDIM	Changes the space allocated to a dynamic array
SADD	Returns the address of a string expression
SHARED	Gives global access to variables
STATIC	Designates multiline function or subprogram variables as local to the function or subprogram
SUB...END SUB	Marks the beginning and end of a subprogram

10.3 Using Metacommands

Metacommands tell the compiler to perform certain actions while it is compiling the source file. QuickBASIC metacommands can do the following:

- Read in and compile other BASIC source files at specific points during compilation (**$INCLUDE**)
- Control the allocation of dimensioned arrays (**$STATIC** and **$DYNAMIC**)

Sections 10.3.1, 10.3.2, and 10.3.3 describe the metacommands available with the Microsoft QuickBASIC Compiler, and how to use them.

10.3.1 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 the white-space characters: space, tab, or line feed. 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 between the elements of a metacommand is ignored. The following are all valid forms for metacommands:

```
REM $METACOMMAND1 $METACOMMAND2
REM $METACOMMAND1 : 'string argument'  $METACOMMAND2
```

Note that *no space* may appear between the dollar sign and the rest of the metacommand.

To put comment-only metacommands in comments, 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$METACOMMAND1 $METACOMMAND2
```

10.3.2 Processing Additional Source Files: $INCLUDE

The **$INCLUDE** metacommand instructs the compiler to switch processing from the current source file to the BASIC file named in the argument. When the included source end-of-file is reached, the compiler continues processing the original file. Because compilation begins with the line immediately following the line in which **$INCLUDE** occurred, **$INCLUDE** should be the last statement on the line. The following statement is correct:

```
999 DEFINT I-N : REM $INCLUDE: 'COMMON.BAS'
```

The following three restrictions must be observed:

1. Variables in included files must match variables in the main program.
2. Included files must not contain **END** statements, or **GOTO** statements to nonexistent lines.
3. Included files created with the interpreter must be saved with the **,A** option.

10.3.3 Dimensioned Array Allocation: $STATIC and $DYNAMIC

The **$STATIC** and **$DYNAMIC** metacommands tell the compiler how to allocate memory for arrays. If a **$DYNAMIC** or **$STATIC** metacommand is present, all subsequent array declarations in a **DIM** statement are allocated accordingly, with one exception: implicitly dimensioned arrays (arrays *not* dimensioned with the **DIM** statement) are considered static even when the **$DYNAMIC** metacommand is present.

If the **$STATIC** metacommand is present, all subsequent arrays in a source file are allocated as static.

Using **$STATIC** and **$DYNAMIC** with the **DIM**, **REDIM**, **ERASE**, and **COMMON** statements is discussed in detail in Section 10.1.1, "Dynamic and Static Arrays."

10.4 Compiler-Interpreter Differences

The BASIC language understood by the QuickBASIC compiler is slightly different from, but generally compatible with, the Microsoft GW-BASIC® Interpreter and the IBM Personal Computer BASIC Interpreter found on many compatible computers. Effort has been made to preserve as much compatibility as possible, but some changes are required because of internal differences between the compiler and the interpreter.

If you wish to compile programs originally developed for the interpreter, you may need to modify the source code in your interpreted BASIC program, or use a special compiler option, for one of the following four reasons:

1. Your program would work more efficiently with some of the new or enhanced statements and functions supported by the QuickBASIC compiler.
2. Your program may contain statements and functions that work differently in the compiler and the interpreter.
3. Your program may contain statements and functions, such as **ON ERROR GOTO**, that require it to be compiled with a special option. (See Section 10.4.4, "Operational Differences," for a list of situations where special options are required.)
4. Your interpreted BASIC program may use one of the statements or functions not supported by the compiler.

The following sections describe these language differences and some rules you should keep in mind when preparing source files for the compiler.

10.4.1 A Comparison of Compilation and Interpretation

A microprocessor can execute only its own machine language instructions; it cannot directly execute BASIC language statements. Before a program can be executed, the BASIC program statements must be translated into the machine language of the microprocessor. Compilers and interpreters are two types of programs that perform this translation. The following discussion explains the difference between the two translation processes.

When you run an interpreted BASIC program, the interpreter analyzes each BASIC statement, translates it into machine language, and executes the statement immediately. Statements that are executed repeatedly (inside a **FOR...NEXT** loop, for example) are translated before each execution of the statement. The interpreter cannot look ahead in the program to see what's coming up, or analyze the whole program and perform actions related to that analysis, such as size and speed optimizations. The compiler, on the other hand, translates the entire source program into machine code *before* any individual program instruction is executed. It examines the entire program and generates machine code that executes as fast as possible.

Compiled programs execute faster than interpreted programs. The execution time you save with the compiler depends on the content of your program, but in most cases, QuickBASIC programs execute 3 to 10 times faster than interpreted programs. If you make maximum use of integer variables, program execution can be up to 30 times faster. Programs with lots of graphics statements, or many input or output operations, will not benefit as much because of hardware speed limitations.

10.4.2 Source-File Format

The compiler expects the source file to be in ASCII (American Standard Code for Information Interchange) format. If you create a file with the BASIC interpreter, it must be saved with the **,A** option; otherwise, the interpreter encodes the text of your BASIC program in a special format that the compiler cannot read. If this happens, reload the BASIC interpreter and resave the file in ASCII format, using the **,A** option. For example, the following interpreter command saves the file MYPROG.BAS in ASCII format:

```
SAVE "MYPROG.BAS",A
```

10.4.3 Cassette BASIC

QuickBASIC does not support the cassette device.

10.4.4 Operational Differences

If your program contains any of the statements listed in Table 10.3, you must compile it with the options indicated. Compiling with options is explained in detail in Section 4.4.3, "Compile...". If you are bypassing the QuickBASIC work area and menus, the required command-line options are listed in Appendix D, "Using the Separate Compilation Method."

Table 10.3

Operational Differences

Statement	Option
ON ERROR GOTO	On Error
ON *event* **GOSUB**	Event Trapping
KEY ON/OFF/STOP	Event Trapping or Checking Between Statements
PEN ON/OFF/STOP	Event Trapping or Checking Between Statements
PLAY ON/OFF/STOP	Event Trapping or Checking Between Statements
RESUME	Resume Next
RESUME NEXT	Resume Next
RESUME 0	Resume Next
STRIG ON/OFF/STOP	Event Trapping or Checking Between Statements
TIMER ON/OFF/STOP	Event Trapping or Checking Between Statements
TRON/TROFF	Debug

The compiler supports the following error-handling statements and functions. They work the same as in the interpreter, with the exceptions indicated in Table 10.4.

Table 10.4

Error-Handling Exceptions

Statement or Function	Exception
ERL	Only reports numeric line numbers, not line labels, and returns the nearest line number *above* the error
ERR	None
ERROR	None
ON ERROR GOTO	Can refer to line numbers or labels
RESUME *line*	Can refer to line numbers or labels

10.4.5 Statements and Functions Prohibited by the Compiler

The statements and functions listed in Table 10.5 cannot be used in a compiled program because they perform editing operations on the source file, they interfere with program execution, or they require cassette support.

Table 10.5

Statements and Functions Prohibited by the Compiler

AUTO	LIST	NEW
CONT	LLIST	RENUM
DEF USR	LOAD	SAVE
DELETE	MERGE	USR
EDIT	MOTOR	

10.4.6 Statements Requiring Modification

If your interpreter program contains any of the statements listed in Table 10.6, you will probably need to modify your source code before you can compile the program. These statements are described in detail in Chapter 11, "Statement and Function Reference."

Table 10.6

Statements Requiring Modification

Statement	Modification
BLOAD/BSAVE	Memory locations may be different in the compiler.
CALL*name*	The *name* argument is the name of the subroutine or subprogram being called.
CHAIN	The compiler 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 compiled programs.
DIM	All **DIM** statements declaring static arrays must appear at the beginning of compiled programs.
DRAW	The compiler requires that the **VARPTR$** function be used with embedded variables.
PLAY	The compiler requires that the **VARPTR$** function be used with embedded variables.
RESUME	If an error occurs in a single-line function, the compiler attempts to resume program execution at the line containing the function.
RUN	The object of a **RUN** statement cannot be a **.BAS** file; it must be an executable (**.EXE**) program. The interpreter **R** option is not supported. **RUN** {*linenumber*|*linelabel*}, however, is supported in the compiler; this restarts the program at the specified line.

Increased functionality has been added to the functions and statements in Table 10.7. These functions and statements are described in Chapter 11, "Statement and Function Reference."

Table 10.7
Enhanced Statements and Functions

Statement/Function	Enhancement
COMMON	Data can be shared between the main program and subprograms.
DEF FN	The compiler allows multiline functions, as well as single-line functions.
END	Ends multilinefunction definitions and subroutines, as well as BASIC programs.
ERASE	For static arrays, **ERASE** resets array elements to null strings or zeros; for dynamic arrays, **ERASE** clears arrays from memory.
FOR...NEXT	The compiler supports double-precision control values in **FOR...NEXT** loops.
FRE	**FRE**(–1) returns the size of the largest free block of large numeric-array space; **FRE**(*numeric-expression*), where *numeric-expression* $\neq$ –1, returns the size of the next free block of string space; **FRE**(*string-expression*) returns the size of the next free block of string space, *after* first compacting free string space.
GOSUB...RETURN	The compiler supports line labels, as well as line numbers.
GOTO	The compiler supports branching to line labels, as well as to line numbers.
IF...THEN...ELSE	The compiler supports "block" (multiline) **IF...THEN...ELSE** statements.
RESUME	The compiler supports resuming at line labels, as well as at line numbers.
OPEN	Controls access to opened files in a multi-user environment.
WIDTH	Can assign line width to device (**SCRN**) and disk files.

Chapter 11

Statement and Function Reference

Each statement and function in the alphabetical reference that follows is described using the following format:

Heading	Function
■ **Syntax**	Gives the correct syntax for the statement or function. Functions return a value of a particular type and can be used wherever an expression can be used; functions cannot appear by themselves on a BASIC line. Statements do not return a value, but they can appear by themselves on a BASIC program line.
■ **Action**	Summarizes what the statement or function does.
■ **Remarks**	Describes arguments and options in detail, and explains how to use the statement or function.
■ **Compiler/Interpreter Differences**	Tells whether or not the compiler statement or function is new, enhanced, or different from the same statement in the interpreter. (This is an optional section that does not appear with every reference entry.)
■ **See Also**	Cross-references related statements and functions. (This is an optional section that does not appear with every reference entry.)
■ **Example**	Gives sample commands, programs, and program segments that illustrate the use of the given statement or function. (This is an optional section that does not appear with every reference entry.)

ABS Function

■ Syntax

ABS(x)

■ Action

Returns the absolute value of the numeric expression x

■ Remarks

The absolute-value function returns the unsigned magnitude of its argument; for example, the absolute value of –1 and the absolute value of 1 are both 1.

■ Example

```
PRINT ABS(7*(-5)), ABS(7*5)
```

Output:

```
 35          35
```

■ Syntax

ASC(*string-expression*)

■ Action

Returns a numerical value that is the ASCII code for the first character of the string *string-expression*

■ Remarks

If *string-expression* is null, an "Illegal function call" error is returned.

■ See Also

CHR$

■ Example

```
X$ = "TEST"
PRINT ASC(X$)              'Prints ASCII for "T"
```

Output:

```
 84
```

■ Syntax

ATN(*numeric-expression*)

■ Action

Returns the arctangent of *numeric-expression*, or the angle whose tangent is *numeric-expression*

■ Remarks

The result is given in radians, and is in the range $-\pi/2$ to $\pi/2$ radians, where $\pi = 3.141593$. The expression *numeric-expression* can be any numeric type. **ATN** is evaluated by default in single precision; if *numeric-expression* is a double-precision value, **ATN** is evaluated in double precision.

■ Example

```
X = 3
PRINT ATN(X)
```

Output:

```
 1.249046
```

Thus an angle of `1.249046` radians has a tangent of `3`.

■ Syntax

BEEP

■ Action

Sounds the speaker

■ Remarks

The **BEEP** statement sounds the ASCII bell character. This statement has the same effect as the following statement:

```
PRINT CHR$(7)
```

The sound **BEEP** makes in a compiled program differs in both duration and frequency from the sound it makes in an interpreted program.

■ Example

This example checks to see if X is out of range. If X is less than 20, the computer beeps.

```
IF X < 20 THEN BEEP
```

■ Syntax

BLOAD *filespec* [[,*offset*]]

■ Action

Loads a specified memory-image file, created by **BSAVE**, into memory from any input device

■ Remarks

The *filespec* is a string expression for the file specification; it can contain a path specification.

The *offset* is a numeric expression returning a single-precision value in the range 0 to $2^{20}-1$. This is the offset address at which loading is to start in the segment declared by the last **DEF SEG** statement. The **BLOAD** statement allows a program or data that has been saved as a memory-image file to be loaded anywhere in memory. A memory-image file is a byte-for-byte copy of what was originally in memory.

If the offset is omitted, the segment address and offset contained in the file (i.e., the address specified by the **BSAVE** statement when the file was created) are used. Therefore, the file is loaded into the same location from which it was saved.

If the offset is specified, the segment address used is the one given in the most recently executed **DEF SEG** statement. If no **DEF SEG** statement has been given, the BASIC data segment (**DS**) will be used as the default (because it is the default for **DEF SEG**).

Important

BLOAD does not perform an address-range check. It is therefore possible to load a file anywhere in memory. The user must be careful not to load over BASIC or the operating system. When loading and saving graphic images, beware of problems associated with different memory configurations for different **SCREEN** modes.

The **BLOAD** and **BSAVE** statements do not support the cassette device.

■ See Also

BSAVE

■ Example

```
'Load subroutine at 60:F000:
DEF SEG=&H6000  'Set segment to 6000 Hex
BLOAD "PROG1", &HF000  'Load PROG1
```

This example sets the segment address at &H6000 and loads PROG1 at &HF000.

■ Syntax

BSAVE *filespec,offset,length*

■ Action

Transfers the contents of the specified area of memory to an output device

■ Remarks

The *filespec* is a string expression for the file specification; it can contain a path specification.

The *offset* is a numeric expression returning a single-precision value in the range 0 to $2^{20}-1$. This is the offset address from which saving starts in the segment declared by the last **DEF SEG** statement.

The *length* is a numeric expression returning an unsigned integer in the range 1–65,535. This is the length in bytes of the memory-image file to be saved.

The *filespec*, *offset*, and *length* are all required in the syntax.

The **BSAVE** statement allows data or programs to be saved as memory-image files on disk. A memory-image file is a byte-for-byte copy of what is in memory; this file contains information used by **BLOAD** when loading the file. When loading and saving graphic images, beware of problems associated with different memory configurations for different **SCREEN** modes.

If the offset is omitted, a "Bad file name" error message is issued and the save is terminated. If no **DEF SEG** statement has been executed before the **BSAVE** statement, the program uses the default BASIC data segment (**DS**).

If length is omitted, a "Bad file name" error message is issued and the save is terminated.

The **BLOAD** and **BSAVE** statements do not support the cassette device.

■ See Also

BLOAD

■ Example

```
'Save PROG1:
DEF SEG=&H6000
BSAVE "PROG1", &HF000, 256
```

This example saves 256 bytes starting at 6000:F000 in the file PROG1.

■ Syntax

CALL *name*[[(*argumentlist*)]]
CALL ABSOLUTE ([[*argumentlist*,]] *integervariable*)

■ Action

Calls and transfers control to a subprogram or assembly-language subroutine

■ Remarks

The **CALL** statement takes the following arguments:

Argument	Description
name	The name of the BASIC subprogram or assembly-language subroutine being called. It is limited to 31 characters. If *name* is a BASIC subprogram, it must be declared in a **SUB** statement. If *name* refers to an assembly-language subroutine, it must be a **PUBLIC** name (symbol) within the subroutine. Note that **PUBLIC** names containing "$" and "_" are not permitted, as these names might be used by the run-time system. Duplicate names cause a "Symbol defined more than once" linker error at link time.
argumentlist	The variables or constants that **CALL** passes to the subprogram or subroutine. Each argument in the list is separated from other arguments in the list by commas. Note that these arguments are passed by reference, so their values can be changed by the subprogram or subroutine.

If the *argumentlist* includes an array argument, the array should be specified by the array name, followed by empty parentheses, as in this example:

```
DIM INTARRAY(20)
CALL SHELLSORT(INTARRAY())
```

The **CALLS** statement is also used to call assembly-language subroutines. It has the same syntax as **CALL**, and the same purpose, except that **CALLS** passes the segmented addresses of arguments in the *argumentlist*, while **CALL** passes unsegmented addresses.

The **CALL ABSOLUTE** statement, used to call machine-language subroutines, takes the following arguments:

Argument	Description
argumentlist	Optional arguments passed to a machine-language subroutine.
integervariable	An integer variable containing a value that is the offset from the beginning of the current code segment to the starting location in memory of the subroutine. The *integervariable* argument is not passed to the subroutine. Unless the called routine is in **DGROUP**, your program must execute a **DEF SEG** statement before **CALL ABSOLUTE** to ensure that the code segment is correct. If *integervariable* is not an integer, undefined results may occur.

■ Compiler/Interpreter Differences

Since the compiler allows strings of up to 32,767 characters in length, it uses a 4-byte string descriptor, rather than the 3-byte string descriptor used by the interpreter. The bytes are the low byte and high byte of the length, followed by the low byte and high byte of the address. If your BASIC program calls an assembly-language subroutine that uses string arguments, you may need to make changes in the subroutine byte-string descriptor to reflect this difference.

■ See Also

CALLS

■ Example

In the following example, the main program calls the BASIC subprogram PRINTOUT after first splitting the command line into three separate file names and storing them in the array FILE$(). The PRINTOUT subprogram writes out the contents of the first two files to both the third file and the standard output.

```
DEFINT A-Z
DIM FILE$(3)
CLINE$ = COMMAND$
LENGTH = LEN(CLINE$)
GOSUB ARGSPLIT                  'Separate command line into
                                '3 arguments
CALL PRINTOUT(FILE$())          'Send all three files to PRINTOUT
END

ARGSPLIT:                       'Subroutine to split command line
   'See example under COMMAND$ Function for definition of
   'ARGSPLIT:
RETURN

SUB PRINTOUT(F$(1)) STATIC
   OPEN F$(3) FOR OUTPUT AS #3          'Opens target file
   FOR FILE = 1 TO 2                    'Loop executes twice:
      OPEN F$(FILE) FOR INPUT AS #FILE  '1st time for FILE$(1)
      WHILE NOT EOF(FILE)               '2nd time for FILE$(2)
         LINE INPUT #FILE, TEMP$          'Read file
         WRITE #3, TEMP$                'Write data to target file
         PRINT TEMP$                    'Write file to standard
      WEND                              'output
    NEXT
    CLOSE
END SUB
```

Sample command line (for standalone executable file combine.exe):

```
combine main.inp  sub.inp   myprog.bas
```

Sample command line (for in-memory compilation):

```
qb combine.bas /cmd main.inp  sub.inp   myprog.bas
```

After either sample command line executes, the contents of main.inp and sub.inp will be combined and stored in myprog.bas.

■ Syntax

CALLS *name*[[(*argumentlist*)]]

■ Action

Calls an assembly-language subroutine

■ Remarks

The **CALLS** statement has the same syntax as **CALL**, and the same purpose, except that **CALLS** passes the segmented addresses of arguments in the *argumentlist*, while **CALL** passes unsegmented addresses. In addition, if you want to call a compiled BASIC subprogram, use the **CALL** statement, *not* **CALLS**.

■ Compiler/Interpreter Differences

With the interpreter only, **CALLS** uses the segment address, defined by the most recently executed **DEF SEG** statement, to locate the routine being called.

■ See Also

CALL

CDBL Function

■ Syntax

CDBL(*numeric-expression*)

■ Action

Converts the *numeric-expression* to a double-precision number

■ Remarks

This function has the same effect as assigning the numeric expression to a double-precision variable.

Note that the extra digits of precision are nonsignificant, unless the expression is calculated with double-precision accuracy, as shown in the following example.

■ Example

```
X = 7/9
X# = 7/9
PRINT X
PRINT X#          'Both X# and CDBL(X) will be accurate to only
PRINT CDBL(X)     '7 decimal places, since 7/9 evaluated in single
                  'precision
PRINT 7#/9#       'Accurate to 15 decimal places
```

Output:

```
 .7777778
 .7777777910232544
 .7777777910232544
 .7777777777777778
```

■ Syntax

CHAIN *filespec*

■ Action

Transfers control from current program to another program

■ Remarks

The *filespec* argument is a string expression that identifies the program to which control is passed; *filespec* can contain a path specification. If the file name is given without an extension, the compiler assumes the extension **.EXE**.

You can use **COMMON** to pass variables between programs.

If you are compiling inside the QuickBASIC environment, note that all chained-to programs must be compiled with the Exe output option. If you are compiling a program outside the QuickBASIC environment (that is, you are not compiling into memory), note that the **BCOM20.LIB** library does not support the use of **CHAIN** with **COMMON**. Therefore, if you are using **COMMON** to pass variables, both the chaining and chained-to programs must use the default **BRUN20.LIB** library; that is, the chaining and chained-to programs must be compiled *without* the **/o** option. Note also that, in programs compiled with **BCOM20.LIB**, **CHAIN** and **RUN** are synonymous.

When programs are compiled with **BRUN20.LIB**, files are left open during chaining, unless they are explicitly closed with a **CLOSE** statement.

■ Compiler/Interpreter Differences

The compiler assumes the extension **.EXE** if none is given in *filespec*, unlike the interpreter, which assumes the extension **.BAS**. Therefore, if the object of a **CHAIN** is a file specification without an extension, it should work with both the compiler and the interpreter. However, the object of a **CHAIN** *cannot* have the explicit extension **.BAS** in the compiler; the only explicit extension allowed in the compiler is **.EXE**.

The QuickBASIC compiler does not support the **ALL**, **MERGE**, **DELETE**, or *linenumber* options available with the interpreter. You should be careful when a chained-to program chains back to the chaining program; since you cannot specify a *linenumber* to return to, execution starts again at the beginning of the chaining program. There is a danger of going into an "endless loop" if this happens. (See the following example for a way to avoid this problem.)

■ See Also

CALL, COMMON

■ Example

The following example converts a number in any base to a decimal number. The base and the number are input in the main program, MAIN, which then chains to a second program, DIGIT. This second program splits the number from the main program into separate digits, converts these characters to numeric values, and stores the numeric values in an array, A().

Control then chains to a third program, DEC, which actually changes the number to a decimal number, deallocates (with ERASE A) the array in which the digits are stored, and prints the decimal number. Control then chains back to the main program, which repeats the process, as long as the value input for the base B is not zero. Note the use of COMMON to share variables in all three programs.

```
'This program is MAIN.BAS:
COMMON A(1),N$,B,LN
INPUT;"Base (enter 0 to end): ", B
IF B = 0 THEN END ELSE INPUT "          Number: ",N$
LN = LEN(N$)
DIM A(LN)
CHAIN "DIGIT"

'This program is DIGIT.BAS:
COMMON A(1),N$,B,LN
FOR I = 1 TO LN
    DGT$ = MID$(N$,I,1)
    IF DGT$ >= "0" AND DGT$ <= "9" THEN
        A(I) = VAL(DGT$)
    ELSEIF DGT$ >= "A" AND DGT$ <= "Z" THEN
        A(I) = ASC(DGT$) - ASC("A") + 10
    ELSEIF DGT$ >= "a" AND DGT$ <= "z" THEN
        A(I) = ASC(DGT$) - ASC("a") + 10
    ELSE
        PRINT "Illegal input value."
        CHAIN "MAIN"
    END IF
NEXT
CHAIN "DEC"

'This program is DEC.BAS:
COMMON A(1),N$,B,LN
DECIMAL = 0
FOR I = 1 TO LN
    DECIMAL = B*DECIMAL + A(I)
NEXT
ERASE A
PRINT "Decimal # = " DECIMAL : PRINT
CHAIN "MAIN"
```

Sample output (unless noted otherwise, indicates output is dependent on user input):

```
Base (enter 0 to end): 12              Number: A1B
Decimal # =  1463

Base (enter 0 to end): 0
```

CHDIR Statement

■ Syntax

CHDIR *pathspec*

■ Action

Changes the current default directory for the specified drive

■ Remarks

The *pathspec* is a string expression that identifies the name of the directory that is to be the current directory. The *pathspec* must be a string of fewer than 128 characters, and have the following form:

[[*drive*:]]*directory*[[\ *directory*...]]

The argument *drive*: is an optional drive specification: if it is not given, **CHDIR** changes the directory in the current drive.

CHDIR works exactly like the DOS command **CHDIR**; the syntax in BASIC cannot, however, be shortened to **CD**, as in DOS.

■ See Also

MKDIR, **RMDIR**

■ Examples

```
CHDIR "\HOME\SALES"   'Makes \HOME\SALES the current directory

CHDIR "B:USERS"       'Changes the current directory to USERS
                      'on drive B; it does not, however, change
                      'the default drive to B.
A$ = "C:\JON\WORK"
CHDIR A$              'Changes current directory on drive C to
                      '\JON\WORK; default drive remains unchanged
```

■ Syntax

CHR$(*code*)

■ Action

Returns a one-character string whose ASCII code is *code*

■ Remarks

**CHR$** is commonly used to send a special character to the screen or printer. For instance, you could send the "BELL" character (CHR$(7)) as a preface to an error message, or you could send a form feed (CHR$(12)) to clear a terminal screen and return the cursor to the home position.

CHR$ can also be used to insert a double quote (") in a string expression, as in the example below.

■ See Also

ASC, Appendix A, "ASCII Character Codes"

■ Example

```
Q$ = " My, what lovely eyes you have,  she said."
PRINT Q$
MID$(Q$,1) = CHR$(34)    'Now, insert double quotes in Q$ before
MID$(Q$,32) = CHR$(34)   'My and after second comma.
PRINT Q$
```

Output:

```
 My, what lovely eyes you have,  she said.
"My, what lovely eyes you have," she said.
```

CINT Function

■ Syntax

CINT(*expression*)

■ Action

Converts the numeric expression *expression* to an integer by rounding the fractional part of the expression

■ Remarks

If *expression* is not in the range –32,768 to 32,767, an "Overflow" error message is generated.

CINT differs from the **FIX** and **INT** functions, which return integers by truncating the fractional part. For an example of the differences between these 3 functions, see the example in the **INT** function reference page.

■ See Also

CDBL, CSNG, FIX, INT

■ Example

```
CONVERT = 180/3.141593
INPUT "Angle in radians = ",ANGLE
ANGLE = ANGLE * CONVERT      'Convert radian input to degrees
MIN = ANGLE - INT(ANGLE)     'Get the fractional part
MIN = CINT(MIN * 60)         'Convert fraction to value
                             'between 0 and 60
ANGLE = INT(ANGLE)           'Get whole number part
IF MIN = 60 THEN
        ANGLE = ANGLE + 1    '60 minutes = 1 degree
    MIN = 0
END IF
PRINT "Angle equals" ANGLE "degrees" MIN "minutes"
```

Sample output:

```
Angle in radians = 1.5708
Angle equals 90 degrees 0 minutes
```

■ Syntax

CIRCLE [[**STEP**]] (*x*,*y*),*radius*[,[[*color*]][,[[*start*],[[*end*]][,*aspect*]]]]

■ Action

Draws an ellipse or circle with the specified center and radius

■ Remarks

CIRCLE works only in graphics screen modes. For more information on graphics modes, see the **SCREEN** statement reference pages.

The **STEP** option makes the specified *x* and *y* coordinates relative to the most recent point, instead of absolute, mapped coordinates.

The other arguments to **CIRCLE** are the following:

Argument	Description
x,y	The *x* and *y* coordinates for the center of the circle or ellipse.
radius	The radius of the circle or ellipse in the current coordinate system.
color	The number for the color desired; see the **COLOR** statement reference pages. The default color is the foreground color.
start, end	The *start* and *end* angles in radians for an arc of an ellipse. They may range from -2π to 2π, where $\pi = 3.141593$. If the start or end angle is negative, then **CIRCLE** draws a radius to that point on the ellipse and treats the angle as if it were positive. If you want to draw a radius to angle 0 (in other words, a horizontal line segment to the right), do not give the angle as –0; use a very small nonzero value instead, as in the following:

```
'Draws a pie-shaped quarter-wedge of a circle:
SCREEN 2
CIRCLE (200,100),60,,-.0001,-1.57
```

The start angle can be less than the end angle. If you specify *end* but not *start*, the arc is drawn from 2π to *end*; if you specify *start*, but not *end*, the statement will draw the entire ellipse.

aspect — The aspect ratio, or the ratio of the *y* radius to the *x* radius. When you do not specify *aspect*, **CIRCLE** draws a round circle. The default value for aspect is the value required to draw a round circle in the given screen mode. This value is equal to the following:

4 * (*ypixels*/*xpixels*)/3

where *xpixels* x *ypixels* is the current screen resolution. For example, in **SCREEN 1**, where the resolution is 320 x 200, the default for *aspect* is 4 * (200/320)/3, or 5/6.

If the aspect ratio is less than one, the radius (*radius*) is the *x* radius. If aspect is greater than one, *radius* is equal to the *y* radius.

The last point that **CIRCLE** references, after drawing an ellipse, is the center of the ellipse.

It is not an error to supply coordinates that are outside the screen or viewport.

You can show coordinates as absolutes, as in the syntax shown above, or you can use the **STEP** option to show the position of the center point in relation to the previous point of reference. For example, if the previous point of reference is (10,10), then the following statement causes a circle to be drawn with radius 75 and center offset 10 from the current *x* coordinate and 5 from the current *y* coordinate; that is, the circle has a center (20,15):

```
CIRCLE STEP (10,5), 75
```

■ Example 1

Assume that the last point plotted was (100,50). Then,

```
SCREEN 2
CIRCLE (200,200),50
```

and

```
SCREEN 2
CIRCLE STEP (100,150),50
```

will each draw a circle with center at (200,200) and radius 50. The first statement uses absolute notation; the second uses relative notation.

■ Example 2

The following example draws a circle on the screen, with the top left quarter removed:

```
SCREEN 2
PI = 3.141593
CIRCLE (100,150), 50,, -PI, -PI/2
```

■ Example 3

Since the aspect ratio is less than one, this example draws an ellipse that has a wide horizontal axis and a narrow vertical axis. This also means that 60 is the x radius, so the length of the horizontal axis equals 120, or 2 times 60.

```
SCREEN 2
CIRCLE (160,100), 60,,,, 5/18
```

■ Example 4

The following program draws a torus, or "doughnut-shaped" figure:

```
SCREEN 2
WINDOW (-320,-100)-(320,100)
FOR X = -202 TO 202 STEP 4
    ASPECT = ABS(202/X)*5/12
    IF ASPECT < 1 THEN RADIUS = 20/ASPECT ELSE RADIUS = 20
    Y = 40 * SQR(1 - X^2/40804)
    CIRCLE (X,Y),RADIUS,,,,ASPECT
    CIRCLE (X,-Y),RADIUS,,,,ASPECT
NEXT
```

■ Syntax

CLEAR [[,,*stack*]]

■ Action

The **CLEAR** statement performs the following actions:

- Closes all files
- Clears all **COMMON** variables
- Resets numeric variables and arrays to zero
- Resets the stack and optionally changes its size
- Resets all string variables to null
- Releases all disk buffers

■ Remarks

The *stack* parameter sets aside stack space for BASIC. The default for *stack* is 768 bytes; the minimum stack size is 512 bytes. You may want to increase the stack size if your program has deeply nested subroutines (subroutines calling other subroutines, which in turn call still other subroutines, and so on), or if your program calls subprograms or assembly-language routines and passes a large number of arguments.

Since the stack is also used by the system interrupts, a safe value for *stack* is 512 plus the amount needed by BASIC.

Clearing the stack destroys the return addresses placed on the stack during the execution of a control structure such as **GOSUB...RETURN**, **FOR...NEXT**, or **WHILE...WEND**, thus rendering a normal exit from the structure impossible. The only alternatives for exiting a structure after execution of a **CLEAR** within the structure are **GOTO** or **RUN**{ *linenumber* | *linelabel* }.

■ Compiler/Interpreter Differences

This statement may require modification of interpreted BASIC programs when they are used with the compiler. In interpreted programs, any functions declared with **DEF FN** statements, or any data types declared with **DEF***type* statements are lost after **CLEAR**; in compiled programs, this information is not lost. These declarations are fixed at compile time and cannot vary.

Note also that the default stack size of 768 bytes is different from the default of 512 bytes in the interpreter.

■ Examples

The following statement sets all numeric variables to zero, all string variables to null, and closes all files opened by BASIC:

```
CLEAR
```

The following statement clears the data and sets the stack to 2,000 bytes:

```
CLEAR ,,2000
```

■ Syntax

CLOSE[[[#] *filenumber* [,[#] *filenumber...*]]]

■ Action

Concludes I/O to a file or device

■ Remarks

The **CLOSE** statement complements the **OPEN** statement.

The *filenumber* is the number under which the file was opened. A **CLOSE** with no arguments closes all open files and devices.

The association of a particular file and a file number terminates upon execution of a **CLOSE** statement. You may then reopen the file using the same or a different file number. Once you close a file, you may use that file's number for any unopened file.

A **CLOSE** for a file or device that was opened for sequential output writes the final buffer of output to that file or device.

The **CLEAR**, **END**, **RESET**, **RUN**, and **SYSTEM** statements always close all files automatically.

■ Example

```
CLOSE #1,#2
```

■ Syntax

CLS [[{0 | 1 | 2}]]

■ Action

Clears the screen

■ Remarks

There are several ways to use **CLS**, as described in the following list:

Statement	Effect
CLS 0	Clears the screen of all text and graphics.
CLS 1	Clears only the graphics viewport.
CLS 2	Clears only the text window.
CLS	If the graphics viewport is active, then **CLS** with no argument clears only the viewport. If the graphics viewport is inactive, then **CLS** clears the text window.

CLS also returns the cursor to the home position in the top left corner of the screen.

■ Example

```
SCREEN 1                    'Clears screen to violet
COLOR 13,1
CLS
```

■ Syntax

```
COLOR [foreground][,[background][,border]]    'Screen mode 0
COLOR [background][,[palette]]                'Screen mode 1
COLOR [foreground][,[background]]             'Screen modes 7–10
```

■ Action

Selects display colors

■ Remarks

In general, **COLOR** allows you to select the foreground and background colors for the display. In **SCREEN 0** a border color can also be selected. In **SCREEN 1** no foreground color can be selected, but one of two four-color palettes can be selected for use with graphics statements. The different syntaxes and effects that apply to the various screen modes are described below:

Mode	Effect
SCREEN 0	Modifies the current default text foreground and background colors, and the screen border. The *foreground* color must be an integer expression in the range 0–31. It is used to determine the "foreground" color in text mode, which is the default color of text. Sixteen colors can be selected with the integers 0–15. A blinking version of each color can be selected by adding 16 to the color number; for example, a blinking color 7 is equal to 7 + 16, or 23. Thus, the legal integer range for *foreground* is 0–31. The *background* color must be an integer expresssion in the range 0–7, and is the color of the background for each text character. Blinking colors are not permitted. The *border* color is an integer expression in the range 0–15, and is the color used when drawing the screen border. Blinking colors are not permitted. If no arguments are provided to **COLOR**, then the default color for *background* and *border* is black (color

0), and for *foreground*, is as described in the **SCREEN** statement reference pages.

SCREEN 1

In mode 1, the **COLOR** statement has a unique syntax that includes a *palette* argument, which is an odd or even integer expression. This argument determines the set of display colors to use when displaying particular color numbers.

For hardware configurations that do not have an IBM Enhanced Graphics Adapter (EGA), the default color settings for the *palette* parameter are equivalent to the following:

```
COLOR ,0        'Same as the next three PALETTE
                'statements
                '1 = green, 2 = red, 3 = yellow

COLOR ,1        'Same as the next three PALETTE
                'statements
                '1 = cyan, 2 = magenta, 3 = hi.
                'intens. white
```

With the EGA, the default color settings for the *palette* parameter are equivalent to the following:

```
COLOR ,0        'Same as the next three PALETTE
                'statements
PALETTE 1,2     'Attribute 1 = color 3 (green)
PALETTE 2,4     'Attribute 2 = color 5 (red)
PALETTE 3,6     'Attribute 3 = color 6 (brown)

COLOR ,1        'Same as the next three PALETTE
                'statements
PALETTE 1,3     'Attribute 1 = color 3 (cyan)
PALETTE 2,5     'Attribute 2 = color 5 (magenta)
PALETTE 3,7     'Attribute 3 = color 15 (white)
```

Note that a **COLOR** statement will override previous **PALETTE** statements.

SCREEN 2

No effect. An "Illegal function call" message results if **COLOR** is used in this mode.

SCREEN 7–SCREEN 10

In these modes, no *border* color can be specified. The graphics background is given by the *background* color number, which must be in the valid range of color numbers appropriate to the screen mode. See the

SCREEN statement reference pages for more details. The *foreground* color argument is the default line drawing color.

Arguments outside valid numeric ranges result in "Illegal function call" errors.

The foreground color may be the same as the background color, making displayed characters invisible. The default background color is black, or color number 0, for all display hardware configurations and all screen modes.

With the Enhanced Graphics Adapter (EGA) installed, the **PALETTE** statement gives you flexibility in assigning different display colors to the actual color-number ranges for the *foreground*, *background*, and *border* colors discussed above. See the **PALETTE** statement reference pages for more details.

■ See Also

CIRCLE, DRAW, LINE, PALETTE, PAINT, PRESET, PSET, SCREEN

■ Examples

The following series of examples show **COLOR** statements and their effects in the various screen modes:

```
SCREEN 0
COLOR  1, 2, 3  'foreground=1, background=2, border=3

SCREEN 1
COLOR  1,0    'foreground=1, even palette number
COLOR  2,1    'foreground=2, odd palette number

SCREEN 7
COLOR  3,5    'foreground=3, background=5

SCREEN 8
COLOR  6,7    'foreground=6, background=7

SCREEN 9
COLOR  1,2    'foreground=1, background=2
```

■ Syntax

COM(*n*) ON
COM(*n*) OFF
COM(*n*) STOP

■ Action

Enables, disables, or inhibits event trapping of communications activity on the specified port

■ Remarks

The parameter *n* is the number of the communications port; *n* can be either 1 or 2.

The **COM ON** statement enables communications event trapping by an **ON COM** statement. If you specify a nonzero line number in the **ON COM** statement while trapping is enabled, the program checks between every statement (if the Checking Between Statements or **/v** option is set), or between every line (if the Event Trapping or **/w** option is set), to see if a character has been received on the communications channel since the last check. If it has, it executes the **ON COM** statement.

COM OFF disables communications event trapping. If an event takes place, it is not remembered.

COM STOP inhibits communications event trapping; that is, if an event occurs, it is not immediately trapped, but it is remembered. If there is a subsequent **COM ON** statement, the remembered event will be successfully trapped.

■ Compiler/Interpreter Differences

See the **ON COM** statement reference pages.

■ Example

This enables error trapping of communications activity on COM1:

```
10 COM(1) ON
```

COMMAND$ Function

■ Syntax

COMMAND$

■ Action

Returns the command line used to invoke the program

■ Remarks

COMMAND$** works only in the compiler. This function returns the complete command line entered after your QuickBASIC program name, including any optional parameters, after first stripping all leading blanks from the command line and converting all letters to uppercase (capital letters). **COMMAND$ can be used in standalone executable files, or, if you are using in-memory compilation, when QuickBASIC is invoked with the **/cmd** option on the **qb** command line. (See Section 4.1, "Starting QuickBASIC.")

■ Example

The ARGSPLIT subprogram in the following example breaks the command line into separate arguments and stores them in an array, ARG$. Each argument is separated from adjoining arguments by one or more blanks on the command line.

```
DEFINT A-Z
CLINE$ = COMMAND$
LENGTH = LEN(CLINE$)
MAX = (LENGTH/2) + 1            'Maximum number of arguments possible
DIM ARG$(MAX)                   'Array ARG$ is dynamic, since
GOSUB ARGSPLIT                  'dimensioned with variable

PRINT "NUMBER OF ARGUMENTS = " NUM+1
PRINT "ARGUMENTS ARE:"
FOR I = 0 TO NUM : PRINT ARG$(I) : NEXT
END
```

```
ARGSPLIT:                          'Subroutine to split command line
    TRUE = -1 : FALSE = 0          'Define logical flags
    I = 1 : NUM = 0 : INWORD = TRUE
    WHILE I <= LENGTH
        CH$ = MID$(CLINE$,I,1)
        IF CH$ <> " " THEN
            IF NOT INWORD THEN INWORD = TRUE
            ARG$(NUM) = ARG$(NUM) + CH$
        ELSEIF INWORD THEN
            NUM = NUM + 1
            INWORD = FALSE
        END IF
        I = I + 1
    WEND
RETURN
```

Sample command line and output for standalone executable file (assumes program name is `arg.bas`):

```
arg one  two   three    four     five      six
```

Output:

```
Number of arguments =  6
Arguments are:
ONE
TWO
THREE
FOUR
FIVE
SIX
```

If compiled in memory, the command line for invoking QuickBASIC to produce the same output would look like this:

```
qb /cmd one  two   three    four     five      six
```

■ Syntax

COMMON [[**SHARED**]] [[/*blockname*/]] *variablelist*

■ Action

Passes variables to a chained program or a called subprogram

■ Remarks

SHARED is an optional attribute. Use **SHARED** to pass variables from a main program to a subprogram within the same module; this way, you don't need the **SHARED** statement, or another **COMMON** statement, within the subprogram. You do need the **SHARED** attribute with **COMMON** when the subprogram being called is in a user library.

The **COMMON** statement takes the following arguments:

Argument	Description
blockname	Any valid BASIC identifier up to 31 characters long. Use *blockname* to group variables when subprograms need to share only specific groups of variables from the main program; this is known as a "named" **COMMON** block (as opposed to a "blank" **COMMON** block when *blockname* is omitted). Items in a named **COMMON** block are not preserved across a chain to a new program.
variablelist	A list of variables and arrays used by subprograms or chained-to programs. The same variable cannot appear in more than one **COMMON** statement. Static arrays in **COMMON** statements have the form *arrayname*( ), while dynamic arrays have the form *arrayname*(*num*), where *num* is an integer constant indicating the number of dimensions in the array.

The **COMMON** statement must appear in a program before any executable statements. All statements are executable, except the following:

- **COMMON**
- **DEF***type*
- **DIM** (for static arrays)
- **OPTION BASE**
- **REM**
- All metacommands

If an array has been declared static in a **COMMON** statement with the form *arrayname*( ), then it must be dimensioned with integer-constant subscripts in a preceding **DIM** statement. If an array has been declared dynamic with the form *arrayname*(*num*), then the actual array elements are not allocated in the **COMMON** block; only an array descriptor is allocated. The dynamic array must then be allocated later in a **DIM** or **REDIM** statement.

Variable order is significant in **COMMON** statements. In the following fragment, it is the order of the variables in the two COMMON statements that is significant, not the names of the variables themselves:

```
'Main program:
COMMON A, D, E
A = 5 : D = 8 : E = 10
.
.
.

'Called subprogram:
COMMON A, E, D          'A = 5, E = 8, D = 10
.
.
.
```

When a QuickBASIC program shares named **COMMON** blocks with a routine in the user library, the calling program may not redefine the named **COMMON** block to a larger size. However, resizing of blank **COMMON** blocks is allowed.

To ensure that programs can share common areas, place **COMMON** declarations in a single "include" file and use the **$INCLUDE** metacommand in each program. (See Section 10.3, "Using Metacommands," for a discussion of **$INCLUDE**.)

If you are compiling outside the QuickBASIC environment (that is, you are not compiling into memory), note that when you use **COMMON** with **CHAIN**, you must use the **BRUN20.EXE** module. This module is used when you compile your program without the **/o** option.

Using COMMON with CHAIN

When compiled with QuickBASIC, both the chaining program and the chained-to program require a **COMMON** statement. The *order* of variables must be the same for all **COMMON** statements communicating between chaining and chained-to programs. If the *size* of the common region in the chained-to program is smaller than the common region in the chaining program, the extra **COMMON** variables in the chaining program are ignored. If the size of the common region in the chained-to program is larger, any additional **COMMON** numeric variables are initialized to zeros, while any additional string variables are initialized to null strings.

Arrays declared as static in the chaining program must be declared as static in the chained-to program; arrays declared as dynamic in the chaining program must be declared as dynamic in the chained-to program.

■ **See Also**

CALL, CHAIN, SUB...END SUB

■ Example 1

This program fragment shows how to use the $INCLUDE metacommand to share a file containing COMMON statements among programs:

```
REM This file is menu.bas
REM $INCLUDE:'COMDEF.BAS'
.
.
.
CHAIN "PROG1"
END


REM This file is prog1.bas
REM $INCLUDE:'COMDEF.BAS'
.
.
.
END


REM This file is comdef.bas
DIM A(100),B$(200)
COMMON I,J,K,A()
COMMON A$,B$(),X,Y,Z
REM End comdef.bas
```

■ Example 2

Dynamic arrays must be dimensioned after the **COMMON** statement. The chaining program in the following example contains the first allocation of array X:

```
COMMON X(2)    '2 indicates 2 dimensions
DIM X(N,M)     'first allocation
```

The chained-to program reallocates array X, erasing its original contents:

```
COMMON X(2)
REDIM X(O,P)   'reallocation
```

■ Example 3

The following example, which calculates the volume and density of a 3-dimensional rectangle, uses named **COMMON** blocks to share different sets of data with the two called subprograms. Subprogram VOLUME needs to share only the variables representing the lengths of the sides (in **COMMON** block SIDES). Subprogram DENSITY also needs variables representing the weight (in **COMMON** block WEIGHT).

```
'Main program:
DIM S(3)
COMMON /SIDES/ S()
COMMON /WEIGHT/ c

C=52
S(1)=3:S(2)=3:S(3)=6
CALL VOLUME
CALL DENSITY
END

'Subprogram VOLUME in user library:
DIM s(3)
COMMON SHARED /SIDES/ s()

SUB VOLUME STATIC
    VOL=S(1)*S(2)*S(3)
    PRINT "The volume is " VOL
END SUB


'Subprogram DENSITY in user library:
DIM s(3)
COMMON SHARED /SIDES/ S()
COMMON SHARED /WEIGHT/ W

SUB DENSITY STATIC
    VOL=S(1)*S(2)*S(3)
    DENS=W/VOl
    PRINT "The density is " DENS
END SUB
```

■ Syntax

COS(x)

■ Action

Returns the cosine of x, where x is in radians

■ Remarks

The calculation of **COS** is performed in single precision. **COS** is performed in double precision if the argument x is a double-precision value.

You can convert an angle measurement from degrees to radians by multiplying the degrees by $\pi/180$, where $\pi = 3.141593$.

■ See Also

SIN, TAN

■ Example

The following program plots the graph of the polar equation $r = n\theta$ for values of n from 0.1–1.1. This program uses the conversion factors $x = \cos(\theta)$ and $y = \sin(\theta)$ to change polar coordinates to Cartesian x,y coordinates. The figure plotted is sometimes known as the "Spiral of Archimedes."

```
PI = 3.141593
SCREEN 1 : COLOR 7     'Grey background
WINDOW (-4,-6)-(8,2)   'Define window large enough for biggest spiral
LINE (0,0)-(2.2*PI,0),1       'Draw line from origin to the right
FOR N = 1.1 TO .1 STEP -.1    'Draw ten spirals
    PSET (0,0)                'Plot starting point
    FOR ANGLE = 0 TO 2*PI STEP .04
         R = N * ANGLE         'Polar equation for spiral
         X = R * COS(ANGLE)    'Convert polar coordinates to
         Y = R * SIN(ANGLE)    'Cartesian coordinates
         LINE -(X,Y),1         'Draw line from previous point to
    NEXT                      'new point
NEXT
```

CSNG Function

■ Syntax

CSNG(*numeric-expression*)

■ Action

Converts *numeric-expression* to a single-precision number

■ Remarks

The **CSNG** function has the same effect as assigning *numeric-expression* to a single-precision variable.

■ See Also

CDBL, **CINT**

■ Example

```
A# = 975.3421115#
PRINT A#, CSNG(A#)
```

Output:

```
 975.3421115   975.3421
```

■ Syntax

CSRLIN

■ Action

Obtains the current line position of the cursor

■ Remarks

To return the current column position, use the **POS** function.

■ See Also

LOCATE, POS

■ Example

```
Y = CSRLIN          'Record current line
X = POS(0)          'Record current column
LOCATE 24,1
PRINT "HELLO"
LOCATE X,Y          'Restore position to old line and column
```

CVI, CVS, CVD Functions

■ Syntax

CVI(*2-byte-string*)
CVS(*4-byte-string*)
CVD(*8-byte-string*)

■ Action

Converts string values to numeric values

■ Remarks

Numeric values are stored in random disk files as strings; to read these values from the random disk file, they must be converted from strings back into numbers. **CVI** converts a *2-byte-string* created with **MKI$** back to an integer. **CVS** converts a *4-byte-string* created with **MKS$** back to a single-precision number. **CVD** converts an *8-byte-string* created with **MKD$** back to a double-precision number.

The **CVI**, **CVS**, and **CVD** functions do not change the bytes of the actual data. They change only the way BASIC interprets those bytes.

■ See Also

MKI$**, **MKS$, **MKD$**

■ Example

```
'This program illustrates the use of MKS$ and CVS
'with random-access files:
FORMAT$ = "$$######.##"
OPEN "ACCOUNT.INF" FOR RANDOM AS #2 LEN = 32
FIELD #2, 25 AS NAME$, 7 AS CHECK$
.
.
.
MAIN:
    INPUT "Enter account # to update: ",REC%
    GET #2, REC%          'Get the record
    PRINT "This is the account for " NAME$
    INPUT "Is this the account you wanted"; R$
    IF R$ <> "y" THEN GOTO MAIN
    'Convert string to single-precision number:
    CHECKAMT!  = CVS(CHECK$)
    PRINT "The opening balance for this account is";
    PRINT USING FORMAT$; CHECKAMT!
    PRINT "Enter the checks and cash withdrawals for this"
    PRINT "account below. Enter 0 when finished."
    CHECKOUT! = 99
    WHILE CHECKOUT!
        INPUT CHECKOUT!
        CHECKAMT! = CHECKAMT! - CHECKOUT!
    WEND
    PRINT "Enter the deposits for this account below."
    PRINT "Enter 0 when finished."
    CHECKIN! = 99
    WHILE CHECKIN!
        INPUT CHECKIN!
        CHECKAMT! = CHECKAMT! + CHECKIN!
    WEND
    PRINT "The closing balance for this account is";
    PRINT USING FORMAT$; CHECKAMT!
    'Convert single-precision number to string:
    LSET CHECK$ = MKS$(CHECKAMT!)
    PUT #2, REC%          'Store the record
    INPUT "Update another"; R$
    IF R$ = "y" THEN GOTO MAIN ELSE CLOSE #2
END
```

DATA Statement

■ Syntax

DATA *constant1* [[,*constant2*]]...

■ Action

Stores the numeric and string constants used by the program's **READ** statement(s)

■ Remarks

DATA statements are nonexecutable. You may place them anywhere in the program.

A **DATA** statement may contain as many constants as will fit on a line. These constants should be separated by commas.

You may use any number of **DATA** statements in a program. **READ** statements read **DATA** statements in the order in which they appear in the source file. You can think of the items in several **DATA** statements as one continuous list of items, regardless of how many items are in a statement or where the statement is placed in the program.

The constants in a **DATA** statement may be in any format: string, fixed-point, floating-point, or integer. However, the variable type (numeric or string) that you give in the **READ** statement must agree with the corresponding constant in the **DATA** statement. In addition, no numeric expressions are allowed in the list.

You need not put double quotation marks around string constants in **DATA** statements unless they contain commas, colons, or significant leading or trailing spaces.

It is permissible to have a null data item in a data list. If the null item is read into a numeric variable, the variable has the value 0; if the null item is read into a string variable, the variable has the null string value. (See the second example.)

A **READ** statement can reread the data in one or more of your program's **DATA** statements if you use the **RESTORE** statement.

■ See Also

DATA, READ, RESTORE

■ Example 1

```
C$ = DATE$
FOR I% = 1 TO VAL(C$)    'VAL(C$) is a number between 1 and 12
    READ MONTH$
NEXT
DATA January, February, March, April, May, June, July
DATA August, September, October, November, December
DAY$ = MID$(C$,4,2)
IF LEFT$(DAY$,1) = "0" THEN DAY$ = RIGHT$(DAY$,1)
YEAR$ = RIGHT$(C$,4)
PRINT "Today is " MONTH$ " " DAY$ ", " YEAR$
```

Sample output:

```
Today is February 20, 1986
```

■ Example 2

```
'Shows the effect of a null data item in a DATA list:
DATA abc,,def
DATA 1,,2
READ A$, B$, C$          'B$ = ""
PRINT A$, B$, C$
PRINT
READ A, B, C             'B = 0
PRINT A, B, C
```

Output:

```
abc                      def
 1           0            2
```

DATE$ Function

■ Syntax

DATE$

■ Action

Retrieves the current date

■ Remarks

The **DATE$** function returns a ten-character string in the form *mm-dd-yyyy*, where *mm* is the month (01–12), *dd* is the day (01–31), and *yyyy* is the year (1980–2099).

■ See Also

DATE$ Statement

■ Example

Note that the DATE$ function in the following example prints a zero in front of the month; it also prints the answer with the month, day and year separated by hyphens (-), even though they are entered in the DATE$ statement with slashes (/).

```
PRINT DATE$
DATE$ = "5/21/86"
PRINT DATE$
```

Sample output:

```
05-20-1985
05-21-1986
```

■ Syntax

DATE$ = *string-expression*

■ Action

Sets the current date

■ Remarks

This statement complements the **DATE$** function, which retrieves the current date. The *string-expression* must be a string in one of the following forms:

mm-dd-yy
mm-dd-yyyy
mm/dd/yy
mm/dd/yyyy

■ Example

The following program sets the current date:

```
PRINT "Enter the date below (default year is 1986)."
INPUT "    Month:    ",MONTH$
INPUT "    Date:     ",DAY$
INPUT "    Year:     ",YEAR$
IF YEAR$ = "" THEN YEAR$ = "86"
DATE$ = MONTH$ + "/" + DAY$ + "/" + YEAR$
```

DEF FN Statement

■ Syntax 1

DEF FN*name*[[(*parameterlist*)]]= *functiondefinition*

■ Syntax 2

DEF FN*name*[[(*parameterlist*)]]
.
.
.
FN*name* = *expression*
.
.
.
[[**EXIT DEF**]]
.
.
.
END DEF

■ Action

Defines and names a function

■ Remarks

The **DEF FN** statement takes the following arguments:

Argument	Description
name	A legal variable name. This name, preceded by **FN**, becomes the name of the function.
parameterlist	A list of variable names, separated by commas. When the function is called, it assigns the value of each argument to every occurrence of its corresponding parameter. Function arguments are passed by value.
	The *functiondefinition* (Syntax 1) is an expression that performs the operation of the function. It is limited to one logical line. Variable names that appear in this expression are local to the expression and serve to define the function; they do not affect program variables that have the same name. A variable name used in a function definition may or may not appear in the parameter list. If it does, the value of the parameter is supplied when the function is called. Otherwise, the current value of the variable is used.
	In the multiline version of **DEF FN** (Syntax 2), *expression* defines the value returned by the function. A multiline function ends with an **END DEF** statement.

The optional **EXIT DEF** statement exits the function and returns to the calling program, but does not define the end of the function.

Parameter variables, or values appearing in the function call, are called by value.

DEF FN can define either numeric or string functions. **DEF FN** returns a string value if *name* is a string variable name, and a numeric value if *name* is a numeric variable name. If the type (string or numeric) of *functiondefinition* or *expression* does not match the type of **DEF FN***name,* then a "Type mismatch" error occurs.

If the function is numeric, **DEF FN***name* gives the value it returns to the calling statement the precision specified by *name*; i.e, if *name* specifies a double-precision variable, then the value of **DEF FN***name* is in double precision, regardless of the precision of *functiondefinition* or *expression*.
The following rules apply to QuickBASIC multiline functions:

- Your program must define a function with a **DEF FN** statement before the function can be called. If your program calls a function before it is defined, a "Function not defined" error occurs.

- User-defined function definitions cannot appear inside other multiline function definitions, nor can they appear inside the following statement blocks: **IF...THEN...ELSE**, **FOR...NEXT**, **SUB...END SUB**, or **WHILE...WEND**.

- Programs cannot contain recursive function definitions; that is, a function cannot be defined in terms of itself.

- The **RETURN** statement is *not* equivalent to **EXIT DEF**. **RETURN** is used to return from a **GOSUB** or an event trap, and will cause a severe and nonrecoverable error when used to exit a multiline function. If you use **RETURN** to exit a multiline function and you compile your program with the Debug, On Error, Between Line Checking, Resume Next, or Event Trapping options, you will either get a "RETURN without GOSUB" error message or unpredictable results.

- Variables that are not in the *parameterlist* are global; that is, their values are shared with the calling program. To keep a variable value local to a function definition, declare it in a **STATIC** statement.

If you are not careful when constructing multiline functions, you may get unexpected side effects. Most of these occur because, as an optimizing compiler, the QuickBASIC compiler may rearrange arithmetic expressions for greater efficiency. The following program shows the various possibilities for an expression involving a reference to a multiline function:

```
DEF FNA
  I = 10
  FNA = 1
END DEF
I = 1 : PRINT FNA + I + I
```

The expression in the PRINT statement in the above example gives the result 21. The compiler evaluates the expression as follows, giving FNA the value 1, and I the value 10:

```
FNA + (I + I)
```

However, in some circumstances the compiler will reorder the expression, causing the program to produce a different result. For example, the compiler might evaluate the expression in the last line as follows, giving both I and FNA the value 1, and returning 3 as the result:

```
(I + I) + FNA
```

By assigning the result of FNA to a temporary variable and using the new variable, this ambiguity is avoided. For example:

```
I = 1 : X = FNA : PRINT X + I + I
```

Another way to avoid global side effects is to localize variable values with the **STATIC** statement, for example:

```
DEF FNA
  STATIC I
  I=10          'This value of I will be local to the function
  FNA = 1
END DEF
'Now, the value for I in the PRINT statement will be 1
I = 1 : PRINT FNA + I + I       'Output is 3
```

See the **STATIC** statement reference pages for more information.

Nesting run-time features can also cause side effects when using multiline functions. For example, in the following function B and C are printed using the field description from B$ instead of from A$:

```
A$ = "+##.##" : B$ = "-##.##"
DEF FNX
    z=99.27
    PRINT USING B$;Z
END DEF
B=12.44 : C=25.99
PRINT USING A$;FNX,B,C
```

In I/O operations, it is best to avoid calling multiline functions that also perform I/O because of such unexpected side effects.

■ Example

The following example contains a function definition that converts an angle measure in degrees, minutes, and seconds to an angle measure in radians. (An angle must be given in radians for the trigonometric functions of BASIC to return a meaningful answer.)

```
DEF FNDEGRAD(D,M,S)
   STATIC PI
   PI = 3.14159263
   D = D + M/60 + S/3600
   FNDEGRAD = D * (PI/180)
END DEF

PRINT "Enter the angle (degrees, minutes, seconds)."
PRINT "Enter 0 for degrees to end." : PRINT
NEWVAL:
    INPUT ;"SIN(",DEG
    IF DEG = 0 THEN PRINT ")" : END
    PRINT CHR$(248) " ";
    INPUT ;"",MIN : PRINT "' ";
    INPUT ;"",SEC : PRINT CHR$(34);
    RAD = FNDEGRAD(DEG,MIN,SEC)
    PRINT ") ="  SIN(RAD)
    GOTO NEWVAL
```

Sample output:

```
Enter the angle (degrees, minutes, seconds).
Enter 0 for degrees to end.

SIN(45° 10' 10") = .7091949
SIN(45° 10' 20") = .7092291
SIN(0)
```

■ Syntax

DEF SEG [[=*address*]]

■ Action

Assigns the current segment address to be referenced by a subsequent **BLOAD**, **BSAVE**, **CALL ABSOLUTE**, or **POKE** statement, or by a **PEEK** function

■ Remarks

The *address* is a numeric expression returning an unsigned integer in the range 0–65,535. Entry of any value outside this range will result in an "Illegal function call" error. The previous value is retained when this happens. The *address* is saved for use as the segment required by **BLOAD**, **BSAVE**, **POKE**, or **PEEK**.

If you omit the *address* option, the BASIC data segment is used by default.

You must separate **DEF** and **SEG** with a space. Otherwise, BASIC interprets a statement, such as the following, to mean "assign the value 100 to the variable DEFSEG":

```
DEFSEG=100
```

■ Compiler/Interpreter Differences

In the compiler, the **CALL** and **CALLS** statements do not reference **DEF SEG**.

■ Example

```
DEF SEG=&HB800   'Seg segment at B800 Hex
DEF SEG          'Restore segment to BASIC data segment
```

■ Syntax

DEFINT *letter-range* [,*letter-range...*]
DEFSNG *letter-range* [,*letter-range...*]
DEFDBL *letter-range* [,*letter-range...*]
DEFSTR *letter-range* [,*letter-range...*]

■ Action

Declares variables of integer, single-precision, double-precision, or string type

■ Remarks

The *letter-range* takes the following form:

letter1[-*letter2*]

where *letter1* and *letter2* are any of the uppercase or lowercase letters of the English alphabet. Any variable names beginning with the letters specified in *letter-range* are the type of variable specified by the last three letters of the statement: either **INT** (integer), **SNG** (single precision), **DBL** (double precision), or **STR** (string). For example, the following statement declares all variables beginning with the letter A as string variables:

```
DEFSTR A
```

In the next example, all variables beginning with the letters K-M, X-Z are designated integer variables:

```
DEFINT K-M, X-Z
```

A type-declaration character such as %, !, #, or $ always takes precedence over a **DEF*type*** statement.

Note

I!, I#, I$, and I% are all separate and distinct variables; therefore, each one can hold a different value.

■ Compiler/Interpreter Differences

The interpreter and the compiler process **DEF*type*** statements somewhat differently. The interpreter must scan a statement each time before it executes it. If the statement contains a variable that does not have an explicit type (signified by !, #, $, or %), the interpreter determines the current default type and uses it. In the example below, when the interpreter encounters line 20, it determines that the current default type for variables beginning with I is integer. Line 30 then changes this default type to single precision and loops back to line 20. The interpreter must rescan line 20 in order to execute it and this time IFLAG becomes a single-precision variable:

```
10  DEFINT I
20  PRINT IFLAG
30  DEFSNG I
40  GOTO 20
```

In contrast, the compiler scans the text only once. Therefore, once a variable occurs in a program line, its type cannot be changed. The compiler, unlike the interpreter, does not allow you to circumvent a **DEF*type*** statement by directing program flow around it.

You can see these differences in the the example program output.

■ Example

The following program gives different results when you run it with the interpreter and with the compiler. The interpreter assigns variable types each time it scans a statement during program execution, so it allows the program to redeclare the variable type inside the **FOR...NEXT** loop. On the other hand, the compiler statically scans **DEF*type*** statements, assigning variable types at compile time, so line 160 applies only to variables beginning with T in program lines after line 160.

```
100  TEST% = 1           'Integer type
110  TEST! = 10          'Single-precision type
120  DEFINT T
130  FOR I = 1 TO 3
140     PRINT TEST
150     TEST = TEST + 20
160     DEFSNG T
170  NEXT
180  PRINT
190  TEST = TEST + 100
200  PRINT "TEST = ";TEST
210  PRINT "TEST% =";TEST%
220  PRINT "TEST! =";TEST!
```

Interpreter output:

```
 1
 10
 30

TEST =  150
TEST% = 21
TEST! = 150
```

Compiler output:

```
 1
 21
 41

TEST =  110
TEST% = 61
TEST! = 110
```

■ Syntax

DIM [[**SHARED**]] *variable*(*subscripts*)[[, *variable*(*subscripts*)]]...

■ Action

Specifies the maximum values for array-variable subscripts and allocates storage accordingly

■ Remarks

The optional **SHARED** attribute allows a main program to share arrays and simple variables with all the subprograms in a compiland; this differs from the **SHARED** statement, which affects only the variables within a single program.

If you use an array in your program without including the array in a **DIM** statement, the maximum value of the array's subscript(s) is assumed to be 10. If you use a subscript that is greater than the maximum specified, a "Subscript out of range" error occurs. The minimum value for a subscript is 0, unless you specify the value 1 in a preceding **OPTION BASE** statement.

The **DIM** statement sets all the elements of the specified numerical arrays to an initial value of zero and all the elements of string arrays to null strings.

The maximum number of dimensions allowed in a **DIM** statement is 63.

If you try to dimension an array variable with a **DIM** statement, but the array dimension has already been assigned the default value of 10, an "Array already dimensioned" error results. Therefore, it is good programming practice to put the required **DIM** statements at the beginning of a program, outside of any processing loops.

■ Compiler/Interpreter Differences

The compiler scans the **DIM** statement instead of executing it. That is, **DIM** takes effect when it is encountered at compile time and remains in effect until the end of the program. It cannot be re-executed at run-time.

The exception to the above is when the array is a dynamic array. In this case, a **DIM** statement is considered an executable statement.

■ See Also

OPTION BASE, REDIM, SHARED

■ Example

```
DIM A(20)
FOR I=0 TO 20
   READ A(I)
NEXT I
.
.
.
```

■ Syntax

DRAW *string-expression*

■ Action

Draws an object defined by *string-expression*

■ Remarks

The **DRAW** statement combines many of the capabilities of the other graphics statements into a graphics macro language, as described below under "Cursor Movement Commands" and "Angle, Color, and Scale Factor Commands." This macro language defines a set of characteristics that can be used to describe an image. These characteristics include motion (up, down, left, right), color, rotation angle, and scale factor.

The *string-expression* consists of these macro commands, separated from one another by one or more spaces.

Cursor Movement Commands

The following prefix commands can precede any of the movement commands:

Prefix	Action
B	Move, but do not plot any points.
N	Move, but return to original position when done.

The following commands specify movement in units. The default unit size is one point; this unit size can be modified by the **S** command, which sets the scale factor (**S** is described below, under "Angle, Color, and Scale Factor Commands"). If no argument is supplied, the cursor is moved one unit.

Each of the movement commands initiates movement from the current graphics position, this usually being the coordinate of the last graphics point plotted with another graphics macro-language command. The current position defaults to the center of the screen when a program is run.

Movement	Action
U [*n*]	Move up *n* units.
D [*n*]	Move down *n* units.
L [*n*]	Move left *n* units.
R [*n*]	Move right *n* units.
E [*n*]	Move diagonally up and right *n* units.
F [*n*]	Move diagonally down and right *n* units.
G [*n*]	Move diagonally down and left *n* units.
H [*n*]	Move diagonally up and left *n* units.
M *x,y*	Move absolute or relative. If *x* is preceded by a plus (+) or minus (–), the movement is relative to the current point; that is, *x* and *y* are added to (or subtracted from) the coordinates of the current graphics position and connected with that position by a line. If no sign precedes *x*, the movement is absolute; that is, a line is drawn from the current cursor position to the point with coordinates *x,y*.

Angle, Color, and Scale Factor Commands

Command	Action
A *n*	Set angle *n*. The value of *n* may range from 0–3, where 0 is 0°, 1 is 90°, 2 is 180°, and 3 is 270°. Figures rotated 90° or 270° are scaled so they will appear the same size as with 0° or 180° on a monitor screen with the standard aspect ratio of 4/3.
TA *n*	Turn an angle of *n* degrees; *n* must be in the range –360 to 360. If *n* is positive, rotation is counter-clockwise; if *n* is negative, rotation is clockwise. The following example draws spokes:

```
FOR D=0 TO 360  STEP 10
  DRAW "TA="+VARPTR$(D)+"NU50"
NEXT D
```

C *n*

Set color *n*. See the reference pages for **COLOR**, **PALETTE**, and **SCREEN** for discussions of valid colors, numbers, and attributes.

S *n*

Set scale factor *n*, which may range from 1–255. The scale factor multiplied by the distances given with **U**, **D**, **L**, **R**, or relative **M** commands gives the actual distance traveled.

P *paintcolor, bordercolor*

The *paintcolor* is the paint color for a figure's interior, while *bordercolor* is the paint color for the figure's border. "Tile" painting is not supported in **DRAW**.

Substring Command

X *string-expression*

Execute substring. This powerful command allows you to execute a second substring from a string. You can have one string expression execute another, which executes a third, and so on.

Numeric arguments can be constants like 123 or variable names.

The compiler requires the "X" + VARPTR$ (*string-expression*) form of this command. (See "Compiler/Interpreter Differences" below.)

■ Compiler/Interpreter Differences

The **DRAW** statement requires modification of interpreted BASIC programs when used with the compiler. Specifically, the compiler requires the **VARPTR$** form for variables. Statements such as

```
DRAW "XA$"
```

and

```
DRAW "TA = ANGLE"
```

(where A$ and ANGLE are variables) should be changed to

```
DRAW "X" + VARPTR$(A$)
```

and

```
DRAW "TA =" + VARPTR$(ANGLE)
```

in the compiler.

The compiler does not support the

"**X***string-expression*"

command. However, you can execute a substring by appending the character form of the address to X. For example, the following two statements are equivalent. The first statement works with the interpreter and the compiler, while the second works only with the interpreter.

```
DRAW "X" + VARPTR$(A$)
DRAW "XA$"
```

■ Example 1

The following program draws a triangle in magenta and paints the interior cyan:

```
SCREEN 1
DRAW "C2"                       'Set color to magenta
DRAW "F60 L120 E60"             'Draw a triangle
DRAW "BD30"                     'Move down into the triangle
DRAW "P1,2"                     'Paint interior
```

■ Example 2

The following example shows different ways of using the **M** macro command: with absolute movement, relative movement; using string-variable arguments; and using numeric-variable arguments.

```
SCREEN 2
DRAW "M 50,80"          'Absolute movement
DRAW "M 80,50"
DRAW "M+40,-20"         'Relative movement
DRAW "M-40,-20"
DRAW "M-40,+20"
DRAW "M+40,+20"
'Using a string variable:
X$ = "400" : Y$ = "190"
DRAW "M" + X$ + "," + Y$
'Using numeric variables (note the two "=" signs):
A = 300 : B = 120
DRAW "M="+VARPTR$(A)+",="+VARPTR$(B)
```

■ Example 3

The following program displays a clock face in high-resolution graphics mode (SCREEN 2). This clock shows the time returned by the TIME$ function.

```
SCREEN 2
LOOP:
   GOSUB GETMIN
   GOSUB FACE
   MINCHECK$ = MIN$
   WHILE MINCHECK$ = MIN$      'While still the same minute, loop
      A$ = INKEY$              'Check for input from keyboard
      IF A$ <> "" THEN END
      LOCATE 2,37 : PRINT T$
      GOSUB GETMIN
   WEND
   CLS
   GOTO LOOP

GETMIN:          'Get the string representing minutes from TIME$
   T$ = TIME$
   MIN$ = MID$(T$,4,2)
RETURN

FACE:            'Draw the clock face
   LOCATE 23,30 : PRINT "Press any key to end"
   CIRCLE (320,100),175
   HR = VAL(T$) : MIN = VAL(MIN$)      'Convert strings to numbers
   LITTLE = 360 -(30 * HR + MIN/2)     'Convert numbers to angles
   BIG = 360 -(6*MIN)
   DRAW "TA=" + VARPTR$(LITTLE) + "NU40"
   DRAW "TA=" + VARPTR$(BIG) + "NU70"
RETURN
```

END Statement

■ Syntax

END [[{ **DEF** | **IF** | **SUB** }]]

■ Action

Ends a BASIC program, multiline function definition, block **IF...THEN...ELSE** statement, or subprogram

■ Remarks

END DEF ends a multiline function definition; you must use **END DEF** with **DEF FN**.

END IF ends a block **IF...THEN...ELSE** statement; you must use **END IF** with block **IF...THEN...ELSE**.

END SUB ends a BASIC subprogram; you must use **END SUB** with **SUB**.

The **END** statement terminates program execution, closes all files, and returns control to the operating system.

The compiler always assumes an **END** statement at the conclusion of any program, so "running off the end" (omitting an **END** statement at the end of a program) still produces proper program termination.

You can place **END** statements anywhere in the program to terminate its execution.

■ See Also

DEF FN, **IF...THEN...ELSE**, **SUB...END SUB**

■ Example

The following example ends program execution if variable K's value is greater than 1000; otherwise, depending on the value of K, program execution continues at either COMPUTE or INCREASE:

```
IF K > 1000 THEN
   END
ELSEIF K = 1000 THEN
   GOTO COMPUTE
ELSE
   GOTO INCREASE
END IF
```

ENVIRON$ Function

■ Syntax

ENVIRON$ (*environmentstring*)
ENVIRON$ (*n*)

■ Action

Retrieves an environment string from the BASIC environment-string table

■ Remarks

If you specify an *environmentstring* name, but it either cannot be found in the BASIC environment-string table, or does not have any text following it, then **ENVIRON$** returns a null string; otherwise, **ENVIRON$** returns all the associated text that follows *environmentstring*= in the environment-string table.

If you specify a numeric argument (*n*), then the *n*th string in the environment-string table is returned. In such a case, the string includes all the text, including the *environmentstring* name. If the *n*th string does not exist, **ENVIRON$** returns a null string. The *n* argument can be any numeric expression; it is rounded to an integer.

■ Example 1

```
PRINT ENVIRON$("PATH")   'Prints the settings for the PATH
                         'environment variable
```

Sample output:

```
C:\BIN;C:\TOOLS
```

■ Example 2

The following example prints the current environment-string table settings:

```
I = 1
WHILE ENVIRON$(I) <> ""
    PRINT ENVIRON$(I)
    I = I + 1
WEND
```

Sample output:

```
COMSPEC=C:\COMMAND.COM
TMP=c:\tmp
PATH=C:\TOOLS;C:\BIN
PROMPT=$ec4$l$ec7$p$ec4$g$ec7$eb1
INIT=c:\tools
LIB=c:\lib
INCLUDE=c:\include
```

ENVIRON Statement

■ Syntax

ENVIRON *string-expression*

■ Action

Modifies a parameter in the DOS environment-string table

■ Remarks

The *string-expression* must be of the form *parameterid=text,* or *parameterid text.* Everything to the left of the equal sign or space is assumed to be a parameter, and everything to the right, text.

If the *parameterid* has not previously existed in the environment-string table, it is appended to the end of the table. If a *parameterid* exists in the table when the **ENVIRON** statement is executed, it is deleted and the new *parameterid* is appended to the end of the table.

The text string is the new parameter text. If the text is a null string (""), or a semicolon (";"), then the existing parameter is removed from the environment-string table, and the remaining body of the file is compressed.

You can use this statement to change the **PATH** parameter for a "child" process, or to pass parameters to a child by inventing a new environment parameter.

Errors in environment-string tables include parameters that are not strings and lack of free space. An "Out of memory" error message is printed when no more space can be allocated to the environment-string table. The amount of free space in the table is usually quite small.

■ See Also

ENVIRON$, SHELL

Example

```
'Change value of PATH environment variable:
ENVIRON "PATH=A:\SALES;A:\ACCOUNTING"

'Add a new parameter to the environment-string table:
ENVIRON "INIT=TOOLS"
```

EOF Function

■ Syntax

EOF(*filenumber*)

■ Action

Tests for the end-of-file condition

■ Remarks

Returns -1 (true) if the end of a sequential file has been reached. Use **EOF** to test for end-of-file while inputting data, to avoid "Input past end" errors.

When **EOF** is used with random-access files, it returns "true" if the last executed **GET** statement was unable to read an entire record. This happens because of an attempt to read beyond the end-of-file character.

When you use **EOF** with a communications device, the definition of the end-of-file condition is dependent on the mode (ASCII or binary) in which you opened the device. In binary mode, **EOF** is true when the input queue is empty (LOC(*n*)=0). It becomes "false" when the input queue is not empty. In ASCII mode, **EOF** is false until you type CTRL-Z, after which it remains true until you close the device.

■ Example

```
OPEN "DATA" FOR INPUT AS 1
C = 0
WHILE NOT EOF(1)
      INPUT #1, M(C)
      C = C+1
WEND
.
.
.
```

■ Syntax

ERASE *arrayname* [*arrayname...*]

■ Action

Resets the elements of static arrays, also deallocates dynamic arrays

■ Remarks

ERASE resets the elements of a *static* array to either zeros or null strings, depending on the type specified by the *arrayname*.

However, executing **ERASE** on a *dynamic* array causes the array elements to be deallocated. Before your program can refer to the dynamic array again, it must first redimension the array with either a **DIM** or **REDIM** statement. If you try to redimension an array with a **DIM** statement without first erasing it, a "Redimensioned array" run-time error will result. **ERASE** is not required when arrays are redimensioned with **REDIM**.

See Section 10.3, "Using Metacommands," for more information on declaring dynamic and static arrays.

■ See Also

CHAIN, DIM, REDIM

■ Example

```
REM $DYNAMIC
DIM A(100,100)
.
.
.
ERASE A          'This deallocates array A
REDIM A(5,5)     'Redimension array A
REM $STATIC
DIM B(50,50)
.
.
.
ERASE B          'This sets all elements of B equal to zero; B
                 'still has 2 the dimensions assigned by DIM.
```

ERDEV, ERDEV$ Functions

■ Syntax

ERDEV
ERDEV$

■ Action

Provides a way to obtain device-specific status information

■ Remarks

ERDEV is an integer function returning the error code from the last device to declare an error. **ERDEV$** is a string function that returns the name of the device that generated the error.

You cannot set these functions.

ERDEV is set by the "interrupt 24H handler" when an error within the operating system is detected.

ERDEV returns the DOS "INT 24H" error code in its lower eight bits; in its upper eight bits, it returns bits 15, 14, 13, XX, 3, 2, 1, and 0 of the device-attribute word. The XX value is always zero.

■ Example

If a user-installed device driver, MYLPT2, runs out of paper, and the driver's error number for that problem is 9, then

```
PRINT ERDEV, ERDEV$
```

prints this output:

```
 9             MYLPT2
```

■ Syntax

ERR
ERL

■ Action

Returns error status

■ Remarks

If you are using an error-handling routine, the function **ERR** contains the error code for the error, and the function **ERL** contains the line number of the line in which the error was found. The **ERR** and **ERL** functions are usually used in **IF...THEN** statements to direct program flow in the error-handling routine.

You can set **ERL** and **ERR** by using the **ERROR** statement.

Because **ERL** and **ERR** are reserved words, neither may appear to the left of the equal sign in an assignment statement.

■ Compiler/Interpreter Differences

If you are using line labels in a compiled program, the **ERL** function will return only the last line *number* encountered before the line in which the error occurs. If there are no line numbers in your program, **ERL** will return 0.

■ See Also

ERROR, ON ERROR, RESUME

■ Example

```
IF ERR=23 THEN IF ERL=1000 THEN PRINT "Abort."
```

ERROR Statement

■ Syntax

ERROR *integer-expression*

■ Action

Simulates the occurrence of a BASIC error, or allows the user to define error codes

■ Remarks

The value of *integer-expression* must be greater than 0 and less than 256. If the value of *integer-expression* equals an error code already in use by BASIC, then the **ERROR** statement simulates the occurrence of that error and prints the corresponding error message. (See Appendix E, "Error Messages," for a complete list of error codes used by BASIC.)

To define your own error code, use a value that is greater than any used by standard BASIC error codes. (Use the highest available values to maintain future compatibility with Microsoft BASIC error codes.)

If an **ERROR** statement specifies a code for which no error message has been defined, the message

```
Unprintable error
```

is printed. Execution of an **ERROR** statement for which there is no error-handling routine causes an error message to be printed and execution to halt.

■ See Also

ERL, ERR, ON ERROR, RESUME

■ Example

```
ON ERROR GOTO HANDLER
OPENFILE:
    INPUT "Name of file to update";FILESPEC$
    IF FILESPEC$ = "" THEN END
    OPEN FILESPEC$ FOR INPUT AS #1
    PRINT "The first five lines of ";FILESPEC$;" are:" : PRINT
    FOR I = 1 TO 5
        LINE INPUT #1, TEMP$
        PRINT TEMP$
    NEXT
    PRINT : INPUT "Is this the correct file";R$
    IF LEFT$(R$,1) <> "y" THEN ERROR 200    'Define error 200
    .
    .
    .
    END

HANDLER:         'Error-handling routine
    NUMBER = ERR
    IF NUMBER = 53 THEN    'Run-time error for "file not found"
        CLOSE #1
        PRINT "File not in this directory"
        PRINT "Enter new file spec ([d:]\xxx\...\xxx) or"
        PRINT "press <RETURN> to end program"
        RESUME OPENFILE
    ELSEIF NUMBER = 200 THEN    'User entered "n"
        CLOSE #1
        RESUME OPENFILE
    ELSE
        ERROR NUMBER        'Error for which there is no recovery
        ON ERROR GOTO 0     'action: print message, disable error
    END IF                  'handling and stop program execution
```

Sample output:

```
Name of file to update? autoexec.bat
File not in this directory
Enter new file spec ([d:]\xxx\...\xxx) or
press <RETURN> to end program
Name of file to update? \autoexec.bat
The first five lines of \autoexec.bat are:

echo off
support
echo Press the RETURN key to make a back-up copy of the disk
echo you've been working on.
echo Press the CTRL and BREAK keys together if you do not want

Is this the correct file? y
```

EXIT Statement

■ Syntax

EXIT {**DEF** | **SUB**}

■ Action

Exits a multiline function definition or BASIC subprogram

■ Remarks

Use **EXIT DEF** to exit a multiline function definition; use **EXIT SUB** to exit a BASIC subprogram.

The **EXIT DEF** and **EXIT SUB** statements do not define the end of a multiline function definition or subprogram: a complete definition of these program structures still requires an **END DEF** or **END SUB** statement.

■ See Also

DEF FN, SUB...END SUB

■ Syntax

EXP(x)

■ Action

Calculates the exponential function

■ Remarks

This function returns e, (the base of natural logarithms) to the power of x. The exponent x must be $\leq$ 88.02969. If x is greater than 88.02969, the "Overflow" error message is displayed.

The calculation of **EXP** is performed in single precision by default; if the argument x is double precision, **EXP** is calculated in double precision.

■ See Also

LOG

■ Example

The following program uses the **EXP** function to calculate the growth of a bacterial colony over a 30-day period. Since the growth of the population is related to its ever-changing size, its growth is exponential.

```
INPUT "Initial bacterial population"; COLONYO
INPUT "Growth rate per day as a percentage of population"; RATE
R = RATE/100 : FORM$="##          ###,###"
PRINT : PRINT "Day        Population" : PRINT
FOR T = 0 TO 30 STEP 5
    PRINT USING FORM$; T, COLONYO * EXP(R*T)
NEXT
```

Output:

```
Initial bacterial population? 10000
Growth rate per day as a percentage of population? 10

Day         Population

 0            10,000
 5            16,487
10            27,183
15            44,817
20            73,891
25           121,825
30           200,855
```

■ Syntax

FIELD [[#]]*filenumber*, *fieldwidth* **AS** *stringvariable*...

■ Action

Allocates space for variables in a random-access file buffer

■ Remarks

The *filenumber* parameter is the number under which the file is opened. The *fieldwidth* parameter is the number of characters allocated to the corresponding string variable.

The total number of bytes that you allocate in a **FIELD** statement must not exceed the record length that you specified when opening the file. Otherwise, a "Field overflow" error message is generated. (The default record length is 128 bytes.)

Any number of **FIELD** statements may be executed for the same file. All **FIELD** statements that have been executed remain in effect at the same time.

All field definitions for a file are removed when the file is closed; that is, all fielded strings associated with the file are set to null.

Important

Do not use a fielded variable name in an **INPUT** or assignment statement if you wish the variable to remain fielded. Once a variable name is fielded, it points to the correct place in the random-access file buffer. If a subsequent **INPUT** or assignment statement with that variable name is executed, the variable's pointer no longer refers to the random-access record buffer, but to string space.

■ Compiler/Interpreter Differences

When a random-access file is closed with a **CLOSE** or **RESET** statement in a compiled program, all fielded variables associated with that file are reset to null strings. When a random-access file is closed in an interpreted program, fielded variables retain the last value assigned to them by a **GET** statement.

■ See Also

GET, LSET, OPEN, PUT, RSET

■ Example 1

This example allocates the first 20 positions (bytes) in the random-access file buffer to the string variable N$, the next 10 positions to ID$, and the next 40 positions to ADD$. **FIELD** does not place any data in the random-access file buffer.

```
FIELD 1,20 AS N$,10 AS ID$,40 AS ADD$
```

■ Example 2

Example 2 illustrates a multiply-defined random-access file buffer. In the first **FIELD** statement, the 67-byte buffer is broken up into 6 separate variables for name, address, city, state, and zip code. In the second **FIELD** statement, the same buffer is assigned entirely to one variable, PLIST$. The remainder of this example checks to see if ZIP$, which contains the zip code, falls within a certain range; if it does, it prints the complete address string.

```
OPEN "MAILLIST" FOR RANDOM AS #1 LEN=67
FIELD #1, 10 AS FNAM$, 15 AS LNAM$, _
          25 AS ADDR$, 10 AS CTY$, 2 AS ST$, 5 AS ZIP$
FIELD #1, 67 AS PLIST$
GET #1, 1
WHILE NOT EOF(1)
    ZCHECK$ = ZIP$
    IF (ZCHECK$ > "85699" AND ZCHECK$ < "85801") THEN
        INFO$ = PLIST$
        PRINT LEFT$(25,INFO$)
        PRINT MID$(16,25,INFO$)
        PRINT RIGHT$(17,INFO$)
    END IF
    GET #1
WEND
```

■ Example 3

This example shows the construction of a **FIELD** statement using an array of elements of equal size:

```
OPEN "MAILLIST" FOR RANDOM AS #1 LEN=128
FOR LOOP% = 0 TO 7
    FIELD #1,(LOOP%*16) AS OFFSET$,16 AS A$(LOOP%)
NEXT LOOP%
```

The result is equivalent to the following single declaration:

```
FIELD 1,16 AS A$(0),16 AS A$(1),...,16 AS A$(6),16 AS A$(7)
```

■ Example 4

This example creates a FIELD in the same manner as Example 3. However, the element size varies with each element.

```
OPEN "MAILLIST" FOR RANDOM AS #1 LEN = 221
RECLENGTH% = 0
FOR I% = 1 TO 14
    READ SIZE%
    FIELD #1, RECLENGTH% AS OFFSET$, SIZE% AS A$(I)
    RECLENGTH% = RECLENGTH% + SIZE%
NEXT
DATA 15,10,40,5,15,1,30,1,17,8,50,1,20,8
```

The equivalent declaration is:

```
FIELD #1, 15 AS A$(0), 10 AS A$(1),...,8 AS A$(14)
```

FILES Statement

■ Syntax

FILES [[*filespec*]]

■ Action

Prints the names of files residing on the specified disk

■ Remarks

The *filespec* includes either a file name or a path name and an optional device designation.

If you omit *filespec*, all the files in the current directory are listed. The *filespec* is a string formula that can contain question marks (?) or asterisks (*), used as wild cards. A question mark will match any single character in the file name or extension. An asterisk will match one or more characters starting at that position. The asterisk is a shorthand notation for a series of question marks. You don't need to use the asterisk when you are asking for all the files on a drive.

If you use a *filespec* without an explicit path, the current directory is the default.

Note that, regardless of the path name contained in *filespec*, the header printed by **FILES** is always the current directory.

■ Examples

```
FILES                'Shows all files on the current directory

FILES "*.BAS"        'Shows all files with the extension .BAS

FILES "B:*.*"        'Shows all files on drive B

FILES "B:"           'Equivalent to "B:*.*"

FILES "TEST?.BAS"    'Shows all five-letter files whose names start
                     'with "TEST" and end with the .BAS extension

FILES "\SALES"      'If SALES is a directory, this statement
                     'displays all files in SALES; if SALES is
                     'a file in the current directory, this
                     'statement displays the name SALES
```

■ Syntax

FIX(x)

■ Action

Returns the truncated integer part of x

■ Remarks

FIX(x) is equivalent to **SGN**(x)*__INT__(**ABS**(x)). The difference between **FIX** and **INT** is that, for negative x, **FIX** returns the first negative integer greater than x, while **INT** returns the first negative integer less than x.

■ See Also

CINT, **INT**

■ Example

This program calculates the percentage change in profits between quarters. The **FIX** function allows an estimate which is consistently truncated at the decimal, whether the result is positive or negative.

```
DIM PROFITS(4)
FORMAT$="  #                #### %"
FOR QRT = 1 TO 4
    PRINT "Enter profit for Quarter" QRT ":  ";
    INPUT "",PROFITS(QRT)
NEXT
PRINT
PRINT "","Percent Change From"
PRINT "Quarter","Previous Quarter"
PRINT
FOR QRT = 2 TO 4
    CHANGE = 100*(PROFITS(QRT)-PROFITS(QRT-1))/PROFITS(QRT-1)
    PRINT USING FORMAT$; QRT, FIX(CHANGE)
NEXT
```

Output:

```
Enter profit for Quarter 1 :  45600
Enter profit for Quarter 2 :  47800
Enter profit for Quarter 3 :  40900
Enter profit for Quarter 4 :  28500

                Percent Change From
Quarter         Previous Quarter

  2                    4 %
  3                  -14 %
  4                  -30 %
```

■ Syntax

FOR *counter* = *start* **TO** *end* [[**STEP** *increment*]]
.
.
.
NEXT [[*counter* [[,*counter*...]]]]

■ Action

Allows a series of instructions to be performed in a loop a given number of times

■ Remarks

The **FOR** statement takes the following arguments:

Argument	Description
counter	A variable used as the loop counter
start	The initial value of the counter
end	The final value of the counter
increment	The amount by which the counter is incremented each loop

The program lines following the **FOR** statement are executed until the **NEXT** statement is encountered. Then *counter* is adjusted by the amount specified by **STEP**, and compared with the final value, *end*. (If you do not specify **STEP**, the increment is assumed to be one.) If *counter* is still not greater than *end*, control branches back to the statement after the **FOR** statement and the process is repeated. When *counter* is greater than *end*, execution continues with the statement following the **NEXT** statement.

If **STEP** is negative, the final value of the counter is set to be less than the initial value. The counter is decreased each time through the loop, and the loop is executed until the counter is less than the final value.

The body of a **FOR...NEXT** loop is always executed at least once, since the test for termination occurs at the bottom of the loop. For example, the following loop prints the value for EXP(3) before passing control to the statement following NEXT:

```
FOR I = 3 TO 3.2
    PRINT EXP(I)
NEXT
```

It is a good idea not to change the loop variable within the loop, since doing so can make the program more difficult to read and debug.

Nested Loops

You can nest **FOR...NEXT** loops; that is, you can place a **FOR...NEXT** loop within another **FOR...NEXT** loop. When loops are nested, each loop must have a unique variable name as its counter. The **NEXT** statement for the inside loop must appear before the **NEXT** statement for the outside loop. The following construction is the correct form:

```
FOR I = 1 TO 10
   FOR J = 1 TO 10
      FOR K = 1 TO 10
       .
       .
       .
      NEXT K
   NEXT J
NEXT I
```

A **NEXT** statement that has the form

```
NEXT K, J, I
```

is equivalent to the following sequence of statements

```
NEXT K
NEXT J
NEXT I
```

Important

If you omit the variable in a **NEXT** statement, the **NEXT** statement matches the most recent **FOR** statement. If a **NEXT** statement is encountered before its corresponding **FOR** statement, a "NEXT without FOR" error message is generated and execution is terminated.

■ Compiler/Interpreter Differences

Unlike the interpreter, the QuickBASIC compiler supports double-precision control values (*start*, *end*, and *counter*) in its **FOR...NEXT** loops. However, if the control values fall within the range for integers, you should use integer control values for maximum speed.

■ Example

```
'Print the first eleven columns of Pascal's Triangle, in which
'each number is the sum of the two numbers immediately above it
'and immediately below it in the preceding column:

DIM A(11,11)
FOR M = 1 TO 11
    A(M,1) = 1 : A(M,M) = 1  'Top and bottom of each column is 1
NEXT
FOR M = 3 TO 11
    FOR N = 2 TO M-1
        A(M,N) = A(M-1,N-1) + A(M-1,N)
    NEXT
NEXT
STARTROW = 13                'Go to the middle of the screen
FOR M = 1 TO 11
    COL = 6 * M
    ROW = STARTROW
    FOR N = 1 TO M
        LOCATE ROW,COL : PRINT A(M,N)
        ROW = ROW + 2        'Go down 2 rows to print next number
    NEXT
    PRINT
    STARTROW = STARTROW - 1  'Next column starts 1 row above
NEXT                         'preceding column
```

FOR...NEXT Statement

Output:

```
                                                    1
                                                  1   1
                                                1   2   1
                                              1   3   3   1
                                            1   4   6   4   1
                                          1   5  10  10   5   1
                                        1   6  15  20  15   6   1
                                      1   7  21  35  35  21   7   1
                                    1   8  28  56  70  56  28   8   1
                                  1   9  36  84 126 126  84  36   9   1
                                1  10  45 120 210 252 210 120  45  10   1
```

■ Syntax

FRE(*numeric-expression*)
FRE(*string-expression*)

■ Action

Returns the size of free string space

■ Remarks

FRE returns the following values, depending on the nature and value of its argument:

Argument	Value Returned by FRE
numeric-expression	(Where *numeric-expression* is not equal to -1) the size of the next free block of string space (no compression of string space).
string-expression	The size in bytes of the free string value. Before **FRE** returns the number of free bytes, it firsts compacts free string space into a single block.
-1	The size in bytes of the largest free LNA entry.

■ Example

```
PRINT "Before dimensioning arrays:   " FRE(""),FRE(0),FRE(-1)
DIM LARGE%(150,150), BIG$(5000)
PRINT "After dimensioning arrays:    " FRE(""),FRE(0),FRE(-1)
```

Sample output (the actual values returned by **FRE** on your computer may differ from this example):

```
Before dimensioning arrays:    58420      58420      322120
After dimensioning arrays:     38404      38404      276496
```

GET Statement—File I/O

■ Syntax

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

■ Action

Reads a record from a random-access disk file into a random-access buffer

■ Remarks

The *filenumber* is the number under which you opened the file. If you omit *recordnumber,* the next record (the one after the last **GET**) is read into the buffer. The largest possible record number is 2^{24}–1, or 16,777,215.

The **GET** and **PUT** statements allow fixed-length input and output for BASIC communications files. However, because of possible noise associated with telephone-line communications, it is recommended that you not use **GET** and **PUT** for telephone communication. If there is a communications failure, **GET** will wait indefinitely for *recordnumber* characters.

Note

You can use **INPUT#** and **LINE INPUT#** after a **GET** statement to read characters from the random-access file buffer.

You may use the **EOF** function after a **GET** statement to see if that **GET** was beyond the end-of-file character.

■ See Also

CVD, **CVI**, **CVS**, **FIELD**, **LSET**, **MKD$**, **MKI$**, **MKS$**, **PUT** (File I/O)

■ Example

The following fragment opens the file INVENT.DAT for random access, then searches through all the records in the file to find the items ordered on a certain date:

```
OPEN "INVENT.DAT" FOR RANDOM AS #1 LEN=48
'Define record buffer:
FIELD #1, 5 AS PART$, 30 AS DESC$, 8 AS ORDER$, 5 AS QTY$
'Record numbers range from 1 to the number of
'records in the file:
TOTAL% = LOF(1)/48
PRINT "This will print all items ordered on a certain date."
INPUT "Date (mm/dd/yy): ",SEARCH$
FOR I% = 1 TO TOTAL%
    GET #1, I%  'Read record I% into the file buffer
    'Print the other fields of the buffer if the date matches:
    IF ORDER$ = SEARCH$ THEN
        PRINT PART$ : PRINT DESC$ : PRINT QTY$
    END IF
NEXT
```

GET Statement—Graphics

■ Syntax

GET(*x1,y1*)–(*x2,y2*),*arrayname*

■ Action

Stores graphic images from the screen

■ Remarks

The **GET** statement transfers a screen image into the array specified by *arrayname*. This image is bounded by a rectangle that is defined by the specified points. The **PUT** statement, associated with **GET**, transfers the image stored in the array onto the screen.

The range (*x1,y1*)–(*x2,y2*) is a rectangular area on the display screen. The coordinates (*x1,y1*) and (*x2,y2*) represent diagonally opposite corners of the rectangle.

The *arrayname* is the name assigned to the array that holds the image. This array can be of any numeric type; its dimensions must be large enough to hold the entire image. The following formula gives the required size of the array in bytes:

4 + INT((*x2* **−** *x1* **+ 1)****bits-per-pixel* **+ 7)/8) * ((***y2* **−** *y1***) + 1)**

The *bits-per-pixel* value depends on the specification set in the **SCREEN** statement; Table 11.1 shows the number of bits per pixel for each screen mode.

Table 11.1

Bits per Pixel Values for Different Screen Modes

Mode	Bits per Pixel
SCREEN 1	2
SCREEN 2	1
SCREEN 7	4
SCREEN 8	4
SCREEN 9	2 (if 64K of EGA memory) 4 (more than 64K of EGA memory)
SCREEN 10	2

The bytes per element of an array are as follows:

- 2 bytes for an integer array
- 4 bytes for a single-precision array
- 8 bytes for a double-precision array

For example, suppose you wanted to use the **GET** statement to store an image in high resolution (**SCREEN 2**). If the coordinates of the upper-left corner of the image are (0,0), and the coordinates of the lower-right corner are (32,32), then the required size of the array in bytes is 4 + INT((33 + 7)/8) * (33), or 169. This means an integer array with 84 elements would be large enough to hold the image.

Unless the array type is integer, the contents of an array after a **GET** will be meaningless when interpreted directly.

One of the most useful things that can be done with **GET** and **PUT** is animation. See the **PUT** statement (graphics) reference pages for a discussion of animation.

■ See Also

PUT (Graphics)

GOSUB...RETURN Statements

■ Syntax

GOSUB { *linenumber1* ¦ *linelabel1*}
.
.
.
RETURN [[{ *linenumber2* ¦ *linelabel2*}]]

■ Action

Branches to, and returns from, a subroutine

■ Remarks

The first line of the subroutine is the *linenumber1* or *linelabel1* argument in the **GOSUB** statement that identifies the first line of the subroutine. In addition to the simple **RETURN** statement, the compiler supports **RETURN** *linenumber2* (or **RETURN** *linelabel2*), thus allowing a **RETURN** from a subroutine to the statement having the specified line number or label, instead of a normal return to the statement following the **GOSUB** statement. Use this type of return with care, since any other **GOSUB**, **WHILE**, or **FOR** statements that were active at the time of the **GOSUB** will remain active, and errors such as "FOR without NEXT" may result.

Additionally, event-trapping routines should never use the **RETURN** *linenumber2* option unless only one event can occur. This is because an event occurring during an event-trapping routine leaves confusing information on the stack, resulting in unpredictable program behavior.

You can call a subroutine any number of times in a program. You can also call a subroutine from within another subroutine. Such nesting of subroutines is limited only by available memory. However, **GOSUB** statements should not call themselves; such "recursive" subroutine calls may have disastrous effects. Moreover, **RETURN** *linenumber* or **RETURN** *linelabel* can only return control to a statement in the main program, or to a statement in the calling routine, if the current subroutine was called from another subroutine. (See Example 1.)

A subroutine may contain more than one **RETURN** statement. Simple **RETURN** statements (without the *linenumber2* option) in a subroutine cause BASIC to branch back to the statement following the most recent **GOSUB** statement.

Subroutines may appear anywhere in the program, but it is good programming practice to make them readily distinguishable from the main program. To prevent inadvertent entry into a subroutine, precede it with a **STOP**, **END**, or **GOTO** statement that directs program control around the subroutine.

Warning

The preceding discussion of subroutines applies only to the targets of **GOSUB** statements, *not* subprograms delimited by **SUB...END SUB** statements. Entering and exiting **SUB...END SUB** blocks with **GOSUB...RETURN** statements is not supported.

■ See Also

RETURN, **SUB...END SUB**

■ Example 1

The following example shows the correct use of **RETURN***linelabel* statements:

```
PRINT "in main routine"
GOSUB SUB1
PRINT "this line in main routine should be skipped"
LABEL1:
   PRINT "back in main routine"
   END

SUB1:
   PRINT "in subroutine one"
   GOSUB sub2
   PRINT "this line in subroutine one should be skipped"
LABEL2:
   PRINT "back in subroutine one"
   RETURN LABEL1
```

```
SUB2:
   PRINT "in subroutine two"
   RETURN label2          'Cannot return from here to main
                          'program - only to SUB1
```

Output:

```
in main routine
in subroutine one
in subroutine two
back in subroutine one
back in main routine
```

■ Example 2

The following program computes arcsine values for arguments between –1 and 1. Arcsine is the inverse of the sine function, so the value returned will be an angle whose measure is between $\pi/2$ radians and $3\pi/2$ radians (90° to 270°).

```
INPUT "sine"; X
WHILE (X < -1 OR X > 1)
   PRINT "Illegal sine value."
   PRINT "Sine must be >= -1 and =< 1."
   INPUT "sine"; X
WEND
PI = 3.141593
GUESS1 = PI/2 : GUESS2 = 3 * (PI/2)
WHILE ABS(GUESS2 - GUESS1) > .0000005
   GOSUB NEWVAL
WEND
PRINT "arcsin " X "= " MIDVAL "radians"
END

NEWVAL:
    MIDVAL = (GUESS1 + GUESS2)/2
    IF SIN(MIDVAL) > X THEN
        GUESS1 = MIDVAL
    ELSE
        GUESS2 = MIDVAL
    END IF
RETURN
```

Sample output:

```
sine? 3
Illegal sine value.
Sine must be >= -1 and =< 1.
sine? .43
arcsin  .43 = 2.6971 radians
```

GOTO Statement

■ Syntax

GOTO { *linenumber* | *linelabel*}

■ Action

Branches unconditionally to the line specified by *linenumber* or *linelabel*

■ Remarks

The compiler **GOTO** statement allows branching to lines identified by either line numbers or line labels.

If the statement identified by *linenumber* or *linelabel* is an executable statement, execution continues with that statement; if it is a nonexecutable statement, such as a **REM** or **DATA** statement, execution proceeds with the first executable statement encountered after *linenumber* or *linelabel.*

It is good programming practice to use structured control statements (**IF...THEN...ELSE**, **WHILE...WEND**) instead of **GOTO** statements as a way of branching, because a program with many **GOTO** statements can be difficult to read and debug.

A **GOTO** statement can only branch to another statement at the same level of a program; that is, you cannot use **GOTO** to enter or exit a subprogram, subroutine, multiline function definition, or error-handling subroutine. You can, however, use **GOTO** to control program flow within any of these program structures.

■ Example

The following program prints the area of the circle that has the input radius:

```
PRINT "Input 0 to end."
START:
   INPUT R
   IF R <= 0 THEN
     END
   ELSE
      A = 3.14 * R^2
      PRINT "Area =";a
   END IF
GOTO START
```

Sample output:

```
Input 0 to end.
? 5
Area = 78.5
? 7
Area = 153.86
? 12
Area = 452.16
? 0
```

HEX$ Function

■ Syntax

HEX$(*expression*)

■ Action

Returns a string that represents the hexadecimal value of the decimal argument *expression*

■ Remarks

The argument *expression* is rounded to an integer before **HEX$** evaluates it.

■ See Also

OCT$

■ Example

```
INPUT X
A$ = HEX$(X)
PRINT X "decimal is " A$ " hexadecimal"
```

Output:

```
? 32
 32 decimal is 20 hexadecimal
```

■ Syntax 1

IF *booleanexpression* **THEN**
    statement1
    [*statement2*]
    .
    .
    .
[**ELSEIF** *booleanexpression* **THEN**
    statement10
    [*statement11*]]
    .
    .
    .
[**ELSE**
    statement15
    [*statement16*]]
    .
    .
    .
END IF

■ Syntax 2

IF *booleanexpression* **THEN** *statement1* [**ELSE** *statement2*]
IF *booleanexpression* **THEN** *linenumber* [**ELSE** *linenumber*]
IF *booleanexpression* **THEN GOTO** *linelabel* [**ELSE GOTO** *linelabel*]

■ Action

Allows conditional execution or branching, based on the evaluation of a Boolean expression

■ Remarks

In the one-line form of **IF...THEN...ELSE**, (Syntax 2) the statement following the **THEN** is executed if the *booleanexpression* is true; if the *booleanexpression* is false, the statement following the **ELSE** is executed. If the **ELSE** clause is not present, control simply passes to the next statement in the program.

A Boolean expression is any expression returning a true (nonzero) or false (0) value. For example, if X equals 9, and Y equals 10, then the Boolean expression X = 9 OR Y = 9 is true, since at least one of its clauses (X = 9) is true; therefore, the following fragment would print Found a nine:

```
IF X = 9 OR Y = 9 THEN
    PRINT "Found a nine"
END IF
```

For more information on expressions and operators, see Section 9.6.

■ Compiler/Interpreter Differences

Unlike the interpreter, the QuickBASIC compiler supports "block" **IF...THEN...ELSE** (Syntax 1), as well as the single-line form (Syntax 2). The block form provides more structure and flexibility than its single-line counterpart by permitting conditional control transfer across several lines. This not only allows many statements to be executed in the condition, but allows longer and more complicated statements and structures within the **THEN...ELSE** portion of the statement, with no line-length restrictions.

Block **IF...THEN...ELSE** makes clear which statements will be executed if the condition is met, and allows you to create complicated case structures while preserving the readability of your program. Such programs are easier to maintain and debug.

The following four rules apply to block **IF...THEN...ELSE** statements:

1. There can be no statements immediately after **THEN**. If anything besides a comment follows on the same line as **THEN**, the compiler assumes it is a single-line **IF...THEN...ELSE** statement.
2. **ELSE**, **ELSEIF**, and **END IF** can be preceded only by a line number or label; otherwise, they must be the first item on the line.
3. A block **IF** must be the first statement on a line.
4. The block *must* be terminated by the **END IF** statement.

■ Example

The following program fragments demonstrate the use of block **IF...THEN...ELSE**, and illustrate the difference between the single-line form and the block form.

The following example computes simple discount prices using the single-line **IF...THEN...ELSE** form; note that this is just one *logical* line carried over four *physical* lines by use of the line-continuation character (_):

```
IF (x >= 10000) THEN PRINT PRICE! = x*.25! ELSE _
   IF (x < 10000) and (x >= 5000) THEN PRINT PRICE! = x*.2! ELSE _
         IF (x < 2500) AND (x >=1500) THEN PRINT PRICE! = x*.1 ELSE _
            PRINT "no discount"
.
.
.
```

The following example uses block **IF...THEN...ELSE** to make the preceding more readable:

```
IF (x >= 10000) THEN
        PRINT PRICE! = x * .25!
   ELSEIF (x < 10000) AND (x >= 5000) THEN
        PRINT PRICE! = x * .02!
   ELSEIF (x < 5000) AND ( x >= 2500) THEN
        PRINT PRICE! = x*.01
   ELSE PRINT "No discount"
END IF
.
.
.
```

INKEY$ Function

■ Syntax

INKEY$

■ Action

Reads a character from the keyboard

■ Remarks

INKEY$ returns either a one- or two-byte character string containing a character read from the standard input device, or a null string if no character is pending there. A one-character string contains the actual character read from the keyboard, while a two-character string indicates a special extended code, the first character of which is hexadecimal 00. For a complete list of these codes, see Appendix A, "ASCII Character Codes."

The keyboard is usually the standard input device. No characters are echoed to the screen by **INKEY$**; instead, all characters are passed through to the program except for the following:

- CTRL-BREAK, which terminates the program
- CTRL-ALT-DEL, which does a system reboot
- CTRL-NUM LOCK, which causes program execution to pause
- PRTSC, which prints the screen

However, the compiler *can* read a CTRL-BREAK without causing the program to end if the Debug or **/d** option is not specified at compile time.

■ Example

The following example is a subroutine that checks to see if there has been any input from the keyboard during the interval `TIMELIMIT%`. If there hasn't been any input, the subroutine continues until `TIMELIMIT%` is exceeded, the variable `TIMEOUT%` is set to 1 and control is returned to the main program.

If there has been input, the innermost **IF...THEN** block checks to see if the input is ASCII character 13, the carriage return. If it is, TIMEOUT% is set to 0 and control is returned to the main program.

```
REM Timed input subroutine
RESPONSE$ = ""
FOR I% = 1 TO TIMELIMIT%
    A$ = INKEY$
    IF LEN(A$) <> 0 THEN
        IF ASC(A$) = 13 THEN
            TIMEOUT% = 0
            RETURN
        END IF
    ELSE
        RESPONSE$ = RESPONSE$ + A$
    END IF
NEXT
TIMEOUT% = 1
RETURN
```

INP Function

■ Syntax

INP(*port*)

■ Action

Returns the byte read from an I/O *port*

■ Remarks

The *port* must be an integer in the range 0–65,535. The **INP** function complements the **OUT** statement.

■ See Also

OUT

■ Example

This instruction reads a byte from port `54321` and assigns it to the variable `A`:

```
A=INP(54321)
```

In 8086 assembly language, this is equivalent to:

```
MOV DX,54321
IN  AL,DX
```

■ Syntax

INPUT[;][*"promptstring"*{;|,}] *variablelist*

■ Action

Allows input from the keyboard during program execution

■ Remarks

INPUT causes the program to pause and print a question mark to indicate that it is waiting for data. If you include a *promptstring*, **INPUT** prints this before the question mark. You can then enter the required data at the keyboard.

Use a comma instead of a semicolon after the prompt string if you want to suppress the question mark. For example, the following statement prints the prompt with no question mark:

```
INPUT "ENTER BIRTHDATE",B$
```

If **INPUT** is immediately followed by a semicolon, then the RETURN that you type to input data does not echo a carriage-return–line-feed sequence, so the cursor remains on the same line as your response.

The data that you enter is assigned to the variables in *variablelist*. The number of data items that you supply must be the same as the number of variables in the list. Data items are separated by commas. The first character encountered after a comma that is not a space, carriage return, or line feed is assumed to be the start of a new item.

The variable names in the list may be numeric- or string-variable names (including subscripted variables). The type of each data item that you input must agree with the type specified by the variable name. (Strings input to an **INPUT** statement need not be surrounded by quotation marks.) If this first character is a quotation mark ("), the string item will consist of all characters read between the first quotation mark and the second. This means a quoted string may not contain a quotation mark as a character. If the first character of the string is not a quotation mark, the string is an unquoted string, and will terminate on a comma, carriage return, or line feed.

Responding to **INPUT** with too many or too few items, or with the wrong type of value (for example, numeric instead of string), causes the following message to be printed:

```
?Redo from start
```

No assignment of input values is made until you give an acceptable response.

It is possible to edit a line of input before you press RETURN. The following list describes the key combinations that allow you to move the cursor, delete text, and insert text:

Keys	Action
CTRL-\ or RIGHT	Moves cursor one character to the right.
CTRL-] or LEFT	Moves cursor one character to the left.
CTRL-F or CTRL-RIGHT	Moves cursor one word to the right.
CTRL-B or CTRL-LEFT	Moves cursor one word to the left.
CTRL-K or HOME	Moves cursor to the beginning of the input line.
CTRL-N or END	Moves cursor to the end of the input line.
CTRL-R or INS	Toggles insert mode on and off. When insert mode is on, characters above and to the right of the cursor are shifted to the right as new characters are entered.
CTRL-I or TAB	Tabs right and inserts (insert mode on), or overwrites (insert mode off).
DEL	Deletes the character at the cursor.
CTRL-H or BACKSPACE	Deletes the character to the left of the cursor, unless the cursor is at the beginning of the input, in which case it deletes the character at the cursor.
CTRL-E or CTRL-END	Deletes to the end of the line.
CTRL-U or ESC	Deletes entire line, regardless of cursor position.
CTRL-M or RETURN	Stores input line.

CTRL-T	Toggles function key label display on and off at bottom of screen.
CTRL-BREAK or CTRL-C	Terminates input, exits program.

■ Example 1

```
INPUT X
PRINT X "squared is" X^2
END
```

Output:

```
? 5

 5 squared is 25
```

■ Example 2

```
PI = 3.141593 : R = -1
PRINT "Enter 0 to end."
WHILE R
    PRINT
    INPUT;"If radius = ", R
    IF R > 0 THEN
        A = PI*R^2
        PRINT ", the area of the circle =" A
    END IF
WEND
```

Output:

```
Enter 0 to end.

If radius = 4, the area of the circle = 50.26549

If radius = 5, the area of the circle = 78.53983

If radius = 0
```

INPUT# Statement

■ Syntax

INPUT# *filenumber*, *variablelist*

■ Action

Reads data items from a sequential device or file and assigns them to variables

■ Remarks

The *filenumber* is the number used when the file is opened for input. The *variablelist* contains the names of the variables that are assigned values read from the file. (The variable type must match the type specified by the variable name.) Unlike **INPUT**, **INPUT#** does not print a question mark.

The data items in the file should appear just as they would if you were entering data in response to an **INPUT** statement. With numeric values, leading spaces, carriage returns, and line feeds are ignored. The first character encountered that is not a space, carriage return, or line feed is assumed to be the start of a number. The number terminates on a space, carriage return, line feed, or comma.

If BASIC is scanning the sequential data file for a string item, it will also ignore leading spaces, carriage returns, and line feeds. If end-of-file is reached when a numeric or string item is being **INPUT**, the item is terminated.

■ See Also

INPUT, **INPUT$**

■ Example

```
INPUT #2, A, B, C
```

■ Syntax

INPUT$(*n*[,[#]*filenumber*])

■ Action

Returns a string of *n* characters read from the file specified by *filenumber*

■ Remarks

If the given file is opened for random access, the argument *n* must be less than or equal to the record length set by the **LEN** option in the **OPEN** statement (or less than or equal to 128 if the record length is not set). If the given file is opened for sequential access, then *n* must be less than or equal to 32,767.

If the *filenumber* is not specified, the characters are read from the standard input device. (If input has not been redirected, the keyboard is the standard input device).

If the keyboard is used for input, no characters will be echoed on the screen. All control characters are passed through except CTRL-BREAK, which is used to interrupt the execution of the **INPUT$** function.

You can use the DOS redirection symbols (<, >, or >>) or the pipe symbol (|) to redefine the standard input or standard output for an executable file created with QuickBASIC. (See your operating system manual for a complete discussion of redirection and pipes.)

■ Example 1

This program copies the named file to the screen, printing only alphanumeric and punctuation characters in the ASCII range 32–126, as well as tabs, line feeds, and carriage returns.

```
'ASCII codes for tab, line feed, and carriage return:
HT = 9 : LF = 10 : CR = 13

INPUT "Print which file"; FILENAME$
OPEN FILENAME$ FOR INPUT AS #1
CLS
WHILE NOT EOF(1)
    S$=INPUT$(1,#1): C=ASC(S$)
    IF (C >= 32 AND C <= 126) OR C = HT OR C = LF OR C = CR THEN
        PRINT S$;
    END IF
WEND
END
```

■ Syntax

INSTR([*start*,]*string-expression1*,*string-expression2*)

■ Action

Searches for the first occurrence of string *string-expression2* in *string-expression1*, and returns the position at which the match is found

■ Remarks

The optional offset *start* sets the position for starting the search; *start* must be in the range 1–32,767. If *start* is not given, **INSTR** begins the search for *string-expression2* at the first character of *string-expression1*.

The arguments *string-expression1* and *string-expression2* can be string variables, string expressions, or string literals.

The value returned by **INSTR** depends on the following conditions:

Condition	Return Value
string-expression2 found in *string-expression1*	The position at which the match is found
start greater than length of *string-expression1*	0
string-expression1 is null string	0
string-expression2 cannot be found	0
string-expression2 is null string	*start* (if given); otherwise, 1

Use the **LEN** function to find the length of *string-expression1*.

■ Example

```
'Program fragment illustrates use of INSTR combined with
'FNUCASE$ to create case-independent searches:

DEF FNUCASE$(X$)
    'See example under MID$ Statement for FNUCASE$ definition
END DEF

NAME$ = "X"
WHILE LEN(NAME$) < 3 : INPUT "Enter name: ", NAME$ : WEND
NAME$ = FNUCASE$(NAME$)
'Convert lowercase letters to uppercase:
IF INSTR(NAME$,"MS") > 0 OR INSTR(NAME$,"MRS") > 0 THEN
    SEX$ = "F"
ELSEIF INSTR(NAME$,"MR") > 0 THEN
    SEX$ = "M"
ELSE
    SEX$ = "X"
    WHILE SEX$ <> "M" AND SEX$ <> "F"
        INPUT "ENTER SEX (M/F): ", SEX$
        SEX$ = FNUCASE$(SEX$)
    WEND
END IF
.
.
.
```

■ Syntax

INT(*numeric-expression*)

■ Action

Returns the largest integer less than or equal to *numeric-expression*

■ Remarks

INT truncates the fractional part of its argument.

■ See Also

CINT, FIX

■ Example

The following example compares the output from the three functions that convert numeric data to integers:

```
PRINT "  N","INT(N)","CINT(N)","FIX(N)" : PRINT
FOR I% = 1 TO 6
    READ N
    PRINT N, INT(N), CINT(N), FIX(N)
NEXT
DATA  99.3, 99.5, 99.7, -99.3, -99.5, -99.7
```

Output:

```
  N           INT(N)        CINT(N)       FIX(N)

 99.3          99            99            99
 99.5          99            100           99
 99.7          99            100           99
-99.3         -100          -99           -99
-99.5         -100          -100          -99
-99.7         -100          -100          -99
```

IOCTL$ Function

■ Syntax

IOCTL$ ([[#]]*filenumber*)

■ Action

Receives a control data string from a device driver

■ Remarks

The **IOCTL$** function is most frequently used to receive acknowledgement that an **IOCTL** statement succeeded or failed, or to obtain current status information.

You could use **IOCTL$** to ask a communications device to return the current baud rate, information on the last error, logical line width, and so on.

The **IOCTL$** function works only if all three of the following conditions are met:

1. The device driver is installed
2. The device driver states that it processes **IOCTL** strings
3. BASIC performs an **OPEN** statement on a file on that device

■ See Also

IOCTL

■ Example

```
OPEN "\DEV\ENGINE" FOR OUTPUT AS #1
IOCTL #1, "RAW"          'Tells the device that the data is raw
'If the character driver "ENGINE" responds "false" from the
'raw data mode in the IOCTL statement, then the file is closed:
IF IOCTL$(1) = "0" THEN CLOSE 1
```

■ Syntax

IOCTL [[#]]*filenumber, string*

■ Action

Transmits a control character or string to a device driver

■ Remarks

IOCTL commands are generally two to three characters optionally followed by an alphanumeric argument. An **IOCTL$** command string can be up to 32,767 bytes long.

The **IOCTL** statement works only if all three of the following conditions are met:

1. The device driver is installed
2. The device driver states it processes **IOCTL** strings
3. BASIC performs an **OPEN** on a file on that device

Most standard DOS device drivers do not process **IOCTL** strings, and it is necessary for you to determine if the specific driver can handle the command.

■ See Also

IOCTL$

■ Example

If you wanted to set the page length to 66 lines per page on LPT1, your statement might look like this:

```
OPEN "\DEV\LPT1" FOR OUTPUT AS 1
IOCTL #1, "PL66"
```

■ Syntax

KEY *n, string-expression*
KEY LIST
KEY ON
KEY OFF

■ Action

Assigns soft-key string values to function keys and displays the values

■ Remarks

The **KEY** statement allows you to designate special "soft-key" functions. These are designated by assigning each of the function keys a string of up to 15 bytes that is input to BASIC when you press that key. The argument *n* is the number of the function key; *n* can have the values 1–10. The argument *string-expression* is the text assigned to the specified key. Only the first 15 characters of *string-expression* are assigned; if *string-expression* is longer than 15 characters, the extra characters are ignored.

You can display soft keys with the **KEY ON**, **KEY OFF**, and **KEY LIST** statements.

KEY ON causes the soft-key string values to be displayed on the bottom line of the screen.

KEY OFF erases the soft-key display from the bottom line, making that line available for program use. It does not disable the function keys.

KEY LIST displays all soft-key values on the screen, with all 15 characters of each key displayed.

Assigning a null string (string of length 0) to a soft key disables the function key as a soft key.

If the function-key number is not in the range of permissible function-key numbers, an "Illegal function call" error is displayed, and the previous key string expression is retained.

If a soft key is assigned, the **INKEY$** function returns one character of the soft-key string per invocation.

■ Example 1

```
KEY ON              'Displays the soft keys on bottom line of screen

KEY OFF             'Erases soft key display

KEY 1,"MENU"+CHR$(13)   'Assigns the string "MENU" followed
                        'by a carriage return to soft key 1

KEY LIST            'Displays all soft key values
KEY 1,""            'Disables soft key 1
```

■ Example 2

The following routine initializes the first five soft keys. (The first line turns off the key display during initialization; the second line displays the new soft-key values.)

```
KEY OFF
DATA EDIT, LET, SYSTEM ,PRINT, LPRINT
FOR I = 1 TO 5
    READ SOFTK$(I)
    KEY I,SOFTK$(I)
NEXT
KEY ON
```

KEY(*n*) Statements

■ Syntax

KEY(*n*) ON
KEY(*n*) OFF
KEY(*n*) STOP

■ Action

Activates or deactivates trapping of the specified key

■ Remarks

The argument *n* is the number of a function key, a cursor-direction key, or a user-defined key. (See **KEY** statements reference pages for information on assigning soft-key values to function keys.)

Value of *n*	Key
1–10	The function keys F1– F10
11	UP
12	LEFT
13	RIGHT
14	DOWN
15–25	User-defined keys

LEFT, RIGHT, UP, and DOWN refer to the DIRECTION keys.

User-defined keys are defined by the statement:

KEY *n*, **CHR$**(*keyboardflag*) + **CHR$**(*scancode*)

The *keyboardflag* can be any combination of the following hexadecimal values:

Keyboard Flag	Key
&H01–&H03	SHIFT
&H04	CTRL
&H08	ALT

The *scancode* argument is a number identifying one of the 83 keys to trap, as shown in Table 11.2.

Table 11.2
Keyboard Scan Codes

Key	Code in Hex	Key	Code in Hex	Key	Code in Hex
ESC	01	CTRL	1D	SPACE	39
! or 1	02	A	1E	CAPS LOCK	3A
@ or 2	03	S	1F	F1	3B
# or 3	04	D	20	F2	3C
$ or 4	05	F	21	F3	3D
% or 5	06	G	22	F4	3E
^ or 6	07	H	23	F5	3F
& or 7	08	J	24	F6	40
* or 8	09	K	25	F7	41
(or 9	0A	L	26	F8	42
) or 0	0B	: or ;	27	F9	43
_ or -	0C	" or '	28	F10	44
+ or =	0D	~ or `	29	NUM LOCK	45
LEFT	0E	LEFT SHIFT	2A	SCROLL LOCK	46
TAB	0F	| or \	2B	HOME or 7	47
Q	10	Z	2C	UP or 8	48
W	11	X	2D	PG UP or 9	49
E	12	C	2E	-	4A
R	13	V	2F	LEFT or 4	4B
T	14	B	30	5	4C
Y	15	N	31	RIGHT or 6	4D
U	16	M	32	+	4E
I	17	< or ,	33	END or 1	4F
O	18	> or .	34	DOWN or 2	50
P	19	? or /	35	PG DN or 3	51
{ or [	1A	RIGHT SHIFT	36	INS or 0	52
} or]	1B	PRT-SC or *	37	DEL or .	53
RETURN	1C	ALT	38		

Note that the **KEY** statement described previously assigns soft-key and cursor-direction values to function keys and displays these values. Do not confuse **KEY ON** and **KEY OFF**, which display and erase these values, with the event-trapping statements described in this section.

The **KEY**(n) **ON** statement enables soft-key or cursor-direction-key event trapping by an **ON KEY** statement. If you specify a nonzero line number in the **ON KEY** statement while trapping is enabled, BASIC checks between every statement to see if you have pressed **KEY**(n). If you have, it executes the **GOSUB** in the **ON KEY** statement. The text that would normally be associated with a function key is not printed.

KEY(n) **OFF** disables the event trap: even if an event takes place, it is not remembered.

KEY(n) **STOP** inhibits the event trap; that is, if you press the specified key, your action is remembered and an **ON KEY** statement is executed as soon as a **KEY**(n) statement is executed.

■ See Also

ON KEY

■ Example

```
I = 0
PRINT "Press DOWN direction key to end."
KEY 15,CHR$(&H04)+CHR$(&H1F)
KEY(15) ON                'CTRL-S key will now be trapped
KEY(14) ON                'DOWN key will noow be trapped
ON KEY(15) GOSUB KEYTRAP
ON KEY(14) GOSUB ENDPROG
IDLE: GOTO IDLE           'Endless loop

KEYTRAP:          'Counts the number of times CTRL-S pressed
    I = I + 1
RETURN

ENDPROG:
    PRINT "CTRL-S trapped" I "times"
    END
RETURN
```

KILL Statement

■ Syntax

KILL *filespec*

■ Action

Deletes a file from disk

■ Remarks

KILL is similar to the DOS **ERASE** or **DEL** commands. If a **KILL** statement is given for a file that is currently open, a "File already open" error results.

KILL is used for all types of disk files: program files, random data files, and sequential data files. The *filespec* may contain question marks (?) or asterisks (*) used as wild cards. A question mark matches any single character in the file name or extension. An asterisk matches one or more characters starting at its position.

You can use **KILL** only to delete files. To delete directories, use the **RMDIR** command.

Using **KILL** to delete a file that is currently open will result in a "File already open" error.

Warning

Be extremely careful when using wild cards with **KILL**; otherwise, you may inadvertently delete files that you had not intended to delete.

■ Example 1

The following examples show the effect of wild-card characters when used with **KILL**:

```
KILL "DATA1?.DAT"       'Kills any file that has a six-character base
                        'name starting with DATA1 and also has the
                        'extension .DAT

KILL "DATA1.*"          'Kills all files with the base name DATA1
                        'and any extension

KILL "..\GREG\*.DAT" 'Kills all files with the extension .DAT
                        'in a directory called GREG
```

■ Example 2

The following program deletes all files entered after the base name of the standalone executable file created from the following listing:

```
DEFINT A-Z
ON ERROR GOTO ERRORHANDLE
CLINE$ = COMMAND$       'Get command line, store in CLINE$
LENGTH = LEN(CLINE$)
MAX = (LENGTH/2) + 1
DIM ARG$(MAX)
GOSUB ARGSPLIT        'Get individual arguments on command line
FOR I = 0 TO NUM
    KILL ARG$(I)
NEXT
END

ARGSPLIT:
    'Subroutine to split command line (see example under
    'COMMAND$ Function for definition of ARGSPLIT
RETURN

ERRORHANDLE:
    NUMBER = ERR
    IF NUMBER = 53 THEN
        PRINT "Couldn't delete " ARG$(I);
        PRINT "; file does not exist in current directory"
    END IF
RESUME NEXT
```

LBOUND Function

■ Syntax

LBOUND(*array*[,*dimension*])

■ Action

Returns the lower bound (smallest available subscript) for the indicated dimension of an array

■ Remarks

LBOUND is used with the **UBOUND** function to determine the size of an array, and takes the following arguments:

Argument	Description
array	The name of the array being dimensioned
dimension	An integer from 1 to the number of dimensions in *array*

In the array `ACCOUNT(A,B,C,D)`, `A` is dimension 1, `B` is dimension 2, `C` is dimension 3, and `D` is dimension 4. So the following statement:

```
LBOUND(ACCOUNT,1)
```

finds the smallest subscript in dimension `A`,

```
LBOUND(ACCOUNT,2)
```

finds the smallest subscript in dimension `B`, and so on.

The default lower bound for any dimension is either 0 or 1, depending on the setting of the **OPTION BASE** statement. If **OPTION BASE** is 0, the default lower bound is 0, and if **OPTION BASE** is 1, the default lower bound is 1.

You can use the shortened syntax **LBOUND**(*array*) for one-dimensional arrays, since the default value for *dimension* is 1.

Use the **UBOUND** function to find the upper limit of an array dimension.

■ See Also

UBOUND

■ Example

LBOUND and **UBOUND** can be used together to determine the size of an array passed to a subprogram, as in the following program fragment:

```
CALL PRNTMAT(ARRAY())
.
.
.
SUB PRNTMAT(A(2)) STATIC
    FOR I% = LBOUND(A,1) TO UBOUND(A,1)
        FOR J% = LBOUND(A,2) TO UBOUND(A,2)
           PRINT A(I%,J%);" ";
        NEXT J%
    PRINT:PRINT
    NEXT I%
END SUB
```

LEFT$ Function

■ Syntax

LEFT$(*string-expression*,*n*)

■ Action

Returns a string comprising the left-most *n* characters of *string-expression*

■ Remarks

The integer argument *n* must be in the range 0 to 32,767.

The argument *string-expression* can be any string variable, string constant, or string expression. If *n* is greater than the number of characters in *string-expression*, the entire string is returned. To find the number of characters in *string-expression*, use **LEN**(*string-expression*)

If $n = 0$, the null string (length zero) is returned.

■ See Also

MID$**, **RIGHT$

■ Example

This prints the left-most 5 characters of A$:

```
A$="BASIC LANGUAGE"
B$=LEFT$(A$,5)
PRINT B$
```

Output:

```
BASIC
```

■ Syntax

LEN(*string-expression*)

■ Action

Returns the number of characters in *string-expression*

■ Remarks

The argument *string-expression* can be any string variable, string constant, or string expression. Nonprinting characters and blanks are counted.

■ Example

```
'Note: LEN in following line is part of OPEN syntax, and is not
'to be confused with the LEN function.
OPEN "ITEMS" FOR RANDOM AS #1 LEN = 14
FIELD #1, 10 AS NAME$, 4 AS COST$
TOTAL = LOF(1)/14

ENTERNAME:      'Continue prompting for item name if length of
                'input string is greater than 10 characters
    INPUT "ENTER ITEM NAME:    ", ITEMNAME$
    IF LEN(ITEMNAME$) > 10 THEN
        PRINT "   ITEM NAME MUST BE 10 CHARACTERS OR LESS."
        GOTO ENTERNAME
    END IF
ENTERCOST:      'Continue prompting for cost if length of
                'input string is greater than 4 characters
    INPUT "ENTER COST: ", ITEMCOST$
    IF LEN(ITEMCOST$) > 4 THEN
        PRINT "   ITEM COST MUST BE 4 CHARACTERS OR LESS."
        GOTO ENTERCOST
    END IF
    LSET NAME$ = ITEMNAME$
    LSET COST$ = ITEMCOST$
    TOTAL = TOTAL + 1
    PUT #1, TOTAL       'Write new record to ITEMS file
    PRINT : PRINT"MORE DATA?"
    'Wait for key press:
    R$ = "" : WHILE R$ = "" : R$ = INKEY$ : WEND
    IF R$ = "y" OR R$ = "Y" THEN GOTO ENTERNAME ELSE END
```

LET Statement

■ Syntax

[[**LET**]] *variable=expression*

■ Action

Assigns the value of an expression to a variable

■ Remarks

Notice that the word **LET** is optional; that is, the equal sign is sufficient for assigning an expression to a variable name.

■ Example

Corresponding lines perform the same function in these two examples:

```
LET D=12
LET E=12-2
LET F=12-4
LET SUM=D+E+F
.
.
.
```

or

```
D=12
E=12-2
F=12-4
SUM=D+E+F
.
.
.
```

■ Syntax

LINE [[**STEP**] (*x1,y1*)]-[**STEP**] (*x2,y2*) [,[*color*][,B[F]]][,*style*]

■ Action

Draws a line or box on the screen

■ Remarks

The coordinates (*x1,y1*) and (*x2,y2*) specify the end points of the line; note that the order in which these appear is unimportant, since a line from (10,20) to (120,130) is the same as a line from (120,130) to (10,20).

The **STEP** option makes the specified coordinates relative to the "most recent point," instead of absolute, mapped coordinates. For example, if the most recent point referred to by the program is (10,10), then

```
LINE STEP (10,5)
```

refers to a point with x-coordinate 10 + 10, and y-coordinate 10 + 5, or (20,15).

If the **STEP** option is used for the second coordinate in a **LINE** statement, it is relative to the first coordinate in the statement. Other ways to establish a new most recent point are to initialize the screen with the **CLS** and **SCREEN** statements. Using the **PSET**, **PRESET**, **CIRCLE** and **DRAW** statements will also establish a new most recent point.

The *color* is the number of the color in which the line is drawn. (If the **B** or **BF** options are used, the box is drawn in this color.) See the **SCREEN** statement reference page for information on valid colors.

The **B** option draws a box with the points (*x1,y1*) and (*x2,y2*) specifying diagonally opposite corners.

The **BF** option draws a filled box. This option is similar to the **B** option; **BF** also paints the interior of the box with the selected color.

The *style* is a 16-bit integer mask used to put pixels on the screen. This is called "line styling." With line styling, **LINE** reads the bits in *style* from left to right. If a bit is 0, then no point is plotted; if the bit is 1, a point is plotted. After plotting a point, **LINE** selects the next bit position in *style*.

Since a 0 bit in *style* causes no change to the point on the screen, you may prefer to draw a background line before a "styled" line in order to force a known background. Style is used for normal lines and boxes, but has no effect on filled boxes.

When coordinates specify a point that is not in the current viewport, the line segment is clipped to the viewport.

■ See Also

SCREEN

■ Examples

The following examples are different LINE statements that assume a screen 320 pixels wide by 200 pixels high:

```
LINE-(X2,Y2)                'Draws a line in the foreground color from
                            'the most recent point to  X2,Y2

LINE(0,0)-(319,199)         'Draws a diagonal line across the screen
                            '(downward)

LINE(0,100)-(319,100)       'Draws a horizontal line across the screen

LINE(10,10)-(20,20),2       'Draws a line in color 2

FOR X=0 to 319                      'Draws an alternating line
    LINE(X,0)-(X,199),X AND 1       'on/line off pattern on a
NEXT                                'monochrome display

LINE (0,0)-(100,100),,B         'Draws a box in the foreground
                                '(note that the color is not included)

LINE STEP(0,0)-STEP(200,200),2,BF    'Draws a filled box in color 2
                                     '(coordinates are given as
                                     'offsets with the STEP option)

LINE(0,0)-(160,100),3,,&HFF00    'Draws a dashed line from the upper
                                 'left hand corner to the center of
                                 'the screen
```

■ Syntax

LINE INPUT[;] ["*promptstring*";] *stringvariable*

■ Action

Inputs an entire line (up to 254 characters) to a string variable, without the use of delimiters

■ Remarks

The *promptstring* is a string literal that is displayed on the screen before input is accepted. A question mark is not printed unless it is part of *promptstring*. All input from the end of *promptstring* to the carriage return is assigned to *stringvariable*.

If **LINE INPUT** is immediately followed by a semicolon, then the carriage return that you enter to end the input line does not echo a carriage-return–line-feed sequence on the screen.

LINE INPUT uses the same line editor used by **INPUT**.

■ See Also

INPUT

■ Example

This program allows the user to enter text into a file. The **LINE INPUT** allows the user to enter any characters, including those (such as a comma) normally considered delimiters in a regular **INPUT** statement.

```
ENTER:
    CLS
    PRINT "Enter text. To stop, press <RETURN> without ";
    PRINT "entering any new text." : PRINT
    INLINE$ = "XX"
    OPEN "NOTES.TXT" FOR OUTPUT AS #1
    WHILE INLINE$ <> ""
        LINE INPUT INLINE$
        IF INLINE$ <> "" THEN PRINT #1, INLINE$
    WEND
    CLS : CLOSE #1
    OPEN "NOTES.TXT" FOR INPUT AS #1
    PRINT "You entered: " : PRINT
    WHILE NOT EOF(1)
        LINE INPUT #1, INLINE$
        PRINT INLINE$
    WEND
    CLOSE #1
    PRINT : INPUT "Is this correct (Y/N)"; R$
    IF  R$ = "n" OR R$ = "N" THEN GOTO ENTER ELSE END
```

■ Syntax

LINE INPUT# *filenumber,stringvariable*

■ Action

Reads an entire line without delimiters, from a sequential disk data file to a string variable

■ Remarks

The *filenumber* is the number under which the file was opened. The *stringvariable* is the variable name to which the line will be assigned.

LINE INPUT# reads all characters in the sequential file up to a carriage return. It then skips over the carriage-return–line-feed sequence. The next **LINE INPUT#** reads all characters up to the next carriage return. If a carriage-return–line-feed sequence is encountered, it is preserved. In other words, the carriage-return–line-feed characters are returned as part of the string.

LINE INPUT# is especially useful if each line of a data file has been broken into fields, or if a BASIC program saved in ASCII format is being read as data by another program.

■ See Also

INPUT$, LINE INPUT

■ Example

```
OPEN "LIST" FOR OUTPUT AS #1
PRINT "CUSTOMER INFORMATION:"
INFO:           'Write records to file with PRINT#
    INPUT "   LAST NAME:  ", LN$
    INPUT "   FIRST NAME: ", FN$
    INPUT "   AGE:        ", A$
    INPUT "   SEX:        ", S$
    PRINT #1, LN$, FN$, A$, S$
    INPUT "Add another"; R$
    IF R$ = "y" OR R$ = "Y" THEN GOTO INFO ELSE CLOSE #1
OPEN "LIST" FOR INPUT AS #1
CLS
PRINT "Records in file:" : PRINT
WHILE NOT EOF(1)
    LINE INPUT #1, REC$ 'Read records from file with LINE INPUT#
    PRINT REC$          'Print the records on the screen
WEND
```

Sample input:

```
CUSTOMER INFORMATION:
    LAST NAME:  Blair
    FIRST NAME: Shelley
    AGE:        24
    SEX:        F
Add another? y
    LAST NAME:  Hunt
    FIRST NAME: Kym
    AGE:        28
    SEX:        M
Add another? n
```

Sample output:

```
Records in file:

Blair         Shelley        24          F
Hunt          Kym            28          M
```

■ Syntax

LOC(*filenumber*)

■ Action

Returns the current position within the file

■ Remarks

With random-access disk files, **LOC** returns the actual number of the last record read from or written to the file. With sequential files opened for **APPEND** or **OUTPUT**, **LOC** returns the current byte position in the file, divided by 128.

The *filenumber* is the number under which the file was opened.

For a **COM** device, **LOC**(*filenumber*) is used to determine if there are any characters in the input queue waiting to be read. If there are more than 255 characters in the queue, **LOC**(*filenumber*) returns 255.

If fewer than 255 characters remain in the queue, the value returned by **LOC**(*filenumber*) depends on whether the device was opened in ASCII or binary mode. In either mode, **LOC** will return the number of characters that can be read from the device. However, in ASCII mode, the low level routines stop queuing characters as soon as end-of-file is received. The end-of-file itself is not queued and cannot be read. An attempt to read the end-of-file will result in an "Input past end" error.

■ See Also

OPEN COM

■ Example

```
IF LOC(1) > 50 THEN STOP
```

■ Syntax

LOCATE[[*row*][,[*column*][,[*cursor*][,[*start*][,*stop*]]]]]

■ Action

Moves the cursor to the specified position

■ Remarks

The *row* is a line number (vertical) on the screen; *row* is a numeric expression returning an unsigned integer. If *row* is not specified, then the current line number does not change.

The *column* is the column number on the screen; *column* is a numeric expression returning an unsigned integer. If *column* is not specified, then the current column number does not change.

Optional parameters turn the blinking cursor on and off, and define the vertical start and stop lines.

The argument *cursor* is a Boolean value indicating whether the cursor is visible or not. A value of 0 (zero) indicates cursor *off*; a value of 1 indicates cursor *on*.

The *start* is the cursor starting line (vertical) on the screen. It must be a numeric expression returning an unsigned integer.

The argument *stop* is the cursor stop line (vertical) on the screen. It must be a numeric expression returning an unsigned integer.

Any value outside the specified ranges will result in an "Illegal function call" error. In this case, current values are retained.

You may omit any parameter from this statement. If you do this, the program assumes the previous value for the parameter.

Note that the *start* and *stop* lines are the CRT scan lines that specify which pixels on the screen are lit. A wider range between the start and stop lines will produce a taller cursor, such as one that occupies an entire character block.

If the *start* line is given but the *stop* line is omitted, *stop* assumes the same value as *start*.

The last line on the screen is reserved for the soft-key display and is not accessible to the cursor unless the soft-key display is off and **LOCATE** is used to get to it.

■ See Also

CSRLIN, POS

■ Example 1

```
LOCATE 1,1     'Moves cursor to upper-left corner of the screen

LOCATE,,1      'Makes the cursor visible; position remains unchanged

LOCATE,,,7     'Position and cursor visibility remain unchanged;
               'sets the cursor to display at the bottom of
               'the character starting and ending on raster line 7

LOCATE 5,1,1,0,7      'Moves the cursor to line 5, column 1;
                      'turns cursor on; cursor covers entire
                      'character cell starting at scan line
                      '0 and ending on scan line 7
```

■ Example 2

The following example prints a menu on the screen, then waits for input in the allowable range (1–8); if a number outside that range is entered, the program continues to prompt for a selection.

```
ENDPROG = 0
WHILE NOT ENDPROG
    CLS
    PRINT "MAIN MENU" : PRINT
    PRINT "1)  Add Records"
    PRINT "2)  Display/Update/Delete a Record"
    PRINT "3)  Print Out List of People Staying at Hotel"
    PRINT "4)  Print Out Complete List of Everyone Attending"
    PRINT "5)  Print Out Workshops and Names of People Attending"
    PRINT "6)  Print Out Name Tags for Everyone Attending"
    PRINT "7)  Print Mailing List Sorted by Zip Codes"
    PRINT "8)  End Program"
    CH$ = "XX"
    WHILE (CH$ < "1" OR CH$ > "8")
       LOCATE 12,1 : PRINT "What is your selection?"
       CH$ = INPUT$(1)
    WEND
ON VAL(CH$) GOSUB Add,Search,Hotel,Printout,Shops,Tag,Mail,Quit
WEND
END
.
.
.
```

■ Syntax

LOCK [[#]] *filenumber* [[,{ *record* | [[*start*]] **TO** *end*}]]
.
.
.
UNLOCK [[#]] *filenumber* [[,{ *record* | [[*start*]] **TO** *end*}]]

■ Action

Controls access by other processes to all or part of an opened file

■ Remarks

These statements are used in networked environments where several processes might need access to the same file. **LOCK** and **UNLOCK** take the following arguments:

Argument	Description
filenumber	The number with which the file was opened.
record	The number of the record to be locked; *record* can be any number from 1–16,777,215 (2^{24}–1). A record may be up to 32,767 bytes in length.
start	The number of the first record to be locked.
end	The number of the last record to be locked.

If you specify just one record, then only that record is locked or unlocked. If you specify a range of records and omit a starting record (*start*), then all records from the first record to the end of the range (*end*) are locked or unlocked. **LOCK** with no record arguments locks the entire file, while **UNLOCK** with no record arguments unlocks the entire file.

If the file has been opened for *random* input or output, then the range indicates which records are to be locked or unlocked. However, if the file has been opened for *sequential* input or output, **LOCK** and **UNLOCK** affect the entire file, regardless of the range specified by *start* to *end*. **LOCK** and **UNLOCK** only function at run time if you are using versions of DOS that support networking (version 3.0 or later). In addition, you must run the **SHARE.EXE** program (or the network setup program must run it) to

perform any locking operation. Earlier versions of DOS will return an "Advanced Feature" error if **LOCK** and **UNLOCK** are executed.

Important

Be sure to remove all locks with an **UNLOCK** statement before closing a file or terminating your program; otherwise, unpredictable results will occur.

The arguments to **LOCK** and **UNLOCK** must match exactly.

If you attempt to access a file that is locked, the following error messages may appear:

```
Bad record number
Permission denied
```

■ Example 1

The following statement locks the entire file opened as number 2:

```
LOCK #2
```

The following statement locks only record 32 in file number 2:

```
LOCK #2, 32
```

The following statement locks records 1–32 in file number 2:

```
LOCK #2, to 32
```

The following statement locks records 9–32 in file number 2:

```
LOCK #2, 9 to 32
```

■ Example 2

The following two **UNLOCK** statements are legal:

```
LOCK #1, 1 TO 4
LOCK #1, 5 TO 8
UNLOCK #1, 1 TO 4
UNLOCK #1, 5 TO 8
```

The following two **LOCK** statements would be illegal, since the range in an **UNLOCK** statement must match the range in the corresponding **LOCK** statements exactly:

```
LOCK #1, 1 TO 4
LOCK #1, 5 TO 8
UNLOCK #1, 1 TO 8
```

■ Example 3

The following program fragment opens a file and allows a user to lock an individual record before updating the information in that record. When the user is done, the program unlocks the locked record. (Unlocking the locked records allows other processes to access the file.)

```
OPEN "MONITOR" SHARED AS #1 LEN = 59
FIELD #1,15 AS PAYER$,20 AS ADDRESS$,20 AS PLACE$,4 AS OWE$
LET UPDATE$ = "Y"
WHILE (UPDATE$ = "Y" OR UPDATE$ = "y")
   CLS:LOCATE 10,10
   INPUT "Customer Number?   #"; NUMBER%
   LOCK #1, NUMBER%
      GET #1, NUMBER%
      LET DOLLARS! = CVS(OWE$)
      LOCATE 11,10: PRINT "Customer: ";PAYER$
      LOCATE 12,10: PRINT "Address:  ";ADDRESS$
      LOCATE 13,10: PRINT "Currently owes: $";DOLLARS!
      LOCATE 15,10: INPUT "Change (+ or -)", CHANGE!
      LET DOLLARS! = DOLLARS! + CHANGE!
      LSET OWE$ = MKS$(DOLLARS!)
      PUT #1, NUMBER%
   UNLOCK #1, NUMBER%
   LOCATE 17,10: INPUT "Update another? ", CONTINUE$
   LET UPDATE$ = LEFT$(CONTINUE$,1)
WEND
```

LOF Function

■ Syntax

LOF(*filenumber*)

■ Action

Returns the length of the named file in bytes

■ Remark

When a file is opened for **APPEND** or **OUTPUT**, **LOF** returns the size of the file in bytes.

■ Example

The following example uses **LOF** to calculate the total number of records in a random-access file, then uses that information to assign record numbers when writing new data to the file with **PUT**:

```
OPTION BASE 1
DEFINT A-Z
DIM LABEL$(5), BUFFER$(5)
GOSUB OPENFILE
'Calculate the total # of records in the file mail.dat by dividing
'the total # of bytes in the file by the bytes per record:
TOTAL = LOF(1)/RECLENGTH
'Read the prompts from the second DATA statement below:
FOR I = 1 TO 5 : READ LABEL$(I) : NEXT
GOSUB ADDARECORD
.
.
.
END

OPENFILE:
    'Open the file mail.dat and allocate variables in the buffer:
    OPEN "mail.dat" AS #1 LEN = 85       'Bytes per record = 85
    RECLENGTH = 0      'RECLENGTH will be 85 when all numeric values
                       'are read from the first DATA statement below
    FOR I = 1 TO 5
       READ SIZE
       FIELD #1, RECLENGTH AS OFFSET$, SIZE AS BUFFER$(I)
       RECLENGTH = RECLENGTH + SIZE
    NEXT
RETURN
```

```
ADDARECORD:
    CLS
    FOR I = 1 TO 5
       'Print the prompts and enter data:
       PRINT LABEL$(I) ":  "; : LINE INPUT TEMP$
       'Left-justify data in field variables:
       LSET BUFFER$(I) = TEMP$
    NEXT
    TOTAL = TOTAL + 1      'Make the record number one more than
    PUT #1, TOTAL          'current # of records in the file, and
    INPUT "Add another";CH$  'store
    IF CH$ = "y" THEN GOTO ADDARECORD
RETURN

'Data for field variable sizes:
DATA 15,10,40,5,15
'Data for input prompts:
DATA Last name,First name, Address,Zip, Phone #
```

LOG Function

■ Syntax

LOG(*n*)

■ Action

Returns the natural logarithm of *n*

■ Remarks

The *n* argument must be greater than zero.

The natural logarithm is to the base *e*. The number *e* is approximately equal to 2.718282.

LOG is calculated with single-precision accuracy, unless the argument *x* is a double-precision value, in which case **LOG** is calculated with double-precision accuracy.

■ Example 1

```
PRINT LOG(45/7)
```

Output:

```
 1.860752
```

■ Example 2

```
E = EXP(1)          'Calculate the base for natural logs
PRINT E,
FOR I = 1 TO 3
    PRINT LOG(E^I),
NEXT
```

Output:

```
 2.718282        1        2        3
```

■ Syntax

LPOS(*n*)

■ Action

Returns the current position of the line-printer's print head within the printer buffer

■ Remarks

The argument *n* is the index of the printer being tested. For example, `LPT1:` would be tested with `LPOS(1)`, `LPT2:` with `LPOS(2)`, and so on.

LPOS does not necessarily give the physical position of the print head because it does not expand tab characters.

■ Example

The following program prompts the user for team names and the names of players on each team. It then prints the players and their teams on the printer.

```
LPRINT"Team Members"; TAB(76); "TEAM" : LPRINT
INPUT "How many teams"; TEAMS
INPUT "How many players per team";PPT
PRINT
FOR T = 1 TO TEAMS
    INPUT "Team name: ", TEAM$
    FOR P = 1 TO PPT
        INPUT "   Enter player name: ", PLAYER$
        LPRINT PLAYER$;
        IF P < PPT THEN
            IF LPOS(0) > 55 THEN      'Print a new line if print
                                      'head past column 55;
                LPRINT : LPRINT "      ";
            ELSE
                LPRINT ", ";          'otherwise, print a comma
            END IF
        END IF
    NEXT P
    LPRINT STRING$(80-LPOS(0)-LEN(TEAM$),"."); TEAM$
NEXT T
```

LPRINT, LPRINT USING Statements

■ Syntax

LPRINT [[*expressionlist*]] [[{;|,}]]
LPRINT USING *formatstring*; *expressionlist* [[{;|,}]]

■ Action

Prints data on the printer **LPT1:**

■ Remarks

These statements function the same as **PRINT** and **PRINT USING**, except output goes to the line printer, and the file-number option is not permitted.

LPRINT assumes an 80-character-wide printer. This width can be changed with a **WIDTH LPRINT** *width* statement.

Warning

Since the **LPRINT** statement uses the **LPT1** printer device, you should not use **LPRINT** in a program that also contains an **OPEN LPT1** statement. Using these two statements together can lead to unpredictable results.

■ See Also

PRINT, PRINT USING, WIDTH

■ Syntax

LSET *stringvariable=string-expression*

■ Action

Moves data from memory to a random-access-file buffer (in preparation for a **PUT** statement) or left-justifies the value of a string in a string variable

■ Remarks

If *string-expression* requires fewer bytes than were fielded to *stringvariable*, **LSET** left-justifies the string in the field, and **RSET** right-justifies the string. (Spaces are used to pad the extra positions.) If the string is too long for the field, both **LSET** and **RSET** truncate characters from the right. Numeric values must be converted to strings before they are **LSET** or **RSET**.

Note

You may also use **LSET** or **RSET** with a nonfielded string variable to left-justify or right-justify a string in a given field. For example, the program lines:

```
A$=SPACE$(20)
RSET A$=N$
```

right-justify the string `N$` in a 20-character field. This can be very handy for formatting printed output.

■ See Also

MKD$, MKI$, MKS$, PUT, RSET

■ Example

The first line converts the single-precision numeric variable AMT to a 4-byte string and stores that string in A$, left-justified. The second line converts the integer numeric variable COUNT% to a 2-byte string and stores that string in D$, right-justified.

```
LSET A$ = MKS$(AMT)
RSET D$ = MKI$(COUNT%)
```

■ Syntax

MID$(*string-expression,n*[,*length*])

■ Action

Returns a string of *length* characters from *string-expression,* beginning with the *n*th character

■ Remarks

The arguments *n* and *length* must be in the range 1 to 32,767. If *length* is omitted or if there are fewer than *length* characters to the right of the *n*th character, all right-most characters beginning with the *n*th character are returned. The argument *string-expression* can be any string variable, string constant, or string expression. If *n* is greater than the number of characters in *string-expression,* **MID$** returns a null string. (Use the **LEN** function to find the number of characters in *string-expression.*)

■ See Also

LEFT$**, **MID$ (Statement), **RIGHT$**

■ Example

```
INPUT "Binary number = ",BINARY$    'Input binary number as string
LENGTH = LEN(BINARY$)               'Get length of string
DECIMAL = 0
FOR K = 1 TO LENGTH
    'Get individual digits from string from left to right:
    DIGIT$ = MID$(BINARY$,K,1)
    'Convert digit characters to numbers:
    DECIMAL = 2*DECIMAL + VAL(DIGIT$)
NEXT
PRINT "Decimal number =" DECIMAL
```

Sample output:

```
Binary number = 10110
Decimal number = 22
```

MID$ Statement

■ Syntax

MID$(*stringvariable*,*n*[,*length*])=*string-expression*

■ Action

Replaces a portion of a string variable with another string

■ Remarks

The arguments *n* and *length* are integer expressions. The argument *stringvariable* is a string variable, but *string-expression* can be a string variable, string constant, or string expression.

The characters in *stringvariable*, beginning at position *n*, are replaced by the characters in *string-expression*. The optional *length* refers to the number of characters from *string-expression* that are used in the replacement. If *length* is omitted, all of *string-expression* is used. However, regardless of whether *length* is omitted or included, the replacement of characters never goes beyond the original length of *stringvariable*.

■ See Also

MID$ Function

■ Example

The following multiline function uses both the MID$ function (to get a character from a string), and the MID$ statement (to replace that character with an uppercase letter, if necessary):

```
DEF FNUCASE$(X$)          'Define a function to convert lowercase
    LENGTH = LEN(X$)      'to uppercase in string X$
    'Exit if null string:
    IF LENGTH = 0 THEN FNUCASE$ = "" : EXIT DEF
    FOR I = 1 TO LENGTH
        CH = ASC(MID$(X$,I,1))      'Get a character from string
        IF CH > 96 AND CH < 127 THEN   'If lowercase, then
            MID$(X$,I,1) = CHR$(CH-32) 'replace with uppercase
        END IF                         'counterpart
    NEXT
    FNUCASE$ = X$         'Return string
END DEF
```

■ Syntax

MKDIR *pathname*

■ Action

Creates a new directory

■ Remarks

The *pathname* is a string expression specifying the name of the directory to be created. The *pathname* must be a string of less than 128 characters.

MKDIR works like the DOS command **MKDIR**; the syntax in BASIC cannot, however, be shortened to **MD**, as in DOS.

■ See Also

CHDIR, RMDIR

■ Example

```
ON ERROR GOTO ERRORHANDLER
PRINT "This program will create a new directory named MONTHS"
PRINT "in this directory, then create files in that directory"
MKDIR "MONTHS"
START:
    INPUT "Filename"; FILE$
    IF FILE$ = "" THEN END
    OPEN "MONTHS\" + FILE$ FOR OUTPUT AS #1
    .
    .
    .
    CLOSE #1
    GOTO START

ERRORHANDLER:
    'Error 75 means MONTHS directory already exists
    IF ERR = 75 THEN RESUME NEXT
    ON ERROR GOTO 0
```

MKD$, MKI$, MKS$ Functions

■ Syntax

MKI$(*integer-expression*)
MKS$(*single-precision-expression*)
MKD$(*double-precision-expression*)

■ Action

Converts numeric values to string values

■ Remarks

Any numeric value that is placed in a random-access file buffer with an **LSET** or **RSET** statement must be converted to a string. **MKI$** converts an integer to a 2-byte string. **MKS$** converts a single-precision number to a 4-byte string. **MKD$** converts a double-precision number to an 8-byte string.

These functions differ from **STR$** in that they do not really change the data bytes, just the way BASIC interprets those bytes.

■ See Also

CVI, CVS, CVD

■ Example

Line 100 of this example converts the single-precision value in the variable AMT to a 2-byte string which is part of FIELD:

```
    AMT=(K+T)
    FIELD #1,8 AS D$,20 AS N$
100 LSET D$=MKS$(AMT)
    LSET N$=A$
    PUT #1
    .
    .
    .
```

■ Syntax

NAME *oldfilename* **AS** *newfilename*

■ Action

Changes the name of a disk file

■ Remarks

NAME is similar to the DOS **RENAME** command. **NAME**, however, can also be used to move a file from one directory to another—a capability that **RENAME** does not have.

The *oldfilename* must exist and the *newfilename* must not exist; otherwise, an error will result. Also, both files must be on the same drive.

You may not rename a file with a new drive designation. If you attempt this, a "Rename across disks" error will be generated. After a **NAME** statement, the file exists on the same disk, in the same disk space, but with the new name.

You cannot use **NAME** to rename directories.

You must close the file you wish to rename before using **NAME**.

■ See Also

MKDIR, **RMDIR**

■ Examples

```
'Changes the name of file ACCTS to LEDGER:
NAME "ACCTS" AS "LEDGER"

'Moves file CLIENTS from directory X to directory \XYZ\P:
NAME "\X\CLIENTS" AS "\XYZ\P\CLIENTS"
```

OCT$ Function

■ Syntax

OCT$(*numeric-expression*)

■ Action

Returns a string that represents the octal value of the decimal argument *numeric-expression*. The argument *numeric-expression* is rounded to an integer before **OCT$**(*numeric-expression*) is evaluated.

■ See Also

HEX$

■ Example

```
PRINT OCT$(24)
```

Output:

```
 30
```

■ Syntax

ON COM(*n*) **GOSUB** {*linenumber* | *linelabel*}

■ Action

Specifies the first line of a subroutine to be performed when characters are received at a communications port

■ Remarks

The *n* argument is the number of the communications port; *n* can have the value 1 or 2.

The *linenumber* or *linelabel* identifies the first line of a subroutine that is to be performed when activity occurs on the specified communications port. This number or label *must* be defined at the main level of the program; it cannot refer to a line inside a subroutine or subprogram. A *linenumber* of 0 is a special case: it does not branch to line number 0; instead, it disables the communications event trap.

The **ON COM** statement will be executed only if a **COM**(*n*) **ON** statement has previously been executed to enable event trapping for communications channel *n*. If event trapping is enabled, and if the *linenumber* in the **ON COM** statement is not zero, BASIC checks between statements to see if communications activity has occurred on the specified port. If communications activity has occurred, a **GOSUB** is performed that sends program control to the specified line.

If a **COM OFF** statement has been executed for the communications port, the **GOSUB** is not performed and is not remembered.

If a **COM STOP** statement has been executed for the communications port, the **GOSUB** is not performed, but will be performed as soon as a **COM ON** statement is executed.

When an event trap occurs (i.e., the **GOSUB** is performed), an automatic **COM STOP** is executed so that recursive traps cannot take place. The **RETURN** from the trapping subroutine will automatically perform a **COM ON** statement unless an explicit **COM OFF** was performed inside the subroutine.

The **RETURN** *linenumber* form of the **RETURN** statement may be used to return to a specific line number from the trapping subroutine. Use this type of return with care, however, because any other **GOSUB**, **WHILE**, or **FOR** statements that were active at the time of the trap will remain active, and errors such as "FOR without NEXT" may result.

Event trapping is automatically inhibited when an error trap occurs; that is, no errors can be trapped while program control is inside the event-trapping subroutine.

If your program is receiving data using an asynchronous communications adapter, the QuickBASIC command-line **/c** option can be used to set the size of the buffer storing the data. For more information, see Appendix D, "Using the Separate Compilation Method."

■ Compiler/Interpreter Differences

With the compiler, you must use the **/v** or **/w** option on the compiler command line if a program contains an **ON COM** statement. If you are compiling from the QuickBASIC menu, use either the Event Trapping or the Checking Between Statements compile option. These options allow the compiler to function correctly when event-trapping subroutines are included in a program.

■ Syntax

ON ERROR GOTO {*linenumber* | *linelabel*}

■ Action

Enables error handling and specifies the first line of the error-handling routine

■ Remarks

Once error handling has been enabled, all errors detected will cause a jump to the specified error-handling routine. This number or label *must* be defined at the main level of the program; it cannot refer to a line inside a subroutine or subprogram. A *linenumber* of 0 is a special case: it does not branch to line-number 0; rather, it disables all error detecting. Subsequent errors print an error message and halt program execution. An `ON ERROR GOTO 0` statement that appears in an error-handling routine halts program execution and prints the error message for the error that caused the trap. It is recommended that all error-handling routines execute an `ON ERROR GOTO 0` if an error is encountered for which there is no recovery action.

Note

If an error occurs during execution of an error-handling routine, an error message is printed and execution terminates. Error trapping does not occur within the error-handling routine.

■ Compiler/Interpreter Differences

With the compiler, you must use the On Error (or **/e**) compile option if a program contains **ON ERROR GOTO** and **RESUME** {*linenumber* | *linelabel*} statements. If a program contains **RESUME**, **RESUME NEXT**, or **RESUME 0**, then you must use the Resume Next (or **/x**) compile option. If you do not use either of these options, at least one integer line number must appear in the program before the statement that causes the error.

Regardless of the option you use, the compiler supports the use of line labels as the objects of **RESUME** and **ON ERROR GOTO** statements. The purpose of these options is to allow your program to function correctly when error-handling routines are included in a program. Note that the use of these options increases the size of **.OBJ** and **.EXE** files.

Note

The line specified in a **RESUME** [*linenumber* | *linelabel*] statement must be defined at the level of the program that caused the error. For example, if the error occurs in a subroutine, then *linenumber* or *linelabel* must refer to a line in that subroutine.

■ See Also

ERL, ERR, ERROR, RESUME

■ Example

```
ON ERROR GOTO 1000
```

■ Syntax

ON *expression* **GOTO** { *line-number-list* | *line-label-list*}
ON *expression* **GOSUB** { *line-number-list* | *line-label-list*}

■ Action

Branches to one of several specified lines, depending on the value returned when an expression is evaluated

■ Remarks

The *expression* argument can be any numeric expression (*expression* is rounded to an integer before the **ON...GOSUB** or **ON...GOTO** is evaluated). The *line-number-list* or *line-label-list* consists of a list of line numbers or line labels, separated by commas. Each of these numbers or labels identifies the first line of a subroutine; the value of *expression* determines which of these subroutines program control will branch to. For example, if the value is 3, the third line specified in the list is the destination of the branch.

The value of *expression* should be $\geq$ 1 and $\leq$ the number of items in the list; if the value falls outside this range, one of the following results occurs:

Expression Value	Result
0 or > number of items in list	Control drops to the next BASIC statement
Negative or > 255	"Illegal function call" error

It is permissible to mix numbers and alphanumeric labels in the same list.

Warning

Entering or exiting **SUB...END SUB** blocks with **GOSUB...RETURN** statements is not supported in BASIC.

■ See Also

GOSUB, RETURN

■ Example

The following program fragment causes program control to branch to one of the four subroutines listed, depending on the value of CHVAL:

```
CHVAL = -1
WHILE CHVAL
    CLS
    PRINT "1) Display attendance at workshops."
    PRINT "2) Calculate total registration fees paid."
    PRINT "3) Print mailing list."
    PRINT "4) End program."
    PRINT : PRINT "What is your choice?"
    CH$ = "" : WHILE CH$ = "" : CH$ = INKEY$ : WEND
    CHVAL = VAL(CH$)
    IF CHVAL > 0 AND CHVAL < 5 THEN
        ON CHVAL GOSUB SHOP, FEES, MAILER, PROGEND
    END IF
WEND
END
.
.
.
```

■ Syntax

ON KEY(*n*) **GOSUB** { *linenumber* | *linelabel*}

■ Action

Specifies the first line of a subroutine to be performed when the given key is pressed

■ Remarks

The argument *n* is the number of a FUNCTION key, DIRECTION key, or user-defined key (see **KEY**(*n*) statement reference page for the values corresponding to these different keys).

The *linenumber* or *linelabel* specifies the first line of a subroutine that is performed when you press the specified function- or cursor-direction key. This number or label *must* be defined at the main level of the program; it cannot refer to a line inside a subroutine or subprogram. A *linenumber* of 0 is a special case: it does not branch to line number 0; instead, it disables key event trapping.

The **ON KEY** statement is executed only if a **KEY**(*n*) **ON** statement has previously been executed to enable event trapping. If key trapping is enabled, and if the *linenumber* in the **ON KEY** statement is not zero, BASIC checks between statements to see if you have pressed the specified FUNCTION, user-defined, or DIRECTION key. If you have, the program branches to a subroutine specified by the **GOSUB** statement.

If the key has been turned off with a **KEY**(*n*) **OFF** statement, the **GOSUB** is not performed and is not remembered.

If a **KEY STOP** statement has been executed for the specified key, the **GOSUB** is not performed, but will be performed as soon as a **KEY**(*n*) **ON** statement is executed.

When an event trap occurs (i.e., the **GOSUB** is performed), an automatic **KEY**(*n*) **STOP** is executed so that recursive traps cannot take place. The **RETURN** from the trapping subroutine automatically performs a **KEY**(*n*) **ON** statement unless an explicit **KEY**(*n*) **OFF** was performed inside the subroutine.

You may use the **RETURN** *linenumber* form of the **RETURN** statement to return to a specific line number from the trapping subroutine. Use this type of return with care, however, because any other **GOSUB**, **WHILE**, or **FOR** statements that were active at the time of the trap will remain active, and errors such as "FOR without NEXT" may result.

Event trapping is automatically inhibited when an error trap occurs; that is, no errors can be trapped while program control is inside the event-trapping subroutine.

The following four rules apply to keys trapped by BASIC:

1. The line printer's echo-toggle key is processed first. Defining this key as a user-defined-key trap will not prevent characters from being echoed to the line printer when depressed.
2. Function keys and the cursor-direction keys are examined next. Defining a FUNCTION key or DIRECTION key as a user-defined-key trap will have no effect, as these keys are considered predefined.
3. Finally, the user-defined keys are examined.
4. Any key that is trapped is not passed on. That is, the key is not read by BASIC.

The **ON KEY** statement can apply to any key, including BREAK or system reset. This is a powerful feature when you consider that it is now possible to prevent accidentally breaking out of a program, or rebooting the machine.

Note

When a key is trapped, the key event is destroyed. Therefore, you cannot subsequently use the **INPUT** or **INKEY$** statements to find out which key caused the trap. So if you wish to assign different functions to particular keys, you must set up a different subroutine for each key, rather than assigning the various functions within a single subroutine.

The **ON KEY(***n***)** statement allows six additional user-defined **KEY** traps. This allows any key, control-key, shift-key, or super-shift-key to be trapped by the user as follows:

ON KEY(*n***) GOSUB** *linenumber*

where *n* is an integer from 15–20, expressing a legal user-defined key number. (For more information on user-defined keys, see the **KEY**(*n*) statement reference pages).

■ Compiler/Interpreter Differences

With the compiler, you must use the /**v** or /**w** option on the command line if a program contains an **ON KEY** statement. If you are compiling from the QuickBASIC menu, use either the Event Trapping or the Checking Between Statements compile option. These options allow the compiler to function correctly when event-trapping subroutines are included in a program.

■ Example

The following program allows you to scroll through a text file; you can suspend the scrolling by pressing the UP DIRECTION key, or end the program by pressing the DOWN DIRECTION key.

```
KEY(11) ON       'UP direction key will now be trapped.
KEY(14) ON       'DOWN direction key will now be trapped.
ON KEY(11) GOSUB SUSPEND
ON KEY(14) GOSUB STOPPROG
PRINT "If you want to pause, press the UP direction key;"
PRINT "press any other key to restart scrolling. To stop"
PRINT "scrolling, press the DOWN direction key." : PRINT
INPUT "Name of file to scan: ",FILE$
OPEN FILE$ FOR INPUT AS #1
ENDFILE = 0
WHILE NOT ENDFILE
    LINE INPUT #1, TEMP$         'Read a line from the file
    PRINT TEMP$                  'and print on the screen.
    IF EOF(1) THEN ENDFILE = -1
WEND
END

SUSPEND:              'Stop scrolling; wait for key press.
    CONTINUE$ = ""
    WHILE CONTINUE$ = "" : CONTINUE$ = INKEY$ : WEND
RETURN

STOPPROG:
    ENDFILE = -1      'This will terminate WHILE...WEND loop.
RETURN
```

ON PEN Statement

■ Syntax

ON PEN GOSUB { *linenumber* | *linelabel*}

■ Action

Specifies the first line of a subroutine to be performed when the lightpen is activated

■ Remarks

The *linenumber* or *linelabel* identifies the first line of a subroutine that is performed when you activate the lightpen. This number or label *must* be defined at the main level of the program; it cannot refer to a line inside a subroutine or subprogram. A *linenumber* of 0 is a special case: it does not branch to line number 0; instead, it disables pen-event trapping.

The **ON PEN** statement will only be executed if a **PEN ON** statement has previously been executed to enable event trapping. If event trapping is enabled, and if the *linenumber* in the **ON PEN** statement is not zero, BASIC checks between statements to see if the lightpen has been activated. When you activate the lightpen, a **GOSUB** is performed to the specified line.

If a **PEN OFF** statement has been executed, the **GOSUB** is not performed and is not remembered.

If a **PEN STOP** statement has been executed, the **GOSUB** is not performed, but will be performed as soon as a **PEN ON** statement is executed.

When an event trap occurs (i.e., the **GOSUB** is performed), an automatic **PEN STOP** is executed so that recursive traps cannot take place. The **RETURN** from the trapping subroutine automatically performs a **PEN ON** statement unless an explicit **PEN OFF** was performed inside the subroutine.

You can use the **RETURN** *linenumber* form of the **RETURN** statement to return to a specific line number from the trapping subroutine. Use this type of return with care, however, because any other **GOSUB**, **WHILE**, or **FOR** statements that were active at the time of the trap will remain active, and errors such as "FOR without NEXT" may result.

Event trapping is automatically inhibited when an error trap occurs; that is, no errors can be trapped while program control is inside the event-trapping subroutine.

■ Compiler/Interpreter Differences

With the compiler, you must use the /v or /w option on the command line if a program contains an **ON PEN** statement. If you are compiling from the QuickBASIC menu, use either the Event Trapping or the Checking Between Statements compile option. These options allow the compiler to function correctly when event-trapping subroutines are included in a program.

ON PLAY Statement

■ Syntax

ON PLAY (*queuelimit*) **GOSUB** {*linenumber* | *linelabel*}

■ Action

Branches to a specified subroutine when the music queue contains fewer than *queuelimit* notes

■ Remarks

This statement permits continuous music during program execution.

The *queuelimit* argument is an integer expression in the range 1–32. Values outside this range result in an "Illegal function call" error.

The *linenumber* or *linelabel* specifies the first line of the **PLAY** event trap subroutine. This number or label *must* be defined at the main level of the program; it cannot refer to a line inside a subroutine or subprogram. A *linenumber* of 0 is a special case: it does not branch to line number 0; instead, it disables play event trapping.

ON PLAY causes an event trap when the background music queue goes from *queuelimit* notes to *queuelimit* – 1 notes.

The **ON PLAY** statement is used with the **PLAY ON**, **PLAY OFF**, and **PLAY STOP** statements which enable, disable, and suspend **PLAY** event trapping, respectively.

If **PLAY ON** is executed, then a **GOSUB** is executed when a **PLAY** even occurs.

If **PLAY OFF** is executed, then no **GOSUB** is executed and the statement is not remembered.

If a **PLAY STOP** statement has been executed, the **GOSUB** is not performed, but will be performed as soon as a **PLAY ON** statement is executed.

When an event trap occurs (i.e., the **GOSUB** is performed), an automatic **PLAY STOP** is executed so that recursive traps cannot take place. The **RETURN** from the trapping subroutine automatically performs a **PLAY ON** statement unless an explicit **PLAY OFF** was performed inside the subroutine.

You may use the **RETURN** *linenumber* form of the **RETURN** statement to return to a specific line number from the trapping subroutine. Use this type of return with care, however, because any other **GOSUB**, **WHILE**, or **FOR** statement that is active at the time of the trap will remain active, and errors such as "FOR without NEXT" may result.

The following three rules apply to the use of **ON PLAY**:

1. A play event trap occurs only when music is playing in the background. Play event traps do not occur when music is running in the foreground.
2. A play event trap does not occur if the background music queue has already gone from having n to $n - 1$ notes when a **PLAY ON** is executed.
3. If n is a large number, event traps will occur frequently enough to diminish program execution speed.

■ See Also

PLAY ON, PLAY OFF, PLAY STOP

■ Example

In this example, control branches to a subroutine when the background music buffer decreases to 7 notes:

```
PLAY ON
.
.
PLAY "MB L1 XZITHER$"
ON PLAY(8) GOSUB BACKGROUND
.
.
.
BACKGROUND:      'Background music
COUNT% = COUNT% + 1
.
.
RETURN
```

ON STRIG Statement

■ Syntax

ON STRIG(*n*) **GOSUB** { *linenumber* | *linelabel*}

■ Action

Specifies the first line of a subroutine to be performed when the joystick trigger is pressed

■ Remarks

The argument *n* is the number of the joystick trigger.

The *linenumber* or *linelabel* specifies the first line of a subroutine that is performed when you press the joystick trigger. This number or label *must* be defined at the main level of the program; it cannot refer to a line inside a subroutine or subprogram. A *linenumber* of 0 is a special case: it does not branch to line number 0; instead, it disables joystick-trigger event trapping.

The **ON STRIG** statement is executed only if a **STRIG ON** statement has previously been executed to enable event trapping. If event trapping is enabled, and if the *linenumber* in the **ON STRIG** statement is not zero, BASIC checks between statements to see if you have pressed the joystick trigger. If you have, a **GOSUB** is performed to the specified line.

If a **STRIG OFF** statement has been executed, the **GOSUB** is not performed and is not remembered.

If a **STRIG STOP** statement has been executed, the **GOSUB** is not performed, but will be performed as soon as a **STRIG ON** statement is executed.

When an event trap occurs (i.e., the **GOSUB** is performed), an automatic **STRIG STOP** is executed so that recursive traps cannot take place. The **RETURN** from the trapping subroutine will automatically perform a **STRIG ON** statement unless an explicit **STRIG OFF** was performed inside the subroutine.

You may use the **RETURN** *linenumber* form of the **RETURN** statement to return to a specific line number from the trapping subroutine. Use this type of return with care, however, because any other **GOSUB**, **WHILE**, or **FOR** statements that were active at the time of the trap will remain active, and errors such as "FOR without NEXT" may result.

Event trapping is automatically inhibited when an error trap occurs; that is, no errors can be trapped while program control is inside the event-trapping subroutine.

■ Compiler/Interpreter Differences

With the compiler, you must use the /v or /w option on the command line if a program contains an **ON STRIG** statement. If you are compiling from the QuickBASIC menu, use either the Event Trapping or the Checking Between Statements compile option. These options allow the compiler to function correctly when event-trapping subroutines are included in a program.

ON TIMER Statement

■ Syntax

ON TIMER (*n*) **GOSUB** {*linenumber* | *linelabel*}

■ Action

Provides an event trap during real time

■ Remarks

ON TIMER causes an event trap every *n* seconds. The *n* argument must be a numeric expression in the range 1–86,400 (1 second to 24 hours). Values outside this range generate an "Illegal function call" error.

The *linenumber* or *linelabel* specifies the first line of a subroutine that is performed after *n* seconds have elapsed. This number or label *must* be defined at the main level of the program; it cannot refer to a line inside a subroutine or subprogram. A *linenumber* of 0 is a special case: it does not branch to line number 0; instead, it disables timer event trapping.

The **ON TIMER** statement is executed only if a **TIMER ON** statement has previously been executed to enable event trapping. If event trapping is enabled, and if the *linenumber* in the **ON TIMER** statement is not zero, BASIC checks between statements to see if the time has been reached. If it has, a **GOSUB** is performed to the specified line.

If a **TIMER OFF** statement has been executed, the **GOSUB** is not performed and is not remembered.

If a **TIMER STOP** statement has been executed, the **GOSUB** is not performed, but will be performed as soon as a **TIMER ON** statement is executed.

When an event trap occurs (i.e., the **GOSUB** is performed), an automatic **TIMER STOP** is executed so that recursive traps cannot take place. The **RETURN** from the trapping subroutine will automatically perform a **TIMER ON** statement unless an explicit **TIMER OFF** was performed inside the subroutine.

You may use the **RETURN** *linenumber* form of the **RETURN** statement to return to a specific line number from the trapping subroutine. Use this type of return with care, however, because any other **GOSUB**, **WHILE**, or **FOR** statements that were active at the time of the trap will remain active, and errors such as "FOR without NEXT" may result.

■ Compiler/Interpreter Differences

With the compiler, you must use the /v or /w option on the command line if a program contains an **ON TIMER** statement. If you are compiling from the QuickBASIC menu, use either the Event Trapping or the Checking Between Statements compile option. These options allow the compiler to function correctly when event-trapping subroutines are included in a program.

■ See Also

TIMER ON, TIMER OFF, TIMER STOP

■ Example

The following example draws a polygon with a random shape (3 to 7 sides), size, and location every three seconds:

```
SCREEN 1
DEFINT A-Z
DIM X(6), Y(6)
TIMER ON                          'Enable timer event trapping
ON TIMER(3) GOSUB DRAWPOLY        'Draw a new polygon every 3 seconds
PRINT "Press any key to end program"
INPUT "Press <RETURN> to start",TEST$
LOOP:
    TEST$ = INKEY$                'End program if any key pressed
    IF TEST$ <> "" THEN END ELSE GOTO LOOP

DRAWPOLY:
    CLS                           'Erase old polygon
    N = INT(5*RND + 2)            'N is random number from 2 to 6
    FOR I = 0 TO N
        X(I) = INT(RND*319)       'Get coordinates of vertices of
        Y(I) = INT(RND*199)       'polygon
    NEXT
    PSET (X(N),Y(N))
    FOR I = 0 TO N
        LINE -(X(I),Y(I)),2       'Draw new polygon
    NEXT
RETURN
```

■ Syntax

OPEN *file* [[**FOR** *mode1*]][[**ACCESS** *access*]] [[*lock*]] **AS** [[**#**]] *filenum* [[**LEN=***reclen*]]
OPEN *mode2*,[[**#**]]*filenum*, *file* [[,*reclen*]]

■ Action

Allows I/O to a file or device

■ Remarks

The *filespec* (*file*) is a string expression that specifies an optional device, followed by a file name or path name that conforms to DOS file-naming conventions.

In the first syntax, *mode1* is an expression that is one of the following:

Mode	Action
OUTPUT	Specifies sequential output mode.
INPUT	Specifies sequential input mode.
APPEND	Specifies sequential output mode and sets the file pointer at the end of file and the record number as the last record of the file. A **PRINT#** or **WRITE#** statement will then extend (append to) the file.
RANDOM	Specifies random-access file mode, which is the default file mode. In **RANDOM** mode, if no **ACCESS** clause is present, three attempts are made to open the file when the **OPEN** statement is executed. Access is attempted in the following order: 1. Read/write 2. Write-only 3. Read-only

If *mode1* is omitted, the default random-access mode is assumed.

The *access* expression following **ACCESS** specifies the type of operation to be performed on the opened file. If the file is already opened by another process and access of the type specified is not allowed, the **OPEN** will fail and a "Permission denied" error message is generated.

The *access* argument can be one of the following:

Access Type	Action
READ	Opens the file for reading only.
WRITE	Opens the file for writing only.
READ WRITE	Opens the file for both reading and writing. This mode is valid only for **RANDOM** files and files opened for **APPEND**.

The *locktype* (*lock*) clause works in a multiprocessing environment to restrict access by other processes to an open file. The lock types are as follows:

Lock Type	Action
default	If *locktype* is not specified, the file may be opened for reading and writing any number of times by this process, but other processes are denied access to the file while it is opened.
SHARED	Any process on any machine may read from or write to this file. (Do not confuse this with the **SHARED** statement, or the **SHARED** attribute to the **COMMON**, **DIM**, and **REDIM** statements.)
LOCK READ	No other process is granted read access to this file. This access is granted only if no other process has a previous **READ** access to the file.
LOCK WRITE	No other process is granted write access to this file. This lock is granted only if no other process has a previous **WRITE** access to the file.

LOCK READ WRITE	No other process is granted either read or write access to this file. This access is granted only if **READ** or **WRITE** access has not already been granted to another process, or a **LOCK READ** or **LOCK WRITE** is not already in place.

When the **OPEN** is restricted by a previous process, it generates error 70, "Permission Denied," under DOS.

The *filenumber* (*filenum*) is an integer expression whose value is between 1 and 255. The number is then associated with the file for as long as it is open and is used to refer other disk I/O statements to the file.

The *recordlength* (*reclen*) is an integer expression that, if included, sets the record length (number of characters in one record) for random-access files. The default length for records is 128 bytes; *recordlength* cannot exceed 32,767 bytes. For sequential files, *recordlength* need not correspond to an individual record size, since a sequential file may have records of different sizes. When used to open a sequential file, *recordlength* specifies the number of characters to be loaded to the buffer before it is written to, or read from, the disk. A larger buffer means more room taken from BASIC, but faster file I/O.

In the second form of the **OPEN** syntax, *mode2* is a string expression, the first character of which must be one of the following:

Mode	Action
O	Specifies sequential output mode.
I	Specifies sequential input mode.
R	Specifies random-access file input/output mode.
A	Specifies sequential output mode and sets the file pointer at the end of the file, and the record number as the last record of the file. A **PRINT#** or **WRITE#** statement will then extend (append to) the file.

Note

The second form of the **OPEN** syntax does not support any of the access and file-sharing options found in the first syntax, and is supported for compatibility with programs written in earlier versions of BASIC.

You must open a file before any I/O operation can be performed on that file. **OPEN** allocates a buffer for I/O to the file or device, and determines the mode of access that will be used with the buffer.

Since it is possible to refer to the same file in a subdirectory via different paths, it is nearly impossible for BASIC to know that it is the same file simply by looking at the path. For this reason, BASIC will not let you open the file for **OUTPUT** or **APPEND** if it is on the same disk, even if the path is different. For example, the following two statements will result in a "Permission denied" error, even though they refer to two distinct files:

```
OPEN "C:\TMP\REPORT" FOR APPEND AS #1
OPEN "C:\SALES\REPORT" FOR APPEND AS #2
```

BASIC Devices

The following devices are supported by BASIC and can be named and opened with the *filespec* argument:

KYBD:

SCRN:

COM*n***:**

LPT*n***:**

CONS:

The BASIC file I/O system allows you to take advantage of user-installed devices. (See your DOS manual for information on character devices.)

Character devices are opened and used in the same manner as disk files. However, characters are not buffered by BASIC as they are for disk files. The record length for the device files is set to one.

BASIC only sends a carriage return at the end of the line. If the device requires a line feed, the driver must provide it. When writing device drivers, keep in mind that other BASIC users will want to read and write control information. Writing and reading of device-control data is handled by the **IOCTL** statement and **IOCTL$** function.

Note

A file can be opened for sequential input or random access on more than one file number at a time. A file may be opened for output, however, on only one file number at a time.

■ Examples

The following example opens MAILING.DAT as file number 1, and allows data to be added without destroying what is already in MAILING.DAT:

```
OPEN "MAILING.DAT" FOR APPEND AS #1
```

If you wrote and installed a device driver named ROBOT, then the **OPEN** statement might appear as:

```
OPEN "\DEV\ROBOT" FOR OUTPUT AS #1
```

To open the printer for output, you could use either of the following two lines (the first line uses the BASIC device, LPT1:, while the second line uses the DOS device, LPT1):

```
OPEN "LPT1:" FOR OUTPUT AS #1

OPEN "LPT1" FOR OUTPUT AS #1
```

The following statement opens the file RECORDS in random-access mode, for reading only. The statement locks the file for writing, but allows reading by other processes while the OPEN is in effect:

```
OPEN "RECORDS" FOR RANDOM ACCESS READ LOCK WRITE AS #1
.
.
.
```

The following example opens the file named INVEN for input as file number 2:

```
OPEN "I",2,"INVEN"
```

OPEN COM Statement

■ Syntax

OPEN "COM*n***:** *optlist1 optlist2*" [[**FOR***mode*]] **AS** [[**#**]]*filenum* [[**LEN=***reclen*]]

■ Action

Opens and initializes a communications channel for I/O

■ Remarks

COM*n***:** is the name of the device to be opened. The *n* argument is the number of a legal communications device, such as **COM1:** or **COM2:**. The first list of options, *optionlist1* (*optlist1*), has the following form:

[[*speed*]][,[[*parity*]] [,[[*data*]][,[[*stop*]]]]]

Option	Description
speed	The baud rate of the device to be opened.
parity	The parity of the device to be opened. Valid entries for parity are: **N** (none), **E** (even), **O** (odd), **S** (space), or **M** (mark).
data	The number of data bits per byte. Valid entries are 5, 6, 7, or 8.
stop	The number of stop bits. Valid entries are 1, 1.5, or 2.

Options from this list must be entered in the order shown; moreover, if options from *optionlist2* (*optlist2*) are chosen, comma placeholders must still be used even if none of the options from *optionlist1* are chosen. For example:

```
OPEN "COM1: ,,,,CD1500" FOR INPUT AS #1
```

The choices for *optionlist2* are described in the following list. The argument *m* is given in milliseconds; the default value for *m* is 1000.

Option	Action
RS	Suppresses detection of RTS (Request To Send).
CS[*m*]	Controls detection of CTS (Clear To Send).
DS[*m*]	Controls detection of DSR (Data Set Ready).
CD[*m*]	Controls detection of CD (Carrier Detect).
LF	Allows communication files to be printed on a serial line printer. When **LF** is specified, a line-feed character (0AH) is automatically sent after each carriage-return character (0CH). This includes the carriage return sent as a result of the width setting. Note that **INPUT** and **LINE INPUT**, when used to read from a **COM** file that was opened with the **LF** option, stop when they see a carriage return, ignoring the line feed.
BIN	Opens the device in binary mode. This option supersedes the **LF** option. **BIN** is selected by default unless **ASC** is specified. In the **BIN** mode, tabs are not expanded to spaces, a carriage return is not forced at the end-of-line, and CTRL-Z is not treated as end-of-file. When the channel is closed, CTRL-Z will not be sent over the RS-232 line.
ASC	Opens the device in ASCII mode. In ASCII mode, tabs are expanded to spaces, carriage returns are forced at the end-of-line, CTRL-Z is treated as end-of-file, and the XON/XOFF protocol is enabled. When the channel is closed, CTRL-Z is sent over the RS-232 line.

The options from this second list can be entered in any order, but they must be separated from one another by commas. For **CS**[*m*], **DS**[*m*], and **CD**[*m*], if a signal is not high within *m* milliseconds, a time out occurs. If *m* is equal to 0 for any of these options, the option is ignored. All signals are checked on reading and writing, provided *m* is nonzero. When a communications channel is first opened, however, **CS** is not checked.

The *mode* argument is one of the following string expressions:

Mode	Action
OUTPUT	Specifies sequential output mode
INPUT	Specifies sequential input mode

If the *mode* expression is omitted, it is assumed to be random-access input/output. Random-access input/output cannot, however, be explicitly chosen as *mode*.

The *filenumber* (*filenum*) is the number with which the file is opened. The **OPEN COM** statement must be executed before a device can be used for RS-232 communication.

Any syntax errors in the **OPEN COM** statement will result in a "Bad file name" error.

A "Device timeout" error occurs if Data Set Ready (DSR) is not detected.

Important

If you are using the separate compilation method, and you have compiled your program with the **/o** option, you must link your object file with **GWCOM.OBJ** to get the routines required for **COM***n***:** support.

■ Example

The following program fragment opens communications channel 1 in random-access mode at a speed of 9600 baud with no parity bit, 8 data bits, and 1 stop bit. Input/Output will be in the binary mode. Other lines in the program can now access channel 1 as file number 2.

```
OPEN "COM1:9600,N,8,1,BIN" AS 2
```

■ Syntax

OPTION BASE *n*

■ Action

Declares the minimum value for array subscripts

■ Remarks

The value of *n* must be either 0 or 1. The default base is 0. If the statement

```
OPTION BASE 1
```

is executed, the lowest value an array subscript can have is 1.

You must include the **OPTION BASE** statement before you define or use any arrays. Repeated **OPTION BASE** statements within a single compilation are not permitted.

Chained programs may have an **OPTION BASE** statement if no arrays are passed between them, or the specified base is identical in the chained programs. The chained-to program inherits the **OPTION BASE** value of the chaining program.

■ See Also

LBOUND, UBOUND

■ Example

The following statement overrides the default value of zero, so the lowest value a subscript in an array can have in this program is 1:

```
OPTION BASE 1
DIM A(20)
PRINT LBOUND(A), UBOUND(A)
```

Output:

```
 1             20
```

OUT Statement

■ Syntax

OUT *port*, *data*

■ Action

Sends a byte to a machine I/O port

■ Remarks

The *port* is the port number. It must be an integer expression in the range 0–65,535.

The *data* argument is the data to be transmitted. It must be an integer expression in the range 0–255.

■ Example

```
OUT 12345,255
```

In 8086 assembly language, this is equivalent to:

```
MOV DX,12345
MOV AL,255
OUT DX,AL
```

■ Syntax

PAINT (*x*,*y*)[[,*paint* [[,*bordercolor*]] [[,*background*]]]]

■ Action

Fills a graphics area with the color or pattern specified

■ Remarks

The coordinates (*x*,*y*) show the point where painting begins. This point should be either inside or outside the figure you want to paint, never on the border itself. If this point is on the inside, the interior of the figure is painted; if the point is on the outside, the background is painted.

If the *paint* argument is a string expression, then **PAINT** executes "tiling," a process that paints a pattern, rather than a solid color. Tiling is similar to "line-styling," which creates dashed lines, rather than solid lines. Like **LINE**, **PAINT** looks at a "tiling" mask each time a point is put down on the screen.

If *paint* is a numeric expression, then the number must be a valid color. It is used to paint the area as before. If you do not specify *paint,* the foreground color is used. (See the reference pages for **COLOR**, **PALETTE**, and **SCREEN** for discussions of valid colors, numbers, and attributes.)

The *bordercolor* identifies the border color of the figure you are painting. When the border color is encountered, painting of the current line stops. If the *bordercolor* is not specified, the *paint* argument is used.

The *background* is a string value that gives the "background tile slice" to skip when checking for termination of the boundary. Painting is terminated when adjacent points display the paint color; however, specifying a background tile slice allows you to paint over an already painted area. When *background* is omitted, the default is CHR$(0).

Painting is complete when a line is painted without changing the color of any pixel; in other words, when the entire line is equal to the paint color.

The **PAINT** command permits coordinates outside the screen or viewport.

Tiling

Tiling is the design of a **PAINT** pattern that is 8 bits wide per plane and up to 64 bytes long. Each byte in the tile string masks 8 bits along the *x* axis when putting down points. The syntax for constructing this tile mask is as follows:

PAINT (*x*,*y*), **CHR$**(*arg1*)+**CHR$**(*arg2*)+...+**CHR$**(*argn*)

where the arguments to **CHR$** are numbers between 0 and 255, represented in binary form across the *x*-axis of the tile. There can be up to 64 of these **CHR$** addends; each generates an image, not of the assigned character, but of the bit arrangement of the code for that character. For example, the decimal number 85 is binary "01010101"; the graphic image line on a black and white screen generated by CHR$(85) is an eight-pixel line, with even-numbered points white, and odd-numbered points black. That is, each bit equal to 1 turns the associated pixel "on" and each bit equal to 0 turns the associated bit "off" in a black and white system. The ASCII character CHR$(85), which is U, is not displayed in this case.

When supplied, *background* specifies the "background tile slice" to skip when checking for boundary termination.

You cannot specify more than two consecutive bytes that match the tile string in the tile background slice. Specifying more than two consecutive bytes will result in an "Illegal function call" error.

■ Example 1

The following program draws a magenta fish with a cyan tail:

```
PI=3.141593
CLS
SCREEN 1

CIRCLE (190,100),100,1,,,.3    'Outline fish body in cyan.
CIRCLE (265,92),5,1,,,.7       'Outline fish eye in cyan.
PAINT (190,100),2,1            'Fill in fish body with magenta.

LINE (40,120)-STEP (0,-40),2   'Outline
LINE -STEP (60,+20),2          '   tail in
LINE -STEP (-60,+20),2         '      magenta.
PAINT (50,100),1,2             'Paint tail cyan.

CIRCLE (250,100),30,0,PI*3/4,PI* 5/4,1.5  'Draw gills in black.
FOR Y = 90 TO 110 STEP 4
   LINE (40,Y)-(52,Y),0        'Draw comb in tail.
NEXT
```

■ Example 2

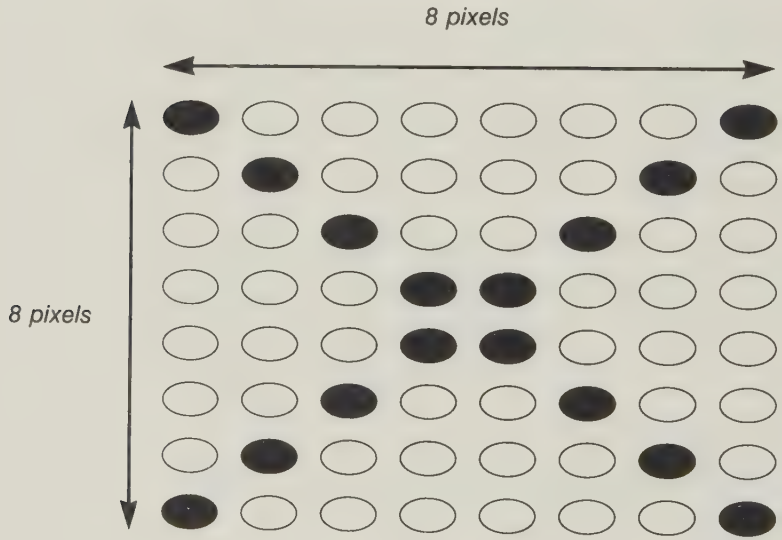
The following example allows the user to enter a pattern in zeros and ones (with a 1 indicating a pixel that is on, and a 0 a pixel that is off); the program then paints the right side of the screen with the pattern.

```
SCREEN 2
I = 0 : TILE$ = ""
NEWVAL:
    PRINT "Tile byte" I "= ";
    INPUT;"",BYTE$                      'Enter binary string
    'If no input (RETURN key pressed), then done:
    IF BYTE$ = "" THEN GOTO FINISHED
    DECIMAL = 0
    FOR K = 1 TO 8                  'Convert string to decimal number
        DIGIT$ = MID$(BYTE$,K,1) 'Get individual digits from string
        DECIMAL = 2*DECIMAL + VAL(DIGIT$)
    NEXT
    PRINT TAB(30) "CHR$(" DECIMAL ")" 'Print decimal value
    TILE$ = TILE$ + CHR$(DECIMAL)      'Add new tile byte to string
    I = I + 1
    GOTO NEWVAL
FINISHED:
    LINE (350,0)-(350,199)   'Draw vertical line
    PAINT (500,100), TILE$   'Paint right half of screen with pattern
```

Sample output:

```
Tile byte 0 = 10000001      CHR$( 129 )
Tile byte 1 = 01000010      CHR$( 66 )
Tile byte 2 = 00100100      CHR$( 36 )
Tile byte 3 = 00011000      CHR$( 24 )
Tile byte 4 = 00011000      CHR$( 24 )
Tile byte 5 = 00100100      CHR$( 36 )
Tile byte 6 = 01000010      CHR$( 66 )
Tile byte 7 = 10000001      CHR$( 129 )
Tile byte 8 =
```

After the above pattern is entered, each eight-by-eight group of pixels on the right side of the screen will have the following appearance:



■ Syntax

PALETTE [[*attribute*,*color*]]
PALETTE USING *integer-array-name* (*arrayindex*)

■ Action

Changes one or more of the colors in the palette

■ Remarks

The **PALETTE** statement works *only* for systems equipped with the IBM Enhanced Graphics Adapter (EGA). A BASIC palette contains a set of colors, with each color specified by an *attribute*. Each *attribute* is paired with an actual display *color*. This *color* determines the actual visual color on the screen, and is dependent on the setting of your screen mode and your actual physical hardware display.

PALETTE with no arguments sets the palette to a known initial setting. This setting is the same as the setting when colors are first initialized.

If arguments are specified, *color* will be displayed whenever *attribute* is specified in any statement that specifies a color. Any color changes on the screen occur immediately. Note that when graphics statements use color arguments, they are actually referring to attributes and not actual colors. **PALETTE** pairs attributes with actual colors.

For example, assume that the current palette consists of *colors* 0, 1, 2, and 3. The following **DRAW** statement

```
DRAW "C3L100"
```

selects attribute 3, and draws a line of 100 pixels using the color associated with the attribute 3, in this case, also 3. If the statement

```
PALETTE 3,2
```

is executed, then the color associated with attribute 3 is changed to color 2. All text or graphics currently on the screen displayed using attribute 3 are instantaneously changed to color 2. All text or graphics subsequently displayed with attribute 3 will also be displayed in color 2. The new palette of *colors* will contain 0, 1, 2, and 2.

With the **USING** option, all entries in the palette can be modified in one **PALETTE** statement. The *integer-array-name* argument is the name of an integer array and the *arrayindex* specifies the index of the first array element in the *integer-array-name* to use in setting your palette. Each *attribute* in the palette is assigned a corresponding *color* from this integer array. The array must be dimensioned large enough to set all the palette entries after *arrayindex*. For example, if you are assigning colors to all 16 attributes, and the index of the first array element given in your **PALETTE USING** statement is 5, then the array must be dimensioned to hold at least 20 elements (since the number of elements from 5–20, inclusive, is 16):

```
DIM PAL%(20)
.
.
.
PALETTE USING PAL%(5)
```

If the *color* argument in an array entry is -1, then the mapping for the associated *attribute* is not changed. All other negative numbers are illegal values for *color*.

You can use the color argument in the **COLOR** statement to set the default text color. (Remember that color arguments in other BASIC statements are actually what are called *attributes* in this discussion.) This color argument specifies the way that text characters appear on the display screen. Under a common initial palette setting, points colored with the *attribute* 0 appear as black on the display screen. Using the **PALETTE** statement, you could, for example, change the mapping of *attribute* 0 from black to white.

Remember that a **PALETTE** statement executed without any parameters assigns all *attributes* their default *colors*.

Table 11.3 lists *attribute* and *color* ranges for various monitor types and screen modes.

Table 11.3

SCREEN Color and Attribute Ranges

SCREEN Mode	Monitor Attached	Adapter	Attribute Range	Color Range
0	Monochrome	MDPA	NA	NA
	Monochrome	EGA	0–15	0–2
	Color	CGA	NA	0–31[a]
	Color/Enhanced[d]	EGA	0–31[a]	0–15
1	Color	CGA	NA	0–3
	Color/Enhanced[d]	EGA	0–3	0–15
2	Color	CGA	NA	0–1
	Color/Enhanced[d]	EGA	0–1	0–15
7	Color/Enhanced[d]	EGA	0–15	0–15
8	Color/Enhanced[d]	EGA	0–15	0–15
9	Enhanced[d]	EGA[b]	0–3	0–15
	Enhanced[d]	EGA[c]	0–15	0–63
10	Monochrome	EGA	0–3	0–8

[a] Attributes 16–31 refer to blinking versions of colors 0–15
[b] With 64K of EGA memory
[c] With greater than 64K of EGA memory
[d] IBM Enhanced Color Display
NA = Not Applicable.
CGA = IBM Color Graphics Adapter
EGA = IBM Enhanced Graphics Adapter
MDPA = IBM Monochrome Display and Printer Adapter

See the **SCREEN** statement reference page for the list of colors available for various **SCREEN** mode, monitor, and graphics adapter combinations.

■ Examples

```
PALETTE 0,2     'Changes all points colored with attribute 0
                'to color 2

PALETTE 0,-1    'Does not modify the palette

PALETTE USING A%(0)     'Changes each palette entry. Since the
                        'array is initialized to zero when it
                        'is first declared, all attributes are
                        'now mapped to display color zero. The
                        'screen will now appear as one single
                        'color. However, it will still be
                        'possible to execute BASIC statements.

PALETTE          'Sets each palette entry to its appropriate
                 'initial display color. Actual initial colors
                 'depend on your screen hardware configuration.
```

■ Syntax

PCOPY *sourcepage, destinationpage*

■ Action

Copies one screen page to another in all screen modes

■ Remarks

The *sourcepage* is an integer expression in the range 0 to *n*, where *n* is determined by the current video-memory size and the size per page for the current screen mode.

The *destinationpage* has the same requirements as the *sourcepage*.

■ See Also

CLEAR, SCREEN

■ Example

This copies the contents of page 1 to page 2:

```
PCOPY 1,2
```

PEEK Function

■ Syntax

PEEK(*n*)

■ Action

Returns the byte read from the indicated memory location *n*

■ Remarks

The returned value is an integer in the range 0–255. The argument *n* is a single-precision value in the range 0–1,048,575. The argument *n* is the offset from the current segment, which was defined by the last **DEF SEG** statement. For the interpretation of a negative value of *n*, see the **VARPTR** function reference page.

The **PEEK** function complements the **POKE** statement.

■ See Also

DEF SEG, **POKE**, **VARPTR**

■ Example

```
A = PEEK(&H5A00)     'Load the value at the location with the
                     'hexadecimal address 5A00 into the variable A
```

■ Syntax

PEN(*n*)

■ Action

Reads the lightpen coordinates

■ Remarks

The argument *n* is a numeric expression returning an unsigned integer in the range 0–9 (see below).

The **PEN**(*n*) function reads the lightpen coordinates, where *n* is:

Argument	Value Returned by PEN
0	–1 if pen was down since last poll, 0 if not
1	The x pixel coordinate where pen was last pressed
2	The y pixel coordinate where pen was last pressed
3	The current pen switch value: –1 if down, 0 if up
4	The last known valid x pixel coordinate
5	The last known valid y pixel coordinate
6	The character row position where pen was last pressed
7	The character column position where pen was last pressed
8	The last known character row where the pen was positioned
9	The last known valid character column where the pen was positioned

■ Example

The following example produces an endless loop to print the current pen-switch status (UP/DOWN):

```
CLS
PEN ON
TRUE = -1
WHILE TRUE
   P = PEN(3)
   LOCATE 1,1 : PRINT "PEN IS ";
   IF P THEN PRINT "DOWN" ELSE PRINT "UP"
   X = PEN(4) : Y = PEN(5)
   PRINT "X =" X, "Y =" Y
WEND
```

■ Syntax

PEN ON
PEN OFF
PEN STOP

■ Action

Enables, disables, or suspends lightpen-event trapping

■ Remarks

The **PEN ON** statement enables lightpen-event trapping.

The **PEN OFF** statement disables lightpen-event trapping.

The **PEN STOP** statement suspends lightpen-event trapping: a **PEN** event is remembered and trapped as soon as event trapping is enabled.

The pen is initially off. A **PEN ON** statement must be executed before any read pen function calls. If a read pen function is called when the pen is off, an "Illegal function call" error will result.

PEN ON also enables event trapping with an **ON PEN** statement.

To speed program execution, the pen should be turned off with a **PEN OFF** statement.

■ See Also

PEN (Function)

PLAY Function

■ Syntax

PLAY (*n*)

■ Action

Returns the number of notes currently in the background music queue

■ Remarks

The argument *n* is a dummy argument and may be any value.

PLAY(*n*) will return 0 when the user is in music-foreground mode.

■ See Also

ON PLAY, PLAY ON, PLAY OFF, PLAY STOP

■ Syntax

PLAY *subcommand-string*

■ Action

Plays music as specified by *subcommand-string*

■ Remarks

The *subcommand-string* is a string expression containing one or more of the subcommands listed below.

PLAY uses a concept similar to **DRAW**, in that it embeds a music macro language (described below) in one statement. A set of subcommands, used as part of the **PLAY** statement, specifies a particular action.

In compiled programs, you should use the **VARPTR$**(*variable*) form for variables. For example, the following interpreter statements:

```
PLAY "XA$"
PLAY "O=I"
```

should be written for the compiler like this:

```
PLAY "X" + VARPTR$(A$)
PLAY "O=" + VARPTR$(I)
```

The *subcommand-string* music macros are described as follows:

Change Octave	Macro Action
>	Increases octave by 1. Octave will not advance beyond 6.
<	Decreases octave by 1. Octave will not drop below 0.
Tone	**Macro Action**
O *n*	Sets the current octave. There are seven octaves, numbered 0–6.

A–G	Plays a note in the range **A–G**. A "#" or a "+" after the note specifies sharp; a "–" specifies flat.
N *n*	Plays note *n*. The range for *n* is 0–84 (in the seven possible octaves, there are 84 notes); *n* = 0 means a rest.
Duration	**Macro Action**
L *n*	Sets the length of each note. **L** 4 is a quarter note, **L** 1 is a whole note, etc. The range for *n* is 1–64. The length may also follow the note when a change of length only is desired for a particular note. In this case, **A** 16 is equivalent to **L** 16 **A**.
MN	Sets "music normal" so that each note will play 7/8 of the time determined by the length (**L**).
ML	Sets "music legato" so that each note will play the full period set by length (**L**).
MS	Sets "music staccato" so that each note will play 3/4 of the time determined by the length (L).
Tempo	**Macro Action**
P *n*	Specifies a pause, ranging from 1–64. This option corresponds to the length of each note, set with **L** *n*.
T *n*	Sets the "tempo," or the number of L 4 quarter notes in one minute. The range for *n* is 32–255. The default for *n* is 120.
Operation	**Macro Action**
MF	Sets music (**PLAY** statement) and **SOUND** to run in the foreground. That is, each subsequent note or sound will not start until the previous note or sound has finished. This is the default setting.
MB	Music (**PLAY** statement) and **SOUND** are set to run in the background. That is, each note or sound is placed in a buffer, allowing the BASIC program to continue executing while the note or

sound plays in the background. The maximum number of notes that can be played in the background at one time is 32.

Suffixes	Macro Action
# or +	Follows a specified note, and turns it into a sharp.
–	Follows a specified note, and turns it into a flat.
.	A period after a note causes the note to play 3/2 times the length determined by **L** (length) times **T** (tempo). Multiple periods can appear after a note. Each period adds a length equal to one half the length of the previous period. For example, A. equals 1 + 1/2, or 3/2; A.. is 1 + 1/2 + 1/4, or 7/4; A... is 1 + 1/2 + 1/4 + 1/8, or 15/8; and so on. Periods can appear after a pause (**P**). In this case, the pause length can be scaled in the same way notes are scaled.

Substring	Macro Action
"X" + VARPTR$(*string***)**	Executes a substring. Because of the slow-clock interrupt rate, some notes will not play at higher tempos (**L** 64 at **T** 255, for example).

■ Example 1

The following example uses ">" to play the scales from octave 0 to octave 6, then reverses with "<" to play the scales from octave 6 to octave 0:

```
SCALES$ = "CDEFGAB"
PLAY "O0 X" + VARPTR$(SCALE$)
FOR I = 1 TO 6
   PLAY ">X" + VARPTR$(SCALE$)
NEXT
PLAY "O6 X" + VARPTR$(SCALE$)
FOR I = 1 TO 6
   PLAY "<X" + VARPTR$(SCALE$)
NEXT
```

■ Example 2

This example will play the beginning of the first movement of Beethoven's Fifth Symphony:

```
LISTEN$ = "T180 O2 P2 P8 18 GGG 12 E-"
FATE$ = "P24 P8 18 FFF 12 D"
PLAY LISTEN$ + FATE$
```

■ Syntax

PLAY ON
PLAY OFF
PLAY STOP

■ Action

PLAY ON enables play event trapping, **PLAY OFF** disables play event trapping, and **PLAY STOP** suspends play event trapping.

■ Remarks

If a **PLAY OFF** statement has been executed, the **GOSUB** is not performed and is not remembered.

If a **PLAY STOP** statement has been executed, the **GOSUB** is not performed, but will be performed as soon as a **PLAY ON** statement is executed.

When an event trap occurs (that is, the **GOSUB** is performed), an automatic **PLAY STOP** is executed so that recursive traps cannot take place. The **RETURN** from the trapping subroutine will automatically perform a **PLAY ON** statement unless an explicit **PLAY OFF** was performed inside the subroutine.

The **RETURN** syntax

RETURN { *linenumber* | *linelabel*}

can be used to return to a specific line from the trapping subroutine. Use this type of return with care, however, because any other **GOSUB**, **WHILE**, or **FOR** statements that were active at the time of the trap will remain active, and errors such as "FOR without NEXT" may result. In addition, the **RETURN** must be to a line in the main program or subroutine that caused the error.

PMAP Function

■ Syntax

PMAP (*expression, function*)

■ Action

Maps world-coordinate expressions to physical locations, or maps physical expressions to a world-coordinate location

■ Remarks

The argument *expression* indicates the coordinate of the point to be mapped.

The argument *function* can have one of the four following values:

Value	Action
0	Maps world expression to physical x coordinate
1	Maps world expression to physical y coordinate
2	Maps physical expression to world x coordinate
3	Maps physical expression to world y coordinate

The four **PMAP** functions allow the user to find equivalent point locations between the world coordinates created with the **WINDOW** statement and the physical coordinate system of the screen or viewport as defined by the **VIEW** statement.

■ See Also

VIEW, WINDOW

■ Example

```
SCREEN 2
'Coordinates of upper-left corner of window defined in following
'statement are (80,100); coordinates of lower-right corner are
'200,200:
WINDOW SCREEN (80,100) - (200,200)

'If physical screen coordinates are (0,0) in the upper-left corner
'and (639,199) in the lower-right corner, then the following
'statements will return the screen coordinates equivalent to the
'window coordinates 80,100:
X = PMAP(80,0)          'X = 0
Y = PMAP(100,1)         'Y = 0

'The following statements will return the screen coordinates
'equivalent to the window coordinates 200,200:
X = PMAP(200,0)         'X = 639
Y = PMAP(200,1)         'Y = 199

'The following statements will return the window coordinates
'equivalent to the screen coordinates 639,199:
X = PMAP(639,2)         'X = 200
Y = PMAP(199,3)         'Y = 200
```

POINT Function

■ Syntax

POINT (*x*,*y*)
POINT (*number*)

■ Action

Reads the color number of a pixel from the screen

■ Remarks

The coordinates *x* and *y* identify the pixel evaluated by **POINT**. If the specified pixel is out of range, **POINT** returns the value –1.

POINT with one argument allows the user to retrieve the current graphics-cursor coordinates. Therefore, the statement

POINT (*number*)

returns the value of the current *x* or *y* graphics accumulator as follows:

Argument	Value Returned by POINT
0	The current physical *x* coordinate.
1	The current physical *y* coordinate.
2	The current logical *x* coordinate. If the **WINDOW** statement has not been used, this will return the same value as the **POINT(0)** function.
3	The current logical *y* coordinate. If the **WINDOW** statement has not been used, this will return the same value as the **POINT(1)** function.

The physical coordinate is the coordinate on the screen or current viewport.

■ Example

The following example redraws the ellipse drawn with the **CIRCLE** statement, tilted the specified number of degrees:

```
DEFINT X,Y
INPUT "Angle of tilt in degrees (0 to 90): ",ANG
SCREEN 1                          'Medium resolution screen
ANG = (3.1415926#/180)*ANG        'Convert degrees to radians
CS = COS(ANG) : SN = SIN(ANG)
CIRCLE (45,70),50,2,,,2           'Draw ellipse
PAINT (45,70),2                   'Paint interior of ellipse
FOR Y = 20 TO 120
    FOR X = 20 TO 70
        'Check each point in rectangle enclosing ellipse:
        IF POINT(X,Y) <> 0 THEN
            'If the point is in the ellipse, plot a corresponding
            'point in the "tilted" ellipse:
            XNEW = (X*CS - Y*SN) + 200 : YNEW = (X*SN + Y*CS)
            PSET(XNEW,YNEW),2
        END IF
    NEXT
NEXT
END
```

POKE Statement

■ Syntax

POKE *address,byte*

■ Action

Writes a byte into a memory location

■ Remarks

The expression *address* is a single-precision value that represents the address of the memory location; *address* must be in the range 0–1,048,575.

The expression *byte* is the data byte to be written; it is an integer value in the range 0–255.

The *address* is the offset from the current segment, which was set by the last **DEF SEG** statement. For interpretation of negative values of *address*, see the reference pages for the **VARPTR** function.

The complementary function to **POKE** is **PEEK**.

Warning

Use **POKE** carefully. If used incorrectly, it can cause BASIC or the operating system to fail.

■ See Also

DEF SEG, PEEK, VARPTR

■ Example

```
POKE &H5A00,&HFF      'Put hexadecimal value FF in memory location
                      'with hexadecimal address 5A00
```

■ Syntax

POS(*column*)

■ Action

Returns the current horizontal *column* position of the cursor

■ Remarks

The left-most position is 1; *column* is a dummy argument. To return the current vertical-line position of the cursor, use the **CSRLIN** function.

■ See Also

CSRLIN, LPOS

■ Example

```
PRINT "This program starts a new line after every forty"
PRINT "characters are printed. Press <CTRL-C> to end."
PRINT
START:
    WHILE POS(0) < 41        'Stay on same line until 40 characters
        CHAR$ = ""           'printed
        WHILE CHAR$ = ""     'Wait for input from the keyboard
            CHAR$ = INKEY$
        WEND
        'If input is key·combination CTRL-C then end; otherwise,
        'print the character:
        IF ASC(CHAR$) = 3 THEN END ELSE PRINT CHAR$;
    WEND
    PRINT                    'Print a new line
    GOTO START
```

PRESET Statement

■ Syntax

PRESET [[**STEP**]](*xcoordinate*,*ycoordinate*) [[,*color*]]

■ Action

Draws a specified point on the screen

■ Remarks

PRESET works exactly like **PSET**, except that if the *color* is not specified, the background color is selected.

The *xcoordinate* and *ycoordinate* specify the pixel that is to be set.

The *color* is the color number that is to be used for the specified point.

The **STEP** option, when used, indicates the given *x* and *y* coordinates will be relative, not absolute. This means *xcoordinate* and *ycoordinate* are distances from the most recent cursor location, not distances from the (0,0) screen coordinate. If a coordinate is outside the current viewport, no action is taken, nor is an error message given.

Coordinates can be shown as absolutes, as in the above syntax, or the **STEP** option can be used to reference a point relative to the most recent point used. For example, if the most recent point referenced were (10,10)

```
PRESET STEP (10,5)
```

would reference the point at (20,15).

■ Example

The following example draws a line twenty pixels long that moves across the screen from left to right:

```
SCREEN 1 : COLOR 1,1 : CLS
FOR I = 0 TO 299 STEP 3
    FOR J = I TO 20+I
        PSET (J,50),2          'Draw the line in new location
    NEXT
    FOR J = I TO 20+I
        PRESET (J,50)          'Erase the line
    NEXT
NEXT
```

PRINT Statement

■ Syntax

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

■ Action

Outputs data on the screen

■ Remarks

If *expressionlist* is omitted, a blank line is printed. If *expressionlist* is included, the values of the expressions are printed on the screen. The expressions in the list may be numeric or string expressions. (String literals must be enclosed in quotation marks.)

Printed numbers are always followed by a space. Positive numbers are preceded by a space. Negative numbers are preceded by a minus sign. Single-precision numbers, which can be represented with 6 or fewer digits in the unscaled format no less accurately than they can be represented in the scaled format, are output using the unscaled format. For example, 1E–7 is output as .0000001, and 1E–8 is output as 1E–08. Double-precision numbers, which can be represented with 16 or fewer digits in the unscaled format no less accurately than they can be represented in the scaled format, are output using the unscaled format. For example, 1D–15 is output as .0000000000000001, and 1D–16 is output as 1D–16.

Print Positions

The position of each printed item is determined by the punctuation used to separate the items in the list. BASIC divides the line into print zones of 14 spaces each. In the list of expressions, a comma causes the next value to be printed at the beginning of the next zone. A semicolon causes the next value to be printed immediately after the last value. Typing one or more spaces between expressions has the same effect as typing a semicolon.

If a comma or a semicolon terminates the list of expressions, the next **PRINT** statement begins printing on the same line, after spacing accordingly. If the list of expressions terminates without a comma or a semicolon, a carriage-return–line-feed sequence is printed at the end of the line. If the printed line is wider than the screen width, BASIC goes to the next physical line and continues printing.

■ Compiler/Interpreter Differences

The compiler does not support the shorthand use of a question mark (?) in place of the word **PRINT**.

■ Example 1

In this example, the commas in the PRINT statement cause each value to be printed at the beginning of the next print zone:

```
X=5
PRINT X+5, X-5, X*(-5), X^5
END
```

Output:

```
 10            0            -25            3125
```

■ Example 2

In this example, the semicolon at the end of the first PRINT statement causes the first two PRINT statements to be printed on the same line. The last PRINT statement causes a blank line to be printed before the next prompt.

```
LOOP = -1
WHILE LOOP
    INPUT "Input X (input 0 to end): "; X
    IF X = 0 THEN
        LOOP = 0
    ELSE
        PRINT X "SQUARED IS" X^2 "AND";
        PRINT X "CUBED IS" X^3
        PRINT
    END IF
WEND
```

Sample output:

```
Input X (input 0 to end): 9
 9 SQUARED IS 81 AND 9 CUBED IS 729

Input X (input 0 to end): 21
 21 SQUARED IS 441 AND 21 CUBED IS 9261

Input X (input 0 to end): 0
```

■ Example 3

In this example, the semicolons in the **PRINT** statement cause each value to be printed immediately after the preceding value. (Remember, a number is always followed by a space, and positive numbers are preceded by a space.)

```
FOR X=1 TO 5
  J=J+5
  K=K+10
  PRINT J;K;
NEXT X
```

Output:

```
 5  10  10  20  15  30  20  40  25  50
```

■ Syntax

PRINT# *filenumber*,[**USING** *string-expression*;] *expressionlist*

■ Action

Writes data to a sequential file

■ Remarks

The *filenumber* is the number used when the file was opened for output. The *string-expression* consists of formatting characters as described in the PRINT USING statement reference pages. The expressions in *expressionlist* are the numeric and/or string expressions to be written to the file.

With **PRINT#**, an image of the data is written to the file, just as it would be displayed on the terminal screen with a **PRINT** statement. For this reason, care should be taken to delimit the data so that it will be output correctly.

In the list of expressions, numeric expressions should be delimited by semicolons. For example:

```
PRINT#1,A;B;C;X;Y;Z
```

Note

If commas are used as delimiters, the extra blanks that are inserted between print fields will also be written to the file.

String expressions must be separated by semicolons in the list. To format the string expressions correctly in the file, use the explicit delimiter CHR$(34) in the list of expressions.

■ See Also

PRINT, WRITE#

■ Example

```
A$ = "CAMERA, AUTOFOCUS" : B$= "September 20, 1985" : C$ = "42"
Q$ = CHR$(34)
OPEN "INVENT.DAT" FOR OUTPUT AS #1  'Open INVENT.DAT for writing
'Write A$, B$, C$ without delimiters:
PRINT #1, A$ B$ C$
'Write A$, B$, C$ with delimiters:
PRINT #1, Q$ A$ Q$ Q$ B$ Q$ Q$ C$ Q$
CLOSE #1
OPEN "STOREINVENT" FOR INPUT AS #1   'Open STOREINVENT for reading
FOR I% = 1 TO 2                      'Read first two records and
    INPUT #1, FIRST$,SECOND$,THIRD$  'print
    PRINT FIRST$ TAB(30) SECOND$ TAB(60) THIRD$ : PRINT
NEXT
CLOSE #1
```

Output:

```
CAMERA                    AUTOFOCUSSeptember 20          198542

CAMERA, AUTOFOCUS         September 20, 1985             42
```

■ Statement Syntax

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

■ Action

Prints strings or numbers using a specified format

■ Remarks

The *expressionlist* contains the string expressions or numeric expressions that are to be printed, separated by semicolons.

The *formatstring* is a string literal (or variable) composed of special formatting characters. These formatting characters determine the field and the format of the printed strings or numbers.

Formatting Strings

When **PRINT USING** is used to print strings, one of three formatting characters can be used to format the string field, as described in the following list:

Character	Action
!	Specifies that only the first character in the given string is to be printed
\ \	Specifies that 2 + *n* characters from the string are to be printed, where *n* is the number of spaces between the two backslashes: if the backslashes are typed with no spaces, two characters will be printed; with one space, three characters will be printed, and so on. If the string is longer than the field, the extra characters are ignored. If the field is longer than the string, the string is left-justified in the field and padded with spaces on the right.
&	Specifies a variable-length string field. When the field is specified with the ampersand (&), the string is output without modification.

■ Example 1

```
'Using the three string-formatting characters to modify the
'appearance of printed output:
A$ = "LOOK" : B$ = "OUT"
PRINT USING "!";A$;B$
PRINT USING "\  \";A$;B$          '2 spaces between backslashes ->
                                  'print 4 characters from A$
PRINT USING "\   \";A$;B$;"!!"    '3 spaces -> print A$ and
PRINT USING "!";A$;               'a blank
PRINT USING "&";B$
```

Output:

```
LO
LOOKOUT
LOOK OUT   !!
LOUT
```

Formatting Numbers

When **PRINT USING** is used to print numbers, the following special characters may be used to format the numeric field:

Character	Action
#	Represents each digit position. Digit positions are always filled. If the number to be printed has fewer digits than positions specified, the number will be right-justified (preceded by spaces) in the field.
.	Prints a decimal point. A decimal point may be inserted at any position in the field. If the format string specifies that a digit is to precede the decimal point, the digit will always be printed (as 0, if necessary). Numbers are rounded as necessary.
+	Causes the sign of the number (plus or minus) to be printed before (if it appears at the beginning of the format string) or after (if it appears at the end of the format string) the number.
–	Causes a negative number to be printed with a trailing minus sign if it appears at the end of the format string.

**	Causes leading spaces in the numeric field to be filled with asterisks. The double asterisk also specifies positions for two more digits.
$$	Causes a dollar sign to be printed to the immediate left of the formatted number. The $$ specifies two more digit positions, one of which is the dollar sign.
**$	Combines the effects of the double-asterisk and double-dollar-sign symbols. Leading spaces are asterisk-filled and a dollar sign is printed before the number. The ** symbols specify three more digit positions, one of which is the dollar sign. When negative numbers are printed, the minus sign appears immediately to the left of the dollar sign.
,	If it appears to the left of the decimal point in a formatting string, it causes a comma to be printed to the left of every third digit left of the decimal point. If it appears at the end of the format string, it is printed as part of the string. A comma specifies another digit position. The comma has no effect if used with exponential (^^^^) format.
^^^^	Specifies exponential format. The four carets allow space for E+xx to be printed. Any decimal point position may be specified. The significant digits are left-justified, and the exponent is adjusted. Unless a leading +, trailing +, or – is specified, one digit position will be used to the left of the decimal point to print a space or a minus sign.
_	An underscore in the format string causes the next character to be output as a literal character. A literal underscore will be printed as the result of two underscores (__) in the format string.
%	If the number to be printed is larger than the specified numeric field, a percent sign is printed in front of the number. If rounding causes the number to exceed the field, a percent sign will be printed in front of the rounded number. If the number of digits specified exceeds 24, an "Illegal function call" error results.

■ Example 2

```
'Format and print numeric data:
PRINT USING "##.##";.78
PRINT USING "###.##";987.654
PRINT USING "##.##   ";10.2,5.3,66.789,.234
PRINT USING "+##.##   ";-68.95,2.4,55.6,-.9
PRINT USING "##.##-   ";-68.95,22.449,-7.01
PRINT USING "**#.#   ";12.39,-0.9,765.1
PRINT USING "$$###.##";456.78
PRINT USING "**$##.##";2.34
PRINT USING "####,.##";1234.5
PRINT USING "##.##^^^^";234.56
PRINT USING ".####^^^^-";-888888
PRINT USING "+.##^^^^";123
PRINT USING "_!##.##_!";12.34
PRINT USING "##.##";111.22
PRINT USING ".##";.999
```

Output:

```
 0.78
987.65
10.20    5.30   66.79    0.23
-68.95    +2.40   +55.60    -0.90
68.95-   22.45     7.01-
*12.4   *-0.9   765.1
 $456.78
***$2.34
1,234.50
 2.35E+02
 .8889E+06-
 +.12E+03
!12.34!
%111.22
%1.00
```

■ Syntax

PSET [**STEP**](*xcoordinate*,*ycoordinate*) [,*color*]

■ Action

Draws a point on the screen

■ Remarks

The *xcoordinate* and *ycoordinate* specify the point on the screen to be colored.

The *color* is the number of the color to be used.

The **STEP** option, when used, indicates the given x and y coordinates will be relative, not absolute. This means the x and y are distances from the most recent cursor location, not distances from the (0,0) screen coordinate. When BASIC scans coordinate values, it will allow them to be beyond the edge of the screen (the size of the screen is dependent on display hardware being used, and can be adjusted with the **WIDTH** statement). However, values outside the integer range –32,768 to 32,767 will cause an "Overflow" error.

Coordinates can be shown as offsets by using the **STEP** option to reference a point relative to the most recent point used. The syntax of the **STEP** option is:

STEP (*xoffset*,*yoffset*)

For example, if the most recent point referenced were (0,0),

```
PSET STEP (10,0)
```

would reference a point at offset 10 from x and offset 0 from y.

The coordinate (0,0) is always the upper-left corner of the screen.

PSET allows the *color* to be left off the command line. If it is omitted, the default is the foreground color.

■ See Also

PRESET

■ Example

This example draws a line from (0,0) to (100,100) and then erases that line by writing over it with the background color:

```
SCREEN 2  'Draw a line from (0,0) to (100,100)
FOR I=0 TO 100
   PSET (I,I)
NEXT I
FOR I=0 TO 100 'Now erase that line
   PSET STEP (-1,-1),0
NEXT I
```

■ Syntax

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

■ Action

Writes a record from a random-access buffer to a random-access file

■ Remarks

The *filenumber* is the number under which the file was opened. If *recordnumber* is omitted, the record will assume the next available record number (after the last **PUT** or **GET**). The largest possible record number is 2^{24}–1, or 16,777,215. The smallest record number is 1.

The **GET** and **PUT** statements allow fixed-length input and output for BASIC communications files. However, because of the possible noise associated with telephone-line communications, it is recommended that you not use **GET** and **PUT** for telephone communication.

Note

LSET, **RSET**, **PRINT#**, **PRINT# USING**, and **WRITE#** may be used to put characters in the random file buffer before executing a **PUT** statement.

In the case of **WRITE#**, BASIC pads the buffer with spaces up to the carriage return. Any attempt to read or write past the end of the buffer causes a "FIELD overflow" error.

■ See Also

CVD, **CVI**, **CVS**, **GET** (File I/O), **LSET**, **MKD$**, **MKI$**, **MKS$**

■ Example

See examples in **CVI**, **CVS**, **CVD** functions and **REDIM** statement reference pages.

■ Syntax

PUT (*x*, *y*) *arrayname* [,*actionverb*]

■ Action

Places a graphic image obtained by a **GET** statement onto the screen

■ Remarks

The **PUT** statement arguments are described as follows:

Argument	Description
(*x1*,*y1*)	Coordinates specifying the top-left corner of the rectangle enclosing the image to be placed in the current output window.
arrayname	The name (assigned in the **GET** statement) to the array that holds the image. See the **GET** statement-graphics reference page for the formula that computes the size of this array.
actionverb	The *actionverb* performs the interaction between the stored image and the one already on the screen; the different values for *actionverb* are described in the following list:

Verb	Action
PSET	Transfers the data point-by-point onto the screen. Each point has the exact color attribute it had when it was taken from the screen with **GET**.
PRESET	The same as **PSET** except that a negative image (for example, black on white) is produced.
AND	Used when the image is to be transferred over an existing image on the screen. The resulting image is the product of the logical **AND** expression; points that had the

same color in both the existing image and the **PUT** image will remain the same color, points that do not have the same color in both the existing image and the **PUT** image will not.

OR Used to superimpose the image onto an existing image; the **PUT** image does not erase the previous screen contents.

XOR A special mode often used for animation. **XOR** causes the points on the screen to be inverted where a point exists in the array image. This behavior is exactly like that of the cursor: when an image is **PUT** against a complex background twice, the background is restored unchanged. This allows you to move an object around the screen without erasing the background.

The default *actionverb* is **XOR**.

Animation

One of the most useful things that can be done with **GET** and **PUT** is animation. Animation for an object is performed by following the next four steps:

1. **PUT** the object on the screen.
2. Recalculate the new position of the object.
3. **PUT** the object on the screen a second time, at the old location (using **XOR**), to remove the old image.
4. Go to step 1, but this time **PUT** the object at the new location.

Movement performed this way will leave the background unchanged. Flicker can be cut down by minimizing the time between steps 4 and 1, and by making sure that there is enough time delay between 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, animation can be performed using the **PSET** action verb. The idea is to leave a border around the image, when it is first produced, that is as large or larger than the maximum distance the object will move. Thus, when an object is moved, this border will effectively erase any points left by the previous **PUT**. This method can be somewhat faster than the method using **XOR**, described above, since only one **PUT** is required to move an object (although you must **PUT** a larger image). Because you can specify (*x2,y2*), the image can be scaled (enlarged or reduced). For example, if you load a circle from the screen with the statement `GET(0,0)-(50,50),A`, then redraw the circle with the statement `PUT(100,100)-(150,200),A`, this would put the array `A` on the screen, elongated along the y axis, producing an oval.

This technique can be used to produce better parallax perspectives during animation. If the moving object becomes larger, it appears to be moving towards the user.

Note that **PUT** does not allow an image to be clipped; that is, coordinates outside the current screen or window will produce a run-time error.

See Also

GET (Graphics), **PRESET**, **PRINT**, **PSET**

■ Example

The following example creates a moving white ball that ricochets off the sides of the screen until you press a key to end the program:

```
DEFINT A-Z
DIM BALL(84)      'Dimension integer array large enough to hold ball

SCREEN 2          'High-resolution screen (640 pixels by 200 pixels)
INPUT "Press any key to end; press <RETURN> to start",TEST$
CLS
CIRCLE (16,16),14        'Draw and paint ball
PAINT (16,16),1
GET (0,0)-(32,32),BALL
X = 0 : Y = 0
XDELTA = 2 : YDELTA = 1

START:
    XBOUND = -1 : YBOUND = -1

    'Continue moving in same direction as long as ball is within
    'the boundaries of the screen - (0,0) to (640,200):
    WHILE XBOUND AND YBOUND
        X = X + XDELTA : Y = Y + YDELTA
        TEST$ = INKEY$           'Test for key press
        IF TEST$ <> "" THEN END
        IF (X < 1 OR X > 600) THEN
            XBOUND = 0
            XDELTA = -XDELTA
        END IF
        IF (Y < 1 OR Y > 160) THEN
            YBOUND = 0
            YDELTA = -YDELTA
        END IF
       'Put new image on screen, simultaneously erasing old image:
        PUT (X,Y),BALL,PSET
    WEND
    GOTO START
END
```

RANDOMIZE Statement

■ Syntax

RANDOMIZE[[*expression*]]

■ Action

Reseeds the random number generator

■ Remarks

If *expression* is omitted, BASIC suspends program execution and asks for a value by printing

```
Random Number Seed (-32768 to 32767)?
```

before executing **RANDOMIZE**.

If *expression* is a variable, the value of that variable is used to seed the random numbers.

If *expression* is the word TIMER, then the **TIMER** function is used to pass a random number seed.

If the random number generator is not reseeded, the **RND** function returns the same sequence of random numbers each time the program is run. To change the sequence of random numbers every time the program is run, place a **RANDOMIZE** statement at the beginning of the program and change the argument with each run.

■ Example

The following fragment is from a program that generates random bridge hands (13 cards) from a deck of 52 cards:

```
DEFINT A-Z
DIM X(13), Y(52)
GOSUB FILLARRAY
FOR I = 1 TO 13 : PRINT X(I) : NEXT
.
.
.
END
```

```
FILLARRAY:
    'Fill array Y with numbers 1 to 52:
    FOR I = 1 TO 52 : Y(I) = I : NEXT
    MAX = 52
    'Generate different sequence each time program is run:
    RANDOMIZE TIMER
    FOR M = 1 TO 13
        INDEX = INT(RND*MAX) + 1    'Pick a number from 1 to MAX
        TEMP = Y(INDEX)
        'Overwrite the number in Y(INDEX) with the number in Y(MAX)
        'so the original number in Y(INDEX) won't be chosen again:
        Y(INDEX) = Y(MAX)
        MAX = MAX - 1               'Decrease the value of MAX by 1
        N = M
        'Sort the array X from high to low:
        WHILE N > 1 AND TEMP > X(N-1)
            X(N) = X(N-1)
            N = N - 1
        WEND
        'Store the number originally in Y(INDEX) in X(N):
        X(N) = TEMP
    NEXT
RETURN
```

Sample output:

```
 52
 51
 48
 45
 33
 32
 24
 21
 18
 14
 13
 9
 2
```

READ Statement

■ Syntax

READ *variablelist*

■ Action

Reads values from a **DATA** statement and assigns them to variables

■ Remarks

A **READ** statement must always be used in conjunction with a **DATA** statement. **READ** statements assign variables to **DATA** statement values on a one-to-one basis. **READ** statement variables may be numeric or string, and the values read must agree with the variable types specified. If they do not agree, a syntax error will result.

A single **READ** statement may access one or more **DATA** statements (they will be accessed in order), or several **READ** statements may access the same **DATA** statement. If the number of variables in *variablelist* exceeds the number of elements in the **DATA** statement(s), an "Out of data" error message is printed. If the number of variables specified is fewer than the number of elements in the **DATA** statement(s), subsequent **READ** statements will begin reading data at the first unread element. If there are no subsequent **READ** statements, the extra data is ignored.

To reread **DATA** statements from the start, use the **RESTORE** statement.

■ See Also

DATA, **RESTORE**

■ Example

This program segment reads the values from the DATA statements into the array A. After execution, the value of A(1) will be 3.08, the value of A(2) will be 5.19, and so on.

```
FOR I = 1 TO 10
   READ A(I)
NEXT I
DATA 3.08,5.19,3.12,3.98,4.24
DATA 5.08,5.55,4.00,3.16,3.37
```

■ Syntax

REDIM [[**SHARED**]] *arrayname*(*subscripts*) [[*arrayname* (*subscripts*)...]]

■ Action

Changes the space allocated to an array that has been declared dynamic

■ Remarks

The **SHARED** option allows you to share variables among all the subprograms in a compilation; it must appear in a **REDIM** statement at the main program level.

REDIM takes the following arguments:

Argument	Description
arrayname	The name of the array you want to redimension
subscripts	The new dimensions of the array

The **REDIM** statement changes the space allocated to an array that has been declared dynamic, either as the result of a $**DYNAMIC** metacommand or as the result of a variable in a **DIM** statement. Static arrays cannot appear in a **REDIM** statement.

When a **REDIM** statement is compiled, all arrays declared in the statement are treated as dynamic. At run time, when a **REDIM** statement is executed, the array is deallocated (if it is already allocated) and then reallocated with the new dimensions. Old array-element values are lost because all numeric elements are reset to 0, and all string elements are reset to null strings.

Note

While it is possible to change the *size* of an array's dimensions with **REDIM**, it is not possible to change the *number* of dimensions. For example, the following statements are legal:

```
' $DYNAMIC
DIM A(50,50)
ERASE A
REDIM A(20,15)     'A still has 2 dimensions
```

However, the following statements are *not* legal, and will produce an "Array already dimensioned" compile-time error:

```
' $DYNAMIC
DIM A(50,50)
ERASE A
REDIM A(5,5,5)     'Changed number of dimensions from 2 to 3
```

■ **See Also**

DIM, ERASE

■ Example

The following program fragment shows a subroutine that deletes a record from a random-access file; this subroutine uses REDIM to allocate a temporary string array (STORE$) to hold the records from STOCK.DAT. After the records are stored in STORE$, STOCK.DAT is closed, deleted, then reopened, and the values in STORE$ are put back into the file. After this is done, the ERASE statement deallocates STORE$.

```
GOSUB OPENFILE
TOTAL% = LOF(1)/50
MAIN:
    PRINT "1) Add a record"
    PRINT "2) Update a record"
    PRINT "3) Delete a record"
    PRINT "4) End program"
    PRINT : INPUT "What is your choice"; BRANCH
    ON BRANCH GOSUB ADDAREC, UPDATEREC, DELETEREC, ENDPROG
    GOTO MAIN

OPENFILE:      'Open STOCK.DAT and allocate random-file buffers
    OPEN "STOCK.DAT" FOR RANDOM AS #1 LEN=50
    FIELD #1, 50 AS RECORD$     'Multiply-defined field
    FIELD #1, 10 AS PART$, 35 AS DESC$, 5 AS QTY$
RETURN
.
.
.
DELETEREC:
    INPUT "Record number to delete: ",REC%
    GET #1, REC% : PRINT DESC$
    INPUT "Is this the correct record"; CH$
    IF CH$ <> "y" THEN GOTO DELETEREC
    GET #1, TOTAL%                'Put the last record where
    PUT #1, REC%                  'the deleted record was
    TOTAL% = TOTAL% - 1
    REDIM STORE$(TOTAL%)          'Allocate temporary array
    FOR I% = 1 TO TOTAL%          'Store STOCK.DAT in array
        GET #1, I%
        STORE$(I%) = RECORD$
    NEXT
    CLOSE #1 : KILL "STOCK.DAT" 'Erase STOCK.DAT
    GOSUB OPENFILE                'Reopen STOCK.DAT
    FOR I% = 1 TO TOTAL%          'Put STORE$ array values
        LSET RECORD$ = STORE$(I%)       'in STOCK.DAT
        PUT #1, I%
    NEXT
    ERASE STORE$                  'Erase array STORE$
RETURN
```

REM Statement

■ Syntax

REM *remark*

■ Action

Allows explanatory remarks to be inserted in a program

■ Remarks

REM statements are not executed, but are output exactly as entered when the program is listed.

REM statements may be branched into from a **GOTO** or **GOSUB** statement. Execution will continue with the first executable statement after the **REM** statement.

A *remark* may be added to the end of a line by preceding the remark with a single quotation mark (') instead of **REM**.

REM statements are also used to introduce compiler metacommands (see Section 10.3, "Using Metacommands").

Important

Do not use the single quotation form of the **REM** statement in a **DATA** statement because it will be considered valid data.

■ Example

The following fragments demonstrate the two different ways to add remarks to a program:

```
DIM V(20)
REM Add the different velocities:
FOR I = 1 TO 20
  SUM = SUM + V(I)
NEXT I
AVERAGE = SUM/20          'Calculate average velocity
```

■ Syntax

RESET

■ Action

Closes all disk files

■ Remarks

RESET closes all open disk files and forces all blocks in memory to be written to disk. Thus, if the machine loses power, all files will be properly updated.

All files must be closed before a disk is removed from its drive.

■ See Also

CLOSE, END, SYSTEM

■ Example

```
RESET
END
```

RESTORE Statement

■ Syntax

RESTORE [[{ *linenumber* | *linelabel*}]]

■ Action

Allows **DATA** statements to be reread from a specified line

■ Remarks

After a **RESTORE** statement without a specified *linenumber* or *linelabel* is executed, the next **READ** statement accesses the first item in the first **DATA** statement in the program.

If *linenumber* or *linelabel* is specified, the next **READ** statement accesses the first item in the specified **DATA** statement.

■ See Also

DATA, READ

■ Example

The RESTORE statement in the following fragment (from a program that generates random bridge hands) allows the program to reread the prompts in the DATA statement so the user can enter new boundary conditions for the different suits:

```
DEFINT A-Z
DIM X(13), PARAM(5,2)
DATA Spades, Hearts, Diamonds, Clubs, Points
NEWVAL:
    CLS
    FOR I = 0 TO 4
        READ SUIT$
        PRINT SUIT$ " (low, high)";
        INPUT PARAM(I,0), PARAM(I,1)
    NEXT
MAIN:
    .
    .
    .
    INPUT "Repeat with same parameters"; CH$
    IF CH$ = "y" THEN GOTO MAIN
    INPUT "Repeat with new parameters"; CH$
    IF CH$ = "y" THEN
        RESTORE
        GOTO NEWVAL
    END IF
    END
```

Partial output:

```
Spades (low, high)? 5,10
Hearts (low, high)? 5,10
Diamonds (low, high)? 6,8
Clubs (low, high)? 6,8
Points (low, high)? 5,15
.
.
.
Repeat with same parameters? n
Repeat with new parameters? y
Spades (low, high)? 4,8
Hearts (low, high)? 4,8
.
.
.
```

RESUME Statement

■ Syntax

RESUME {*linenumber* | *linelabel*}
RESUME [[**0**]]
RESUME NEXT

■ Action

Continues program execution after an error-recovery procedure has been performed

■ Remarks

Depending on where execution is to resume, any one of the three syntaxes shown above may be used as described in the following list:

Statement	Where Execution Resumes
RESUME [[**0**]]	At the statement that caused the error.
RESUME NEXT	At the statement immediately following the one that caused the error, unless the error occurs in a single-line function, in which case **RESUME NEXT** attempts to resume execution at the line containing the function definition. (See Example.)
RESUME *linenumber*	At *linenumber.*
RESUME *linelabel*	At *linelabel.*

A **RESUME** statement that is not in an error-handling routine causes a "RESUME without error" message to be printed.

Statements containing error-handling routines should be compiled with either the On Error or the Resume Next compile option from the Compile... command selection in the QuickBASIC Run menu.

The line specified in a **RESUME** {*linenumber* | *linelabel*} statement must be defined at the level of the program that caused the error. For example, if the error occurs in a subroutine, then *linenumber* or *linelabel* must refer to a line in that subroutine; if the error occurs at the main program level, then *linenumber* or *linelabel* must refer to a line in the main program.

■ Compiler/Interpreter Differences

In the compiler, if an error occurs in a single-line function, both **RESUME** and **RESUME NEXT** attempt to resume program execution at the line containing the function.

■ See Also

ERL, ERR, ERROR, ON ERROR

■ Example

This example has an error-handling routine starting at line 900. In the interpreter, this routine will cause the message `No negative arguments` to be printed for `I` = –1 and –2, then end. In the compiler, however, the error-handling routine will print its message when `I` = –1, then resume at the beginning of the **FOR...NEXT** loop with `I` reset to 4.

```
10 ON ERROR GOTO 900
20 DEF FNTEST(A) = 1 - SQR(A)
30 FOR I = 4 TO -2 STEP -1
40     PRINT I,FNTEST(I)
50 NEXT
60 END

900 'Error-handling routine
910     PRINT "No negative arguments"
920 RESUME NEXT
```

Output:

```
4       -1
3       -.7320509
2       -.4142136
1        0
0        1
-1       No negative arguments
4       -1
3       -.7320509
2       -.4142136
.
.
.
```

RETURN Statement

■ Syntax

RETURN [[{ *linenumber* | *linelabel*}]]

■ Action

Returns execution control from a subroutine

■ Remarks

If no *linenumber* or *linelabel* is given, execution begins with the statement immediately following the last executed **GOSUB** statement. Otherwise, the *linenumber* or *linelabel* in the **RETURN** statement causes an unconditional return from a **GOSUB** subroutine to the specified line. Note, however, that the specified line must be at the program level at which the current subroutine was called. In other words, if the current subroutine was called by another subroutine, the *linenumber* or *linelabel* must refer to a line in the calling subroutine; if the current subroutine was called from the main program, the *linenumber* or *linelabel* must refer to a line in the main program.

A **RETURN** statement cannot be used to return control to a calling program from a subprogram defined by **SUB...END SUB**.

■ See Also

GOSUB

■ Syntax

RIGHT$(*string-expression*,*n*)

■ Action

Returns the right-most *n* characters of *string-expression*

■ Remarks

The argument *string-expression* can be any string variable, string constant, or string expression. If *n* is equal to the number of characters in *string-expression* (that is, *n* = **LEN**(*string-expression*)) then **RIGHT$** returns *string-expression*. If *n* = 0, the null string (length zero) is returned.

■ See Also

LEFT$**, **MID$

■ Example

```
'This program converts names input in the form:
'     Firstname [Middlename] Lastname
'to the form:
'     Lastname, Firstname [Middlename].
'
LINE INPUT "Name: "; NM$
I = 1 : SPPOS = 0
WHILE I
    I = INSTR(SPPOS+1,NM$," ")    'Get position of next space.
    IF I > 0 THEN SPPOS = I
WEND
'SPPOS now points to the position of the last space
IF SPPOS = 0 THEN
    PRINT NM$                     'There was just a last name.
ELSE
    'Everything after last space:
    LASTNAME$ = RIGHT$(NM$,LEN(NM$)-SPPOS)
    'Everything before last space:
    FIRSTNAME$ = LEFT$(NM$,SPPOS-1)
    PRINT LASTNAME$ ", " FIRSTNAME$
END IF
END
```

Sample output:

```
Name: Elwood Albert Swinson
Swinson, Elwood Albert
```

■ Syntax

RMDIR *pathname*

■ Action

Removes an existing directory

■ Remarks

The *pathname* is the name of the directory which is to be deleted. The *pathname* must be a string of less than 128 characters. The directory to be removed must be empty of any files except the working directory ('.') and the parent directory ('..'); otherwise, a "Path not found" or a "Path/file access error" will occur.

RMDIR works like the DOS command of the same name; the syntax in BASIC cannot, however, be shortened to **RD**, as in DOS.

■ See Also

CHDIR, **MKDIR**

■ Example

```
CHDIR "C:\SALES\TEMP"       'Move to \TEMP subdirectory in \SALES
KILL "*.*"                  'Remove all files in \TEMP
CHDIR ".."                  'Move back up to \SALES
RMDIR "TEMP"                'Remove \TEMP subdirectory
```

RND Function

■ Syntax

RND[[(*n*)]]

■ Action

Returns a single-precision random number between 0 and 1

■ Remarks

The same sequence of random numbers is generated each time the program is run unless the random number generator is reseeded (for more information on seeding the random number generator, see the **RANDOMIZE** statement reference page).

The **RND** argument values are described as follows:

RND Argument	Value Returned
$n < 0$	Always restarts the same sequence for any given n
$n > 0$ or n omitted	Generates the next random number in the sequence
$n = 0$	Repeats the last number generated

To produce random *integers* within a given range, use the following formula:

INT ((*upperbound* − *lowerbound* + **1**)***RND** + *lowerbound*)

where *upperbound* is the highest number in the range, and *lowerbound* is the lowest number in the range.

■ Example

The following example draws a polygon with a random shape (3 to 7 sides), size, and location every three seconds:

```
SCREEN 1
DEFINT A-Z
DIM X(6), Y(6)
TIMER ON                          'Enable timer event trapping
ON TIMER(3) GOSUB DRAWPOLY        'Draw a new polygon every 3 seconds
PRINT "Press any key to end program"
INPUT "Press <RETURN> to start",TEST$
LOOP:
    TEST$ = INKEY$                'End program if any key pressed
    IF TEST$ <> "" THEN END ELSE GOTO LOOP

DRAWPOLY:
    CLS                           'Erase old polygon
    N = INT(5*RND + 2)            'N is random number from 2 to 6
    FOR I = 0 TO N
        X(I) = INT(RND*319)       'Get coordinates of vertices of
        Y(I) = INT(RND*199)       'polygon
    NEXT
    PSET (X(N),Y(N))
    FOR I = 0 TO N
        LINE -(X(I),Y(I)),2       'Draw new polygon
    NEXT
RETURN
```

RSET Statement

■ Syntax

RSET *stringvariable=string-expression*

■ Action

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

■ Remarks

If *string-expression* requires fewer bytes than were fielded to *stringvariable*, **LSET** left-justifies the string in the field, and **RSET** right-justifies the string. (Spaces are used to pad the extra positions.) If the string is too long for the field, both **LSET** and **RSET** truncate characters from the right. Numeric values must be converted to strings with **MKD$**, **MKI$**, or **MKS$** before they are **LSET** or **RSET**.

■ See Also

MKD$**, **MKI$, **MKS$**, **LSET**, **PUT**

■ Syntax

RUN [[{ *linenumber* | *filespec*}]]

■ Action

Restarts the program currently in memory, or executes the program specified by *filespec*

■ Remarks

The optional *linenumber* argument specifies the line at which execution will begin. If this argument is not given, execution begins at the first executable line. This line cannot be in a BASIC subprogram or multiline function definition; it must be a line at the main level of the program. Note also that the line must be identified by a number: no alphanumeric labels are permitted in this case, since **RUN** would interpret a label as a *filespec* string expression missing its quotation marks.

The optional *filespec* argument specifies a file to be loaded into memory and run. If there is a program in memory when the command executes, the original program is cleared from memory.

The default *filespec* extension in the compiler is **.EXE**. Therefore, if CATCHALL is the name of an executable file, then the statement

```
RUN "CATCHALL"
```

will execute the program in the file CATCHALL.EXE.

RUN closes all open files and clears the current contents of memory before loading the designated program. The compiler does not support the interpreted BASIC **R** option, which allows all open data files to remain open. If you want to run a new file, yet leave all data files open, use **CHAIN**.

Note

RUN is used to invoke **.EXE** files created by the compiler, or **.EXE** files created by other languages. Compiled programs cannot directly execute **.BAS** source files.

■ See Also

CHAIN

■ Example 1

The following program line transfers program control to the PRICE.EXE file, after closing all open files and deleting the current memory contents:

```
RUN "PRICE"
```

■ Example 2

The following example shows how **RUN** *linenumber* resets all numeric variables to 0: as the line number following RUN increases in lines 60, 70, 80, and 90, the variables in the earlier statements lose their assigned values.

```
10 A = 9
20 B = 7
30 C = 5
40 D = 4
50 PRINT A,B,C,D
60 IF A = 0 THEN 70 ELSE RUN 20
70 IF B = 0 THEN 80 ELSE RUN 30
80 IF C = 0 THEN 90 ELSE RUN 40
90 IF D = 0 THEN END ELSE RUN 50
```

Output:

```
9         7         5         4
0         7         5         4
0         0         5         4
0         0         0         4
0         0         0         0
```

■ Syntax

SADD(*string-expression*)

■ Action

Returns the address of the specified string expression, *string-expression*

■ Remarks

You should use this function with care, since strings in string space can move at any time. Three possible causes of string-space rearrangement are:

1. A string-space compaction as a result of **FRE**
2. Opening or closing a file
3. Allocating a string variable

■ See Also

FRE, PEEK, POKE, VARPTR, VARPTR$

■ Example

```
A$ = "Late arrivals"     'Store string in variable A$
ADD = SADD(A$)           'Get address of A$
LN = LEN(A$) - 1         'Get length of A$
GOSUB PRINTOUT           'Print A$
POKE ADD,ASC("E")        'Change first,
POKE ADD+2,ASC("s")      '       third,
POKE ADD+3,ASC("t")      '       and fourth letters of A$
GOSUB PRINTOUT           'Print new value of A$
END

PRINTOUT:
    FOR I = 0 TO LN
        ASCII = PEEK(ADD + I)  'Get value at address
        PRINT CHR$(ASCII);     'Convert value to character,
    NEXT                       '    and print
    PRINT                      'Print new line
RETURN
```

Output:

```
Late arrivals
Fast arrivals
```

■ Syntax

SCREEN(*row*,*column*[,*z*])

■ Action

Reads a character or its color from a specified screen location

■ Remarks

The *row* is a valid numeric expression returning an unsigned integer.

The *column* is a valid numeric expression returning an unsigned integer.

The argument *z* is a valid numeric expression.

The ordinate of the character at the specified coordinates is stored in the numeric variable. If the optional parameter z is given and is non-zero, the color attribute for the character is returned instead.

■ Examples

If the character at (10,10) is A, then the function would return 65, the ASCII code for A:

```
X=SCREEN(10,10)
```

The following example returns the color attribute of the character in the upper-left corner of the screen:

```
X=SCREEN(1,1,1)
```

SCREEN Statement

■ Syntax

SCREEN [[*mode*]] [[,[[*colorswitch*]]][[,[[*apage*]]][[,[[*vpage*]]]]

■ Action

Sets the specifications for the display screen

■ Remarks

The **SCREEN** statement is chiefly used to select a screen mode appropriate for a particular display-hardware configuration. Supported hardware configurations and screen modes are described below.

MDPA with Monochrome Display: Mode 0

The IBM Monochrome Display and Printer Adapter (MDPA) is used to connect only to a monochrome display. Programs written for this configuration must be text mode only.

CGA with Color Display: Modes 0, 1, and 2

The IBM Color Graphics Adapter (CGA) and Color Display are typically paired with each other. This hardware configuration permits running of text mode programs, and both medium-resolution and high-resolution graphics programs.

EGA with Color Display: Modes 0, 1, 2, 7, and 8

The five screen modes 0, 1, 2, 7, and 8 allow you to interface to the IBM Color Display when it is connected to an IBM Enhanced Graphics Adapter (EGA). If EGA switches are set for CGA compatibility, programs written for modes 1 and 2 will run just as they would with the CGA. Modes 7 and 8 are similar to modes 1 and 2, except that a wider range of colors are available in modes 7 and 8.

EGA with Enhanced Color Display: Modes 0, 1, 2, 7, and 8

With the EGA/IBM Enhanced Color Display configuration, modes 0, 1, 2, 7, and 8 are virtually identical to their EGA/Color Display counterparts. Two possible differences are as follows:

1. In mode 0, the border color cannot be the same as for the EGA/Color Display because the border cannot be set on an Enhanced Color Display when it is in 640 x 350 text mode.
2. The quality of the text is better on the Enhanced Color Display (an 8 x 14 character box for Enhanced Color Display versus an 8 x 8 character box for Color Display).

EGA with Enhanced Display: Mode 9

The full capability of the Enhanced Color Display is taken advantage of in this mode. Mode 9 allows the highest resolution possible for the EGA/Enhanced Color Display configuration. Programs written for this mode will not work for any other hardware configuration.

EGA with Monochrome Display: Mode 10

The IBM Monochrome Display can be used to display monochrome graphics at a very high resolution in this mode. Programs written for this mode will not work for any other hardware configuration.

Arguments

The *mode* argument is an integer expression with legal values 0, 1, 2, 7, 8, 9, and 10. All other values are illegal. Selection of a mode argument depends primarily on your program's anticipated display hardware, as described above.

Each of the **SCREEN** modes is described individually in the following paragraphs.

SCREEN 0

- Text mode only
- Either 40 x 25 or 80 x 25 text format with character-box size of 8 x 8 (8 x 14 with EGA)
- Assignment of 16 colors to any of 2 attributes
- Assignment of 16 colors to any of 16 attributes (with EGA)

SCREEN 1

- 320 x 200 pixel medium-resolution graphics
- 80 x 25 text format with character-box size of 8 x 8
- Assignment of 16 colors to any of 4 attributes
- Supports both EGA and CGA

SCREEN 2

- 640 x 200 pixel medium-resolution graphics
- 40 x 25 text format with character-box size of 8 x 8
- Assignment of 16 colors to any of 2 attributes
- Supports both EGA and CGA

SCREEN 7

- 320 x 200 pixel medium-resolution graphics
- 40 x 25 text format with character-box size of 8 x 8
- 2, 4, or 8 memory pages with 64K, 128K, or 256K of memory, respectively, installed on the EGA
- Assignment of any of 16 colors to 16 attributes
- EGA required

SCREEN 8

- 640 x 200 pixel high-resolution graphics
- 80 x 25 text format with character-box size of 8 x 8
- 1, 2, or 4 memory pages with 64K, 128K, or 256K of memory, respectively, installed on the EGA
- Assignment of any of 16 colors to 16 attributes
- EGA required

SCREEN 9

- 640 x 350 pixel enhanced-resolution graphics
- 80x25 text format with character-box size of 8 x 14,
- Assignment of either 64 colors to 16 attributes (more than 64K of EGA memory), or 16 colors to 4 attributes (64K of EGA memory)
- Two display pages if 256K of EGA memory installed
- EGA required

SCREEN 10

- 640 x 350 enhanced-resolution graphics
- 80 x 25 text format with character-box size of 8 x 14
- Two display pages if 256K of EGA memory installed
- Assignment of up to 9 pseudo-colors to 4 attributes, refer to Tables 11.4 and 11.5
- EGA required

Table 11.4

Default Attributes: SCREEN 10, Monochrome Display

Attribute Value	Displayed Pseudo-Color
0	Off
1	On, normal intensity
2	Blink
3	On, high intensity

Table 11.5

Color Values: SCREEN 10, Monochrome Display

Color Value	Displayed Pseudo-Color
0	Off
1	Blink, off to on
2	Blink, off to high intensity
3	Blink, on to off
4	On
5	Blink, on to high intensity
6	Blink, high intensity to off
7	Blink, high intensity to on
8	High intensity

For both composite monitors and composite TVs, the *colorswitch* is a numeric expression that is either true (non-zero) or false (zero). A value of zero disables color and permits display of black and white images only. A non-zero value permits color. The meaning of the *colorswitch* argument is inverted in **SCREEN** mode 0.

For hardware configurations that include an EGA and enough memory to support multiple-screen pages, two arguments are available. These *apage* and *vpage* arguments determine the "active" and "visual" memory pages. The active page is the area in memory where graphics statements are written; the visual page is the area of memory that is displayed on the screen.

Animation can be achieved by alternating the display of graphics pages. The goal is to display the visual page with completed graphics output, while executing graphics statements in one or more active pages. A page is displayed only when graphics output to that page is complete. Thus, the following program fragment is typical:

```
SCREEN 7,,1,2    'work in page 1, show page 2
.
.    Graphics output to page 1
.    while viewing page 2
.
SCREEN 7,,2,1    'work in page 2, show page 1
.
.    Graphics output to page 2
.    while viewing page 1
.
```

The number of pages available depends on the **SCREEN** mode and the amount of available memory, as described in Table 11.6.

Table 11.6
SCREEN Mode Specifications

Mode	Resolution	Attribute Range	Color Range	EGA Memory	Pages	Page Size
0	40–column text	NA	0–15[a]	NA	1	2K
	80–column text	NA	0–15[a]	NA	1	4K
1	320 x 200	0–3[b]	0–3	NA	1	16K
2	640 x 200	0–1[b]	0–1	NA	1	16K
7	320 x 200	0–15	0–15	64K	2	32K
				128K	4	
				256K	8	
8	640 x 200	0–15	0–15	64K	1	64K
				128K	2	
				256K	4	
9	640 x 350	0–3	0–15	64K	1	64K
		0–15	0–63	128K	1	128K
		0–15	0–63	256K	2	
10	640 x 350	0–3	0–8	128K	1	128K
				256K	2	

[a] Numbers in the range 16–31 are blinking versions of the colors 0–15.
[b] Attributes applicable only with EGA.
NA=Not Applicable

Attributes and Colors

For various screen modes and display hardware configurations, different attribute and color settings exist. (See the **PALETTE** statement reference pages for a discussion of attribute and color number.) The majority of these attribute and color configurations are summarized in Table 11.7.

Table 11.7

Default Attributes and Colors for Most SCREEN Modes

Attributes for Mode			Color Display		Monochrome Display	
1,9	2	0,7,8,9[b]	Number[c]	Color	Number[c]	Color
0	0	0	0	Black	0	Off
		1	1	Blue		(Underlined)[a]
		2	2	Green	1	On[a]
		3	3	Cyan	1	On[a]
		4	4	Red	1	On[a]
		5	5	Magenta	1	On[a]
		6	6	Brown	1	On[a]
		7	7	White	1	On[a]
		8	8	Gray	0	Off
		9	9	Light Blue		High intensity (underlined)
		10	10	Light Green	2	High intensity
1		11	11	Light Cyan	2	High intensity
		12	12	Light Red	2	High intensity
2		13	13	Light Magenta	2	High intensity
		14	14	Yellow	2	High intensity
3	1	15	15	High-intensity White	0	Off

[a] Off when used for background
[b] With EGA memory > 64K
[c] Only for mode 0 monochrome

The default foreground colors for the various modes are given in Table 11.8.

Table 11.8

Default Foreground Colors

Mode	Default Foreground Attribute		Default Foreground Color	
	Color/Enhanced[a] Display	Monochrome Display	Color/Enhanced[a] Display	Monochrome Display
0	7	7	7	1
1	3	NA	15	NA
2	1	NA	15	NA
7	15	NA	15	NA
8	15	NA	15	NA
9	3[b]	NA	63	NA
10	NA	3	NA	8

[a] IBM Enhanced Color Display
[b] 15 if greater than 64K of EGA memory
NA=Not Applicable

■ Syntax

SGN(*numeric-expression*)

■ Action

Indicates the sign of a numeric expression

■ Remarks

If *numeric-expression* > 0, **SGN**(*numeric-expression*) returns 1.

If *numeric-expression* = 0, then **SGN**(*numeric-expression*) returns 0.

If *numeric-expression* < 0, then **SGN**(*numeric-expression*) returns –1.

■ Example

The following program calculates and prints the solution set for the input quadratic (or second-degree) equation. If TEST is negative, the program branches to NOSOL (since SGN(TEST) + 2 would equal 1); if TEST equals zero, the program branches to ONESOL (since SGN(TEST) + 2 would equal 2); and if TEST is positive, the program branches to TWOSOL (since SGN(TEST) + 2 would equal 3).

```
'Input coefficients of quadratic equation ax^2 + bx + c = 0:
INPUT;"a = ",A : INPUT;",  b = ",B : INPUT ",  c = ",C
TEST = B^2 - 4*(A*C)
ON (SGN(TEST) + 2) GOTO NOSOL,ONESOL,TWOSOL
NOSOL:
    PRINT "This equation has no real-number solutions."
    END
ONESOL:
    PRINT "This equation has one solution, x = ";-B/(2*A)
    END
TWOSOL:
    PRINT "This equation has two solutions: ";
    PRINT (-B + SQR(TEST))/(2*A),(-B - SQR(TEST))/(2*A)
    END
```

Sample output:

```
a = 12,  b = -5,  c = -2
This equation has two solutions:  .6666667      -.25
```

■ Syntax

SHARED *variable* [[,*variable*...]]

■ Action

Gives a subprogram access to variables declared in the main program without having to pass them as parameters.

■ Remarks

The argument *variable* is either a variable name or an array name followed by ().

By using either the **SHARED** statement in a subprogram, or the **SHARED** attribute with **COMMON** or **DIM** at the main program level, you can use variables in a subprogram without passing them as parameters. The **SHARED** *attribute* shares variables among all subprograms in a module, while the **SHARED** *statement* affects variables within a single subprogram.

Note

The **SHARED** statement only shares variables within a single compiled module. It does not share variables with programs in the user library, or with subprograms compiled separately and run outside the QuickBASIC environment.

The **SHARED** statement must appear only within a named subprogram; if **SHARED** occurs outside a subprogram, a subroutine error occurs. See the **SUB...END SUB** statement reference pages.

■ See Also

COMMON, DIM, SUB...END SUB

■ Example

The following program calls a subprogram CONVERT that converts the input decimal number to its string representation in the given new base. The string N$ is shared by the subprogram and the main program.

```
DEFINT A-Z
START:
    INPUT "Decimal number (input 0 to quit): ",DECIMAL
    IF DECIMAL <= 0 THEN END
    INPUT "New base: ",NEWBASE
    N$ = ""
    PRINT DECIMAL "base 10 equals ";
    WHILE DECIMAL
        CALL CONVERT (DECIMAL,NEWBASE)
        DECIMAL = DECIMAL\NEWBASE
    WEND
    PRINT N$ " base" NEWBASE
    PRINT
    GOTO START

SUB CONVERT (D,NB) STATIC
    SHARED N$
    R = D MOD NB
    IF R < 10 THEN DIGIT$ = STR$(R) ELSE DIGIT$ = CHR$(R+55)
    N$ = RIGHT$(DIGIT$,1) + N$
END SUB
```

Sample output:

```
Decimal number (input 0 to quit): 238
New base: 15
 238 base 10 equals 10D base 15

Decimal number (input 0 to quit): 88
New base: 3
 88 base 10 equals 10021 base 3

Decimal number (input 0 to quit): 0
```

SHELL Statement

■ Syntax

SHELL [[*commandstring*]]

■ Action

Exits the BASIC program, runs a **.COM**, **.EXE**, or **.BAT** program (or a built-in DOS command such as **DIR** or **TYPE**), and returns to the BASIC program at the line after the **SHELL** statement

■ Remarks

The *commandstring* must be a valid string expression, containing the name of a program to run and (optionally) command arguments.

A **.COM**, **.EXE**, or **.BAT** program or DOS function that runs under the **SHELL** statement is called a "child process." Child processes are executed by **SHELL** loading and running a copy of **COMMAND.COM** with the **/c** option; this option allows any parameters in *commandstring* to be passed to the child process. It also allows redirection of standard input and output, and execution of built-in commands such as **DIR**, **PATH**, and **SORT**.

The program name in *commandstring* may have any extension you wish. If no extension is supplied, **COMMAND.COM** will look for a **.COM** file, then an **.EXE** file, and finally, a **.BAT** file. If **COMMAND.COM** is not found, **SHELL** will issue a "File not found" message. No error is generated if **COMMAND.COM** cannot find the file specified in *commandstring*.

Any text separated from the program name by at least one blank will be processed by **COMMAND.COM** as program parameters.

BASIC remains in memory while the child process is running. When the child finishes, BASIC continues.

SHELL with no *commandstring* will give you a new **COMMAND.COM** shell. You may now do anything that **COMMAND.COM** allows. When ready to return to BASIC, enter the DOS command, **EXIT**.

■ Example 1

```
SHELL    'get a new COMMAND

A>DATE

A>EXIT
```

■ Example 2

```
'This program will copy all files in this directory modified
'on a certain date to the drive and directory you specify.
LINE INPUT "Enter target drive and directory: ",PATHSPEC$
SHELL "DIR > DIRFILE"     'Store directory listing in DIRFILE
START:
   OPEN "DIRFILE" FOR INPUT AS #1
   INPUT "Enter date (MM-DD-YY): ",MDATE$
   PRINT
   WHILE NOT EOF(1)       'Read DIRFILE, test for input date
      LINE INPUT #1, TMP$
      IF INSTR(TMP$,MDATE$) > 0 AND LEFT$(TMP$,1) <> "." THEN
         GOSUB GETNAME
         IF FILESPEC$ <> "DIRFILE" THEN
            DOLINE$ = "copy " + FILESPEC$ + "  " + PATHSPEC$
            PRINT DOLINE$
            SHELL DOLINE$
         END IF
      END IF
   WEND
   CLOSE #1
   PRINT : INPUT "New date"; R$
   CLS
   IF R$ = "y" OR R$ = "Y" THEN GOTO START
   KILL "DIRFILE"
   END
```

```
GETNAME:          'This subroutine gets the file name and
                  'extension from the directory-listing line.
   FILESPEC$ = ""
   I = 1
   'Eight significant characters for DOS file names:
   WHILE I < 9
      CH$ = MID$(TMP$,I,1)
      IF CH$ <> " " THEN
         FILESPEC$ = FILESPEC$ + CH$
         I = I + 1
      ELSE
         I = 9
      END IF
   WEND
   EXT$ = MID$(TMP$,10,3)        'Check for extension
   IF EXT$ > "   " THEN FILESPEC$ = FILESPEC$ + "." + EXT$
RETURN
```

Sample output:

```
Enter target drive and directory:  c:\bas\temp
Enter date (MM-DD-YY): 2-25-86

copy CONF.DAT  c:\bas\temp
copy TEST.BAS  c:\bas\temp
copy TEMPFILE  c:\bas\temp

New date? n
```

■ Syntax

SIN(x)

■ Action

Returns the sine of x, where x is in radians

■ Remarks

COS(x) = **SIN**(x + **3.14159/2**).

SIN is calculated with single-precision accuracy, unless the argument x is a double-precision value, in which case **SIN** is calculated with double-precision accuracy.

■ See Also

COS

■ Example

The following program plots the graph of the polar equation $r = 1 + \sin n(\theta)$. This figure is sometimes known as a cardioid, owing to its resemblance to a heart when n equals 1.

```
PI = 3.141593
SCREEN 1 : COLOR 1,1              'Medium resolution, blue background
WINDOW (-3,-2)-(3,2)              'Convert screen to Cartesian
                                  'coordinates

INPUT "Number of petals = ", N
CLS
PSET (1,0)                        'Set initial point
FOR ANGLE = 0 TO 2*PI STEP .02
    R = 1 + SIN(N*ANGLE)          'Polar equation for "flower"
    X = R * COS(ANGLE)            'Convert polar coordinates to
    Y = R * SIN(ANGLE)            'Cartesian coordinates
    LINE -(X,Y)                   'Draw line from previous point
                                  'to new point
NEXT
```

SOUND Statement

■ Syntax

SOUND *frequency,duration*

■ Action

Generates sound through the speaker

■ Remarks

The *frequency* is the desired frequency in hertz (cycles/second). This must be a numeric expression returning an unsigned integer in the range 37–32,767.

The *duration* is the duration in clock ticks. (Clock ticks occur 18.2 times per second.) The *duration* must be a numeric expression returning an unsigned integer in the range 0–65,535.

If the duration is zero, any current **SOUND** statement that is running is turned off. If no **SOUND** statement is currently running, a **SOUND** statement with a duration of zero has no effect.

■ See Also

PLAY

■ Example 1

This short fragment creates fifty random sounds whose frequency ranges from 37–5,000 hertz, and whose duration ranges from one clock tick to one second:

```
FOR I = 1 TO 50
    FREQ = INT(4964*RND) + 37
    DUR = INT(173*RND)/10 + 1
    SOUND FREQ, DUR
NEXT
```

■ Example 2

This next program fragment produces a glissando up and down:

```
FOR I = 440 TO 1000 STEP 5
   SOUND I, I/1000
NEXT
FOR I = 1000 TO 440 STEP -5
  SOUND I, I/1000
NEXT
```

SPACE$ Function

■ Syntax

SPACE$(*n*)

■ Action

Returns a string of spaces of length *n*

■ Remarks

The expression *n* is rounded to an integer and must be in the range 0–32,767.

■ See Also

SPC

■ Example

```
FOR I=1 TO 5
   X$=SPACE$(I)
   PRINT X$;I
NEXT I
```

Output:

```
1
 2
  3
   4
    5
```

■ Syntax

SPC(*n*)

■ Action

Skips *n* spaces in a **PRINT** statement

■ Remarks

SPC may only be used with **PRINT** and **LPRINT** statements. The argument *n* must be in the range 0–32,767. A semicolon (;) is assumed to follow the **SPC**(*n*) command.

■ See Also

SPACE$

■ Example

```
PRINT "OVER" SPC(15) "THERE"
```

Output:

```
OVER               THERE
```

SQR Function

■ Syntax

SQR(*n*)

■ Action

Returns the square root of *n*

■ Remarks

The argument *n* must be ≥ 0.

■ Example

The following program plots the graph of $y = \sqrt{-x}$ for $-9 \leq x < 0$, and $y = \sqrt{x}$ for $0 \leq x \leq 9$:

```
SCREEN 1 : COLOR 1          'Low-resolution color graphics mode
WINDOW (-9,-.25)-(9,3.25)   'Convert screen to Cartesian
                            'coordinates.
LINE (-9,0)-(9,0)           'Draw X-axis.
LINE (0,-.25)-(0,3.25)      'Draw Y-axis.

FOR X = -9 TO 9
    LINE(X,.04)-(X,-.04)    'Put tick marks on X-axis.
NEXT

FOR Y = .25 TO 3.25 STEP .25
    LINE (-.08,Y)-(.12,Y)   'Put tick marks on Y-axis.
NEXT

PSET (-9,3)                 'Plot the first point of the function.
FOR X = -9 TO 9 STEP .25
    Y = SQR(ABS(X))         'SQR argument cannot be negative.
    LINE -(X,Y),2           'Draw a line to the next point.
NEXT
```

■ Syntax

STATIC *variable* [[,*variable...*]]

■ Action

Designates simple variables or arrays as local to a function definition or subprogram, and preserves their values when the subprogram or function is exited and then re-entered

■ Remarks

The argument *variable* is either a variable name or an array name followed by an integer constant in parentheses. This integer constant represents the number of dimensions in the array, not the actual value of the dimensions.

The **STATIC** statement can appear only within a named subroutine or multiline function definition; if the statement occurs outside either one of these, a subroutine error (SB) occurs (see the **SUB...END SUB** and **DEF FN** statements reference pages).

Normally, simple variables or arrays that are declared or referred to in a subprogram are considered local to that program, with initial values of zero or null string assumed. However, if the subprogram is exited and then reentered, the values contained in the variables may have changed. By declaring the variables in a **STATIC** statement, you guarantee that subprogram variables will retain their previous values.

Simple variables or arrays declared within a **STATIC** statement override any shared variables or arrays with the same name.

Usually, variables used in multiline function definitions (**DEF FN**) are global; however, you can use the **STATIC** statement inside **DEF FN** to declare a variable as local to that function only.

Note

Do not confuse the **STATIC** statement with the **STATIC** that is part of the **SUB...END SUB** syntax or with the $**STATIC** metacommand. The **STATIC** attribute in **SUB...END SUB** shows that the subprogram is nonrecursive, while $**STATIC** statically allocates memory for arrays.

■ See Also

DEF FN, SHARED, SUB...END SUB

■ Example 1

This example contrasts the **STATIC** and **SHARED** statements within a subprogram, AUGMENT. The variables R and N are both local to this subprogram, while the variables REP and NUM are shared between the main program and the subprogram. The STATIC R,N statement ensures that R and N retain the last values assigned to them each time the main program calls the subprogram.

```
REP = 0 : NUM = 0
PRINT "Before loop, REP = ";REP;", NUM = ";NUM;
PRINT ", R = ";R;", N = ";N
FOR I = 1 TO 10
   CALL AUGMENT
NEXT
PRINT "After loop, REP = ";REP;", NUM = ";NUM;
PRINT ", r = ";R;", n = ";N
END
SUB AUGMENT STATIC
   SHARED REP,NUM     'SHARED WITH MAIN PROGRAM
   STATIC R,N         'NOT SHARED WITH MAIN PROGRAM
   R = R + 1          'BOTH R & N INITIALLY EQUAL 0
   N = N + 2
   REP = R
   NUM = N
END SUB
```

Output:

```
Before loop, REP = 0 , NUM = 0 , R = 0 , N = 0
After loop, REP = 10 , NUM = 20 , R = 0 , N = 0
```

■ Example 2

The following program searches for every occurrence of a certain string expression (stored in the variable OLD$) in the specified file and replaces that string with the string stored in NW$. The name of the file with these changes is the old filename with the extension .NEW.

The program also prints the number of substitutions and the number of lines changed:

```
INPUT "Name of file";F1$
INPUT "String to replace";OLD$
INPUT "Replace with";NW$
REP = 0 : NUM = 0
M = LEN(OLD$)
OPEN F1$ FOR INPUT AS #1
CALL EXTENSION
OPEN F2$ FOR OUTPUT AS #2
WHILE NOT EOF(1)
   LINE INPUT #1, TEMP$
   CALL SEARCH
   PRINT #2, TEMP$
WEND
CLOSE
PRINT "There were ";REP;" substitutions in ";NUM;" lines."
PRINT "Substitutions are in file ";F2$
END

SUB EXTENSION STATIC
   SHARED F1$,F2$
   MARK = INSTR(F1$,".")
   IF MARK = 0 THEN
      F2$ = F1$ + ".NEW"
   ELSE
      F2$ = LEFT$(F1$,MARK - 1) + ".NEW"
   END IF
END SUB
```

```
SUB SEARCH STATIC
   SHARED TEMP$,OLD$,NW$,REP,NUM,M
   STATIC R
   MARK = INSTR(TEMP$,OLD$)
   WHILE MARK
      PART1$ = LEFT$(TEMP$,MARK - 1)
      PART2$ = MID$(TEMP$,MARK + M)
      TEMP$ = PART1$ + NW$ + PART2$
      R = R + 1
      MARK = INSTR(TEMP$,OLD$)
   WEND
   IF REP = R THEN
       EXIT SUB
   ELSE
       REP = R
       NUM = NUM + 1
   END IF
END SUB
```

Sample output:

```
Name of file? CHAP1.S
String to replace? CHAPTER 1
Replace with? INTRODUCTION
There were 23 substitutions in 19 lines.
Substitutions are in file CHAP1.NEW
```

The file CHAP1.NEW now contains every line in CHAP1.S, with each occurrence of the string CHAPTER 1 replaced by INTRODUCTION.

■ Syntax

STICK(*n*)

■ Action

Returns the *x* and *y* coordinates of the two joysticks

■ Remarks

The argument *n* is a numeric expression returning an unsigned integer in the range 0 to 3:

Value of *n*	Value Returned by STICK
0	The *x* coordinate for joystick A: also stores the *x* and *y* values for both joysticks for function calls 1–3
1	The *y* coordinate of joystick A
2	The *x* coordinate of joystick B
3	The *y* coordinate of joystick B

■ Example

This example displays the value of the *x,y* coordinate for joystick A:

```
CLS
LOCATE 1,1
PRINT "X=5";STICK(0)
PRINT "Y=5";STICK(1)
INPUT "Press RETURN to continue program"; DUMMY$
.
.
.
```

STOP Statement

■ Syntax

STOP

■ Action

- Terminates program execution
- Closes all files
- Exits to the operating system

■ Remarks

STOP statements can be used anywhere in a program to terminate execution. **STOP** is often used for debugging. When a **STOP** is encountered during in-memory execution, the following message is printed:

```
Stop statement executed. Press any key.
```

■ Compiler/Interpreter Differences

In the interpreter, the **STOP** statement does not close files or return control to the command level. Execution is resumed by using a **CONT** command in direct mode; the **CONT** command is not supported in the compiler.

With the compiler, **STOP** closes all open files.

If the **/d**, **/e**, or **/x** compiler options are given on the command line, or if the Debug, On Error, or Resume Next compile options are chosen from the QuickBASIC menu, the **STOP** message prints the line number at which execution has stopped, provided your program has line numbers. If there is no line number associated with the **STOP** statement, specifying these options will cause the most recent line number to be printed. If your program has no line numbers at all, then the line number printed is 0.

■ Example

```
INPUT A,B,C
K = A^2*5.3 : L = B^3/.26
PRINT K,L
STOP
M = C*K+100 : PRINT M
```

Output:

```
? 1,2,3
 5.3              30.76923

Stop statement executed. Press any key.
```

STR$ Function

■ Syntax

STR$(*numeric-expression*)

■ Action

Returns a string representation of the value of *numeric-expression*

■ Remarks

If *numeric-expression* is positive, the string returned by **STR$** contains a leading blank.

The **VAL** function complements **STR$**.

■ See Also

VAL

■ Example

The following example contains a multiline function definition that uses **STR$** to convert a number to its string representation; the function then strips out the leading and trailing blanks that BASIC prints with numeric output.

```
DEF FNNUM$(X)
    X$ = STR$(X)
    LENGTH = LEN(X$)
    IF LEFT$(X$,1) <> "-" THEN LENGTH = LENGTH - 1
    FNNUM$ = RIGHT$(X$,LENGTH)
END DEF

WHILE NUM
    INPUT "Find cosine of: ",NUM        'Enter 0 to end.
    IF NUM THEN PRINT "COS(" FNNUM$(NUM) ") = " COS(NUM)
WEND
```

Sample output:

```
Find cosine of: 3.1
COS(3.1) = -.9991351
Find cosine of: 0
```

■ Syntax

STRIG(*n*)

■ Action

Returns the status of a specified joystick trigger

■ Remarks

The numeric expression *n* is an unsigned integer in the range 0–3, designating which trigger is to be checked. The values returned by **STRIG**(*n*) for different values of *n* are described in the following list:

Argument	Value Returned
0	–1 if trigger A was pressed since the last `STRIG(0)` statement, 0 if not
1	–1 if trigger A is currently down, 0 if not
2	–1 if trigger B was pressed since the last `STRIG(2)` statement, 0 if not
3	–1 if trigger B is currently down, 0 if not

A **STRIG ON** statement must be executed to enable **STRIG**(*n*). After **STRIG ON**, the compiler checks between each statement to see if a trigger has been pressed.

When a joystick event is trapped by an **ON STRIG** statement, the event that caused the trap is destroyed. Therefore, you cannot use **STRIG**(*n*) inside the event-trapping subroutine to determine which joystick trigger set off the trap. You can, however, use **STRIG**(*n*) to get information about trigger activity while the program is inside the event-trapping subroutine.

If you wish to perform different procedures for various joysticks, you must set up a different subroutine for each joystick, rather than including all the procedures in a single subroutine.

■ Compiler/Interpreter Differences

With the compiler, you must use the **/v** or **/w** option on the command line if a program contains a **STRIG** statement. If you are compiling from the QuickBASIC menu, use either the Event Trapping or the Checking Between Statements compile option. These options allow the compiler to function correctly when event-trapping subroutines are included in a program.

■ See Also

STRIG, STRIG ON, STRIG OFF, STRIG STOP

■ Compiler/Interpreter Differences

See compiler note under "ON STRIG Statement."

■ Example

```
ATRIG = 0
'Wait for trigger A to be pressed:
WHILE NOT ATRIG : ATRIG = STRIG(0) : WEND
'As long as trigger A is down, beep:
WHILE ATRIG : ATRIG = STRIG(1) : BEEP : WEND
.
.
.
```

■ Syntax

STRIG ON
STRIG OFF
STRIG STOP

■ Action

STRIG ON enables event trapping of joystick activity, **STRIG OFF** disables event trapping of joystick activity, and **STRIG STOP** inhibits event trapping of joystick activity.

■ Remarks

The **STRIG ON** statement enables joystick event trapping by an **ON STRIG** statement (see **STRIG** function reference page). While trapping is enabled, and if a non-zero line number is specified in the **ON STRIG** statement, BASIC checks between every statement to see if the joystick trigger has been pressed.

The **STRIG OFF** statement disables event trapping. If a subsequent event occurs (i.e., if the trigger is pressed), it will not be remembered when the next **STRIG ON** is invoked.

The **STRIG STOP** statement inhibits event trapping. If an event occurs it is remembered, and the event trap takes place as soon as trapping is re-enabled.

■ See Also

STRIG

STRING$ Function

■ Syntax

STRING$(*m*,*n*)
STRING$(*m*,*string-expression*)

■ Action

Returns a string of length *m* whose characters all have ASCII code *n* or whose characters are all the first character of *string-expression.*

■ Example 1

```
DASH$ = STRING$(10,45)
PRINT DASH$;"MONTHLY REPORT";DASH$
```

Output:

```
----------MONTHLY REPORT----------
```

■ Example 2

The following program uses **STRING$** to generate a bar graph.

```
PRINT TAB(7);"Daily Mean Temperature in Seattle" : PRINT
FOR MONTH = 1 TO 12 STEP 2    'Get data for each month and graph.
    READ MONTH$, TEMP
    PRINT MONTH$;" +"; STRING$(TEMP-35,"*") 'Print TEMP-35 stars.
    PRINT "     |"
NEXT MONTH
PRINT "    +";                 'Print horizontal line.
FOR X = 1 TO 7
    PRINT "----+";
NEXT X
PRINT
FOR X = 4 TO 39 STEP 5          'Print temperature labels.
    PRINT TAB(X); X+31;
NEXT X
PRINT

DATA Jan, 40, Mar, 46, May, 56
DATA Jul, 66, Sep, 61, Nov, 46
```

Output:

```
      Daily Mean Temperature in Seattle

Jan +*****
    |
Mar +***********
    |
May +**********************
    |
Jul +*******************************
    |
Sep +**************************
    |
Nov +***********
    |
    +----+----+----+----+----+----+----+
    35   40   45   50   55   60   65   70
```

■ Syntax

SUB *global-name*[[(*parameter-list*)]] **STATIC**
.
.
.
[[**EXIT SUB**]]
.
.
.
END SUB

■ Action

Marks the beginning and end of a subprogram

■ Remarks

SUB takes the following arguments:

Argument	Description
global-name	A variable name up to 31 characters long. This name cannot appear in any other **SUB** statement in the same program or the user library. If duplicate names are present, the linker will bind to the local subroutine in the user library, and no error message will appear.
parameter-list	Contains the names of simple variables and arrays passed to the subprogram by a **CALL** statement in the main program; each name is separated from a preceding name in the list by a comma. Note that these variables and arrays are passed by reference, so any change to an argument's value in the subprogram also changes its value in the calling program. An array name in a **SUB** statement is followed by an integer constant in parentheses. This integer constant represents the number of dimensions in the array, not the actual value of the dimensions.

A subprogram is similar to a multiline function. However, unlike a multiline function, a subprogram does not return a value associated with its name, and therefore cannot appear as part of an expression.

The **STATIC** attribute shows that the subprogram is nonrecursive; that is, it does not contain an instruction that causes the subprogram to call itself, or call a second subprogram, that in turn calls the first subprogram. This version of BASIC supports only nonrecursive subprograms, and generates an error if you omit **STATIC**.

SUB and **END SUB** mark, respectively, the beginning and end of a subprogram. You can also use the optional **EXIT SUB** statement to exit a subprogram.

Subprograms are called by a **CALL** statement. When a subprogram is exited and later re-entered, the value in a particular subprogram variable may be affected by another part of your program. To guarantee that the variable retains its assigned value upon re-entry to the subprogram, use the **STATIC** statement.

Any subprogram variables or arrays are considered local to that subprogram, unless they are explicitly declared as shared variables in a **SHARED** statement.

User defined functions are not permitted inside a **SUB...END SUB** block.

Note

You cannot use **GOSUB**, **GOTO**, or **RETURN** to enter or exit a subprogram

---

■ **See Also**

CALL, SHARED, STATIC

■ Example

In this example, the main program calls a subprogram, LINESEARCH, which searches for the given string, P$, in each line of input from file F$. When the subprogram finds P$ in a line, it prints the line, along with the number of the line.

```
INPUT "File to be searched";F$
INPUT "Pattern to search for";P$
OPEN F$ FOR INPUT AS #1
WHILE NOT EOF(1)
   LINE INPUT #1, TEST$
   CALL LINESEARCH(TEST$,P$)
WEND

SUB LINESEARCH(TEST$,P$) STATIC
   STATIC NUM
   NUM = NUM + 1
   X = INSTR(TEST$,P$)
   IF X = 0 THEN
      EXIT SUB
   ELSEIF X > 0 THEN
      PRINT "Line #";NUM;": ";TEST$
   END IF
END SUB
```

The output from the above example might look like this:

```
File to be searched? search.bas
Pattern to search for? sub
```

Output:

```
Line # 9: SUB LINESEARCH(TEST$,P$) STATIC
Line # 14:    EXIT SUB
Line # 18: END SUB
```

■ Syntax

SWAP *variable1,variable2*

■ Action

Exchanges the values of two variables

■ Remarks

Any type variable may be swapped (integer, single precision, double precision, string), but the two variables must be of the same type or a "Data type conflict" error results.

If the second variable is not already defined when **SWAP** is executed, an "Illegal function call" error will result.

■ Example

The following program fragment shows a subroutine (SORTREC) that sorts the elements of a string array in ascending order; SORTREC uses **SWAP** to exchange array elements that are out of order.

```
.
.
.
CALL SORTREC(RECORD$())
.
.
.
END

SUB SORTREC(REC$(1)) STATIC
    'Subroutine for sorting the elements of a string array
    'from low (A) to high (Z):
    MAX = UBOUND(REC$)
    GAP = MAX/2
    WHILE GAP > 0
        FOR I = 1 TO GAP
            FLIPS = -1
            WHILE FLIPS
                FLIPS = 0
                FOR J = I TO MAX-GAP STEP GAP
                    IF REC$(J) > REC$(J+GAP) THEN
                        SWAP REC$(J),REC$(J+GAP)
                        FLIPS = -1
                    END IF
                NEXT
            WEND
       NEXT
       GAP = INT(GAP/2)
    WEND
END SUB
```

■ Syntax

SYSTEM

■ Action

Closes all open files and returns control to the operating system

■ Remarks

When a **SYSTEM** command is executed, all files are closed, and BASIC exits to the operating system (for standalone executable programs), or displays the following message (if the program is compiled in memory):

```
Normal termination. Press any key.
```

■ Compiler/Interpreter Differences

END and **SYSTEM** have identical effects in the compiler.

TAB Function

■ Syntax

TAB(*column*)

■ Action

Moves the print position to *column*

■ Remarks

If the current print position is already beyond *column*, **TAB** goes to that position on the next line. Space 1 is the left-most position, and the right-most position is the current line width of the output device, minus one. If the argument *column* is greater than this output width, then **TAB**(*column*) moves the print position to *column* **MOD** *width*. If *column* is less than 1, then **TAB**(*column*) moves the print position to column 1.

TAB can be used only in **PRINT** and **LPRINT** statements.

■ Example 1

```
FOR I = 1 TO 4
    READ A$,B$
    PRINT A$ TAB(25) B$
NEXT
DATA NAME, AMOUNT,,, G.T. JONES, $25.00, H.L. STEVENS, $32.25
```

Output:

```
NAME                      AMOUNT

G.T. JONES                $25.00
H.L. STEVENS              $32.25
```

■ Example 2

```
'Assumes 80-column screen width:
PRINT TAB(1287) "one"
PRINT TAB(255) "two"
PRINT TAB(-5) "three"
PRINT "12345678901234567890123456789O" TAB(20) "four"
```

Output:

```
      one
               two
three
12345678901234567890123456789O
                   four
```

TAN Function

■ Syntax

TAN(x)

■ Action

Returns the tangent of x, where x is in radians

■ Remarks

TAN is calculated with single-precision accuracy, unless the argument x is a double-precision value, in which case **TAN** is calculated with double-precision accuracy.

■ Compiler/Interpreter Differences

In the interpreter, if **TAN** overflows, the "Overflow" error message is displayed, machine infinity with the appropriate sign is supplied as the result, and execution continues.

In the compiler, if **TAN** overflows, machine infinity is not displayed, and execution does not continue (unless the program has an error-handling routine and has been compiled with the On Error or On Event options).

■ Example

The following example computes the height of an object, once the distance from the viewer to the object and the angle of elevation are known. The program then draws the triangle produced by the base and the computed height.

```
SCREEN 2
INPUT "LENGTH OF BASE: ",BASELEN
INPUT "ANGLE OF ELEVATION (DEGREES,MINUTES): ",DEG,MIN
ANG = (3.141593/180)*(DEG + MIN/60)      'Convert to radians
HEIGHT = BASELEN*TAN(ANG)                'Calculate height
PRINT "HEIGHT =" HEIGHT
H = 180 - HEIGHT
B = 15 + BASELEN
LINE (15,180)-(B,180)                    'Draw triangle
LINE -(B,H)
LINE -(10,180)
```

■ Syntax

TIME$

■ Action

Retrieves the current time

■ Remarks

The **TIME$** function returns an eight-character string in the form *hh:mm:ss*, where *hh* is the hour (00–23), *mm* is minutes (00–59), and *ss* is seconds (00–59). A 24-hour clock is used; therefore, 8:00 PM is shown as 20:00:00.

To set the time, use the **TIME$** statement.

■ See Also

TIME$ (Statement), **TIMER**

■ Example

```
'Convert the 24-hour clock used by TIME$ to 12-hour output
'followed by "AM" or "PM":
T$ = TIME$
HR = VAL(T$)
IF HR < 12 THEN AMPM$ = " AM" ELSE AMPM$ = " PM"
IF HR > 12 THEN HR = HR - 12
PRINT "The time is" STR$(HR) RIGHT$(T$,6) AMPM$
```

Sample output:

```
The time is 11:26:31 AM
```

TIME$ Statement

■ Syntax

TIME$=*string-expression*

■ Action

Sets the time

■ Remarks

The *string-expression* must be in one of the following forms:

Form	Description
hh	Sets the hour; minutes and seconds default to 00
hh:mm	Sets the hour and minutes; seconds default to 00
hh:mm:ss	Sets the hour, minutes, and seconds

A 24-hour clock is used; 8:00 p.m., therefore, should be entered as 20:00:00.

This statement complements the **TIME$** function, which retrieves the time.

■ See Also

TIME$ (Function)

■ Example

The current time is set at 8:00 a.m:

```
TIME$="08:00:00"
```

■ Syntax

TIMER

■ Action

Returns the number of seconds elapsed since midnight

■ Remarks

The timer function can be used with the **RANDOMIZE** statement to generate a random number. It can also be used to time programs or parts of programs.

■ Example

The following program searches for the prime numbers from 3 to 10,000. The **TIMER** function is used to time the execution speed of the program.

```
DEFINT A-Z
DIM MARK(10000)
START! = TIMER
NUM = 0
FOR N = 3 TO 10000 STEP 2
    IF NOT MARK(N) THEN
        'PRINT N,    'To print the primes, remove the remark
                     'delimiter in front of the PRINT statement
        DELTA = 2*N
        FOR I = 3*N TO 10000 STEP DELTA
            MARK(I) = -1
        NEXT
        NUM = NUM + 1
    END IF
NEXT
FINISH! = TIMER
PRINT : PRINT "Program took" FINISH!-START!;
PRINT "seconds to find the" NUM "primes"
END
```

Output:

```
Program took 2.632813 seconds to find the 1228 primes
```

TIMER ON, TIMER OFF, TIMER STOP Statements

■ Syntax

TIMER ON
TIMER OFF
TIMER STOP

■ Action

TIMER ON enables event trapping during real time, **TIMER OFF** disables event trapping during real time, and **TIMER STOP** suspends real-time event trapping.

■ Remarks

The **TIMER ON** statement enables real-time event trapping by an **ON TIMER** statement. While trapping is enabled with the **ON TIMER** statement, a check is made after every statement to see if to see if the specified time has elapsed. If it has, the **ON TIMER** statement is executed.

TIMER OFF disables the event trap. If an event takes place, it is not remembered if a subsequent **TIMER ON** is used.

TIMER STOP disables the event trap, but if an event occurs, it is remembered and an **ON TIMER** statement will be executed as soon as trapping is enabled.

■ See Also

ON TIMER

■ Example

This example displays the time of day on line 1, and updates the display once a minute:

```
TIMER ON
ON TIMER(60) GOSUB DISPLAY
.
.
.
END

DISPLAY:
    OLDROW = CSRLIN              'Save current row
    OLDCOL = POS(0)              'Save current column
    LOCATE 1,1 : PRINT TIME$;
    LOCATE OLDROW,OLDCOL         'Restore row & column
RETURN
```

■ Syntax

TRON
TROFF

■ Action

Traces the execution of program statements

■ Remarks

As an aid in debugging standalone executable files, the **TRON** statement enables a trace flag that prints each line number of a program as it is executed. The line numbers appear enclosed in square brackets. The trace flag is disabled with the **TROFF** statement.

When used in programs compiled in memory, **TRON** activates the QuickBASIC debugging system. For more information on using **TRON** to debug your programs from within QuickBASIC, see Section 3.9, "Using Debugging Mode," and Section 5.4, "Debugging Mode."

■ Compiler/Interpreter Differences

To use **TRON/TROFF**, the compiler Debug option (**/d**) must be turned on. Otherwise, **TRON** and **TROFF** are ignored, and a warning message is generated. The interpreter does not require a Debug (**/d**) option to use **TRON/TROFF**.

■ Example

The output shown assumes the following program is compiled as a stand-alone executable file.

```
5  TRON
10 K=10
20 FOR J=1 TO 2
30   L=K + 10
40   PRINT J;K;L
50   K=K + 10
60 NEXT J
70 END
```

Output:

```
[10][20][30][40] 1   10   20
[50][60][30][40] 2   20   30
[50][60][70]
```

UBOUND Function

■ Syntax

UBOUND(*array*[,*dimension*])

■ Action

Returns the upper bound (largest available subscript) for the indicated dimension of an array

■ Remarks

The argument *dimension* is an integer from 1 to the number of dimensions in *array*.

In the example array `ACCOUNT(A,B,C,D)`, `A` is dimension 1, `B` is dimension 2, `C` is dimension 3, and `D` is dimension 4. So the function

```
UBOUND(ACCOUNT,1)
```

finds the largest subscript in dimension `A`, the function

```
UBOUND(ACCOUNT,2)
```

finds the largest subscript in dimension `B`, and so on.

You can use the shortened syntax **UBOUND**(*array*) for one-dimensional arrays, since the default value for *dimension* is 1.

Use the **LBOUND** function to find the lower limit of an array dimension.

■ See Also

LBOUND

■ Example

LBOUND and **UBOUND** can be used together to determine the size of an array passed to a subprogram, as in the following program fragment:

```
CALL PRNTMAT(ARRAY())
.
.
.
SUB PRNTMAT(A(2)) STATIC
    FOR I% = LBOUND(A,1) TO UBOUND(A,1)
        FOR J% = LBOUND(A,2) TO UBOUND(A,2)
           PRINT A(I%,J%);" ";
        NEXT J%
    PRINT:PRINT
    NEXT I%
END SUB
```

UNLOCK Statement

■ Syntax

UNLOCK [[#]] *filenumber* [[,{ *record* | [[*start*]] **TO** *end*}]]

■ Action

Releases access restrictions applied to specified portions of a file

■ Remarks

The **UNLOCK** statement is used only after a **LOCK** statement. See the **LOCK** statement reference pages for examples and a complete discussion.

■ See Also

LOCK

■ Syntax

VAL(*string-expression*)

■ Action

Returns the numeric value of string *string-expression*

■ Remarks

The *string-expression* is a sequence of characters that can be interpreted as a numeric value; **VAL** stops reading the string at the first character that it cannot recognize as part of a number. The **VAL** function also strips leading blanks, tabs, and line feeds from the argument string. For example,

```
VAL("     -33/LP")
```

returns the value –33.

■ See Also

STR$

■ Example

The following program prints the names and addresses of people with specific telephone area codes:

```
INPUT "Search for which area: ", TARGETAREA
OPEN "MAIL.DAT" FOR INPUT AS #1
WHILE NOT EOF(1)
    INPUT #1, NM$, PHONENUM$, STREET$, CITYSTATE$
    'VAL will read everything up to the first non-numeric
    'character ("-" in this case):
    AREA = VAL(PHONENUM$)
    IF AREA = TARGETAREA THEN
        PRINT : PRINT NM$
        PRINT STREET$
        PRINT CITYSTATE$
    END IF
WEND
CLOSE : END
```

VARPTR Function

■ Syntax

VARPTR(*variablename*)
VARPTR(*# filenumber*)

■ Action

VARPTR(*variablename*) returns the offset of the variable identified by *variablename* to the current data segment of memory; the form **VARPTR**(*# filenumber*) returns the starting address of the I/O buffer associated with *filenumber*.

■ Remarks

The *variablename* must have been defined prior to the execution of the **VARPTR** function. Otherwise, an "Illegal function call" error results. Variables are defined by executing any reference to the variable.

Any type variable name may be used (numeric, string, array). For string variables, the address of the first byte of the string descriptor is returned. The address returned is a single-precision value in the range 0-1,048,575 (2^{20}-1).

VARPTR is usually used to obtain the address of a variable or array so that it may be passed to an assembly-language subroutine. A function call of the form

```
VARPTR(A(0))
```

is usually specified when passing an array, so that the lowest-addressed element of the array A in the above case, is returned.

Note

Dynamic array data may move when arrays are implicitly or explicitly dimensioned, or when memory is allocated or compacted by any of a large number of statements and functions in BASIC.

Therefore, **VARPTR** must be executed immediately before each use of its result to guarantee correct results.

For sequential files, **VARPTR**(*filenumber*) returns the starting address of the disk I/O buffer assigned to *filenumber*. For randodm-access files, **VARPTR**(*filenumber*) returns the address of the **FIELD** buffer assigned to *filenumber*.

■ Example

```
DEFINT X
X = 500
XADD = VARPTR(X)          'Get address of X
GOSUB SUB1
POKE XADD,0               'Store 0 in low-order byte
GOSUB SUB1
END

SUB1:       'Calculate and print value stored at address XADD
    XVAL = PEEK(XADD) + 256*PEEK(XADD+1)    'Integers stored
    PRINT XVAL                              'as 2 bytes
RETURN
```

Output:

```
 500
 256
```

VARPTR$ Function

■ Syntax

VARPTR$(*variablename*)

■ Action

Returns a string representation of the memory address of the variable

■ Remarks

The *variablename* is the name of a variable in the program. **VARPTR$** returns a three-byte string with the following values:

byte 0 = type

byte 1 = low byte of address

byte 2 = high byte of address

Note, however, that the individual parts of the string are not considered characters.

If *variablename* is an array element, then the array must have been dimensioned prior to the execution of **VARPTR$**. The array must not be a large numeric array.

Note

Dynamic array data may move when arrays are implicitly or explicitly dimensioned, or when memory is allocated or compacted by any of a large number of statements and functions in BASIC.

Therefore, **VARPTR$** must be executed immediately before each use of its result to guarantee correct results.

■ Compiler/Interpreter Differences

In compiled programs, **VARPTR$** must be used in the **DRAW** and **PLAY** statements to execute substrings containing variables.

The interpreter supports both the **VARPTR$** syntax and the syntax containing just the variable name.

■ See Also

DRAW, PLAY

■ Example

This uses the **PLAY** subcommand `X` (execute), plus the contents of `A$`, as the string expression in the `PLAY` statement:

```
PLAY "X"+VARPTR$(A$)
```

■ Syntax

VIEW [[**SCREEN**]] [[(*x1*,*y1*)-(*x2*,*y2*) [,[[*color*]] [,*border*]]]]

■ Action

Defines screen limits for graphics output

■ Remarks

VIEW defines a "physical viewport," or a rectangular section of the screen into which graphics can be mapped. The coordinates *x1*,*y1* and *x2*,*y2* define diagonally opposite corners of this rectangle. All coordinates must be within the physical bounds of the screen.

If **VIEW** is given with no arguments, the entire screen is defined as the viewport. **RUN** and **SCREEN** also define the entire screen as the viewport, thus disabling any viewports defined by **VIEW**.

The *color* attribute allows the user to fill the view area with a color. If *color* is omitted, the view area is not filled.

The *border* attribute allows the user to draw a line surrounding the viewport if space for a border is available. If *border* is omitted, no border is drawn.

The **SCREEN** option dictates that the *x* and *y* coordinates are absolute to the screen, not relative to the border of the physical viewport, and only graphics within the viewport will be plotted. If **SCREEN** is not given, all points plotted are relative to the viewport (that is, *x1* and *y1* are added to the *x* and *y* coordinates before putting the point down on the screen).

■ Example 1

The following example contrasts the output after VIEW and VIEW SCREEN are executed:

```
SCREEN 2
'SCREEN option makes all screen coordinates absolute:
VIEW SCREEN (10,10)-(200,100)
'This line will not be visible, since its end
'points are outside the viewport:
LINE (5,5)-(100,5)
PRINT "Press any key to continue."
SUSPEND$ = INPUT$(1)     'Wait for a key to be pressed.
'No SCREEN option - all screen coordinates relative
'to this viewport:
VIEW (10,10)-(200,100)
'This line is now visible, since its end points
'are added to (10,10):
LINE (5,5)-(100,5)
```

■ Example 2

A number of **VIEW** statements can be executed. If the newly described viewport is not wholly within the previous viewport, the screen can be re-initialized with the **VIEW** statement. Then the new viewport can be stated. If the new viewport is entirely within the previous one, as in the following example, the intermediate **VIEW** statement is not necessary. This example opens three viewports, each smaller than the previous one. In each case, the points of the line that lie outside the viewport borders are clipped and do not appear on the screen.

```
SCREEN 2
CLS
VIEW     'Make the viewport the entire screen.
VIEW (10,10) - (300,180),,1
   CLS
   LINE (0,0) - (310,190),1
   LOCATE 1,11: PRINT "A big viewport"
VIEW SCREEN (50,50)-(250,150),,1
   CLS  'Note CLS clears only viewport
   LINE (300,0)-(0,199),1
   LOCATE 9,9: PRINT "A medium viewport"
VIEW SCREEN (80,80)-(200,125),,1
   CLS
   CIRCLE (150,100),20,1
   LOCATE 11,9: PRINT "A small viewport"
```

VIEW PRINT Statement

■ Syntax

VIEW PRINT [[*topline* **TO** *bottomline*]]

■ Action

Sets the boundaries of the screen text window

■ Remarks

VIEW PRINT without *topline* and *bottomline* parameters initializes the whole screen area as the text window. The whole screen area consists of lines 1 to 24; by default, line 25 is not used.

Statements and functions that operate within the defined text window include **CLS**, **LOCATE**, **PRINT**, and **SCREEN**.

The screen editor will limit functions such as scroll and cursor movement to the text window.

■ See Also

VIEW

■ Syntax

WAIT *portnumber,and-expression*[,*xor-expression*]

■ Action

Suspends program execution while monitoring the status of a machine input port

■ Remarks

The arguments *and-expression* and *xor-expression* are integer expressions.

The **WAIT** statement causes execution to be suspended until a specified machine input port develops a specified bit pattern. The data read at the port is **XOR**ed with the integer expression *xor-expression*, then **AND**ed with *and-expression*. If the result is zero, BASIC loops back and reads the data at the port again. If the result is nonzero, execution continues with the next statement. If *xor-expression* is omitted, it is assumed to be 0.

Warning

It is possible to enter an infinite loop with the **WAIT** statement if the given input port never develops a nonzero bit pattern. In this case it will be necessary to manually restart the machine. To avoid this, **WAIT** must have a nonzero value at *portnumber* during some point in the program execution.

■ Example

```
WAIT 32,2
```

WHILE...WEND Statement

■ Syntax

WHILE *condition*
.
.
.
[[*statements*]]
.
.
.
WEND

■ Action

Executes a series of statements in a loop as long as a given condition is true

■ Remarks

If the *condition* is true (that is, if it does not equal zero), then any intervening statements are executed until the **WEND** statement is encountered. BASIC then returns to the **WHILE** statement and checks *condition*. If it is still true, the process is repeated. If it is not true (or if it equals zero), execution resumes with the statement following the **WEND** statement.

WHILE...WEND loops may be nested to any level. Each **WEND** will match the most recent **WHILE**. An unmatched **WHILE** statement causes a "WHILE without WEND" error message to be generated, and an unmatched **WEND** statement causes a "WEND without WHILE" error message to be generated.

Warning

Do not direct program flow into a **WHILE...WEND** loop without entering through the **WHILE** statement, as this will confuse structuring of control flow.

■ Example

The following fragment performs a bubble sort on the array A$. The second line makes the variable FLIPS true by assigning it a non-zero value; this forces one pass through the **WHILE...WEND** loop. When there are finally no more swaps, then all the elements of A$ are sorted, FLIPS is false (that is, equal to zero), and the program continues execution with the line following WEND.

```
'Bubble sort array A$
FLIPS = -1        'Force one pass through loop
WHILE FLIPS
   FLIPS = 0
   MAX = UBOUND(A$)
   FOR I = 2 TO MAX
      IF A$(I-1) > A$(I) THEN
         FLIPS = -1
         SWAP A$(I-1),A$(I)
      END IF
   NEXT
WEND
.
.
.
```

■ Syntax

WIDTH [[{ # *filenumber* | *device*} ,]] *width*
WIDTH LPRINT *width*

■ Action

Assigns an output line width to a file or device

■ Remarks

In the compiler, both files and devices can be assigned an output-line width. The possible syntax variations for this statement are explained in the following list:

Syntax	Action
WIDTH *width*	Sets the width of the terminal screen to *width*. The only allowable values for *width* are 40 and 80.
WIDTH # *filenumber*,*width*	Sets the width of a file opened to an output device (e.g., **LST** or **CON**) to *width*. The *filenumber* is the number associated with the file in the **OPEN** statement. This form permits altering the width while a file is open, since the statement takes place immediately.
WIDTH *device*,*width*	Sets the width of *device* (a device file name) to *width*. The *device* should be a string expression: for example, `"CONS:"`. Note that this width assignment is *deferred* until the next **OPEN** statement affecting the device; the assignment does not affect output for an already open file.

WIDTH LPRINT *width* — Sets the width of the line printer for use by subsequent **LPRINT** statements.

■ Example 1

In the following example, the record width in file #1 (the lineprinter) is set to different widths:

```
OPEN "LPT1:" FOR OUTPUT AS #1
TEST$ = "1234567890"
WIDTH #1, 3
PRINT #1, TEST$
WIDTH #1, 4
PRINT #1, TEST$
CLOSE
```

Output on line printer:

```
123
456
789
0
1234
5678
90
```

■ Example 2

In the following example, the width of the console display is set to 35 columns:

```
WIDTH "SCRN:",35
PRINT TAB(10) "DECEMBER 1986"
PRINT
FOR I = 0 TO 6
    READ DAY$
    PRINT TAB(5*I) DAY$;
NEXT
FOR X = 1 TO 31
   PRINT TAB(5*X) X;
NEXT
DATA " Su"," M"," Tu"," W"," Th"," F"," Sa"
```

Output:

```
          DECEMBER 1986

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  31
```

■ Syntax

WINDOW [[**SCREEN**] (*x1*,*y1*)-(*x2*,*y2*)]

■ Action

Defines the logical dimensions of the current viewport

■ Remarks

The **WINDOW** statement allows the user to create a customized mapping system to draw lines, graphs, or objects in space without constraint by the physical dimensions of the screen (or "physical coordinates"). This is done by redefining the screen-border coordinates with the "world coordinates" (*x1*, *y1*) and (*x2*, *y2*). These world coordinates are single-precision, floating-point numbers.

WINDOW defines the section of the floating-point world (or window) to be mapped to the physical space of the viewing surface (screen). All subsequent graphics statements will be defined with these new world coordinates and will be displayed within the current viewport. (The size of the viewport can be changed with the **VIEW** statement.)

RUN, or **WINDOW** with no arguments, disables the "window" transformation.

The **WINDOW SCREEN** variant inverts the normal Cartesian direction of the *y* coordinate, so *y* values go from negative to positive from top to bottom.

Figure 11.1 shows the standard screen coordinates after a **SCREEN 2** statement. Note that the coordinates of the pixel in the upper-left corner are (0,0); also, *x*-coordinate values increase from left to right and *y*-coordinate values increase from top to bottom.

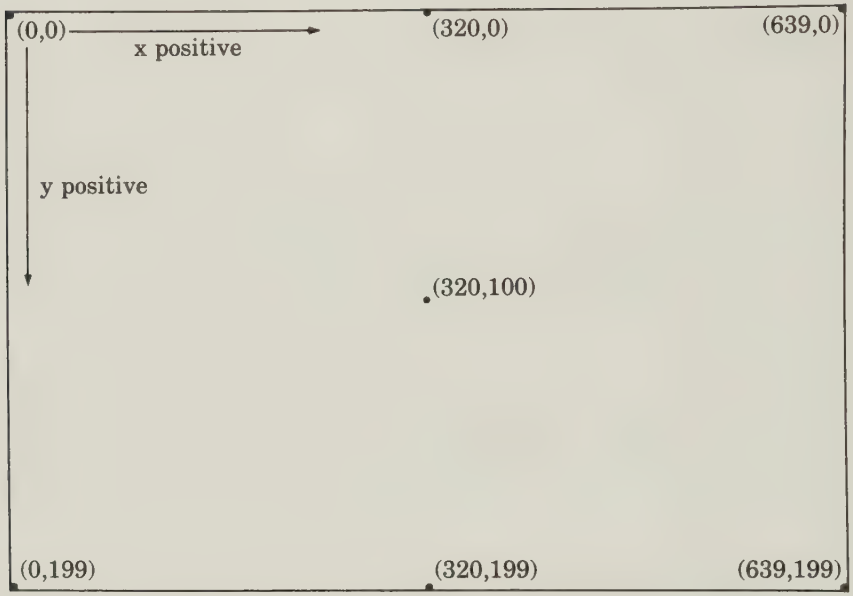


Figure 11.1 Default Pixel Coordinates

Figure 11.2 shows the change in screen coordinates after a **WINDOW** statement redefines the entire screen. Pixel coordinates are now defined in terms of this window. Note that the coordinates of the pixel in the center of the screen are now (0,0). In addition, *x*-coordinate and *y*-coordinate values behave as they do in a traditional Cartesian grid: "up" and "right" are positive; "down" and "left" are negative.

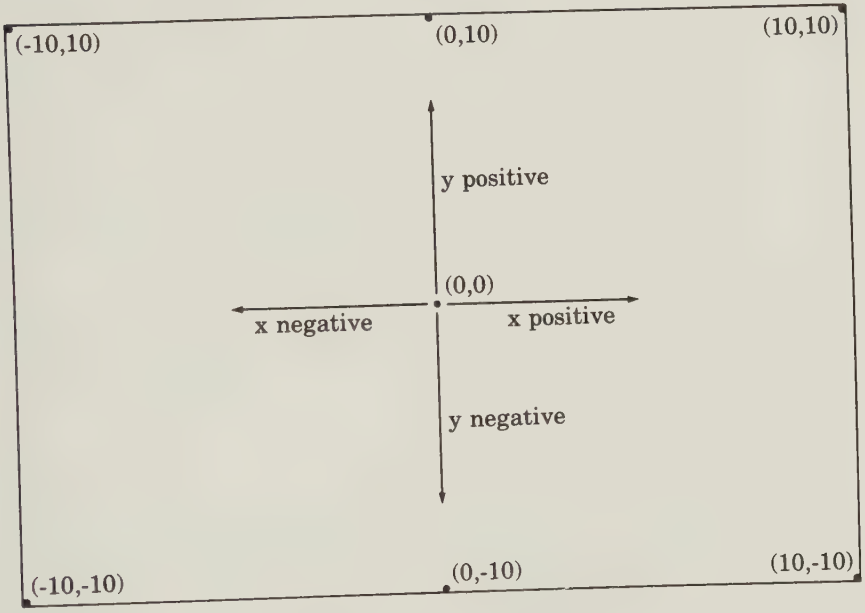


Figure 11.2 Coordinates After WINDOW (–10,–10)-(10,10)

Figure 11.3 should be contrasted with figure 11.2 to see how the **SCREEN** option changes the behavior of the window. With **SCREEN**, "up" is now negative, while "down" is now positive.

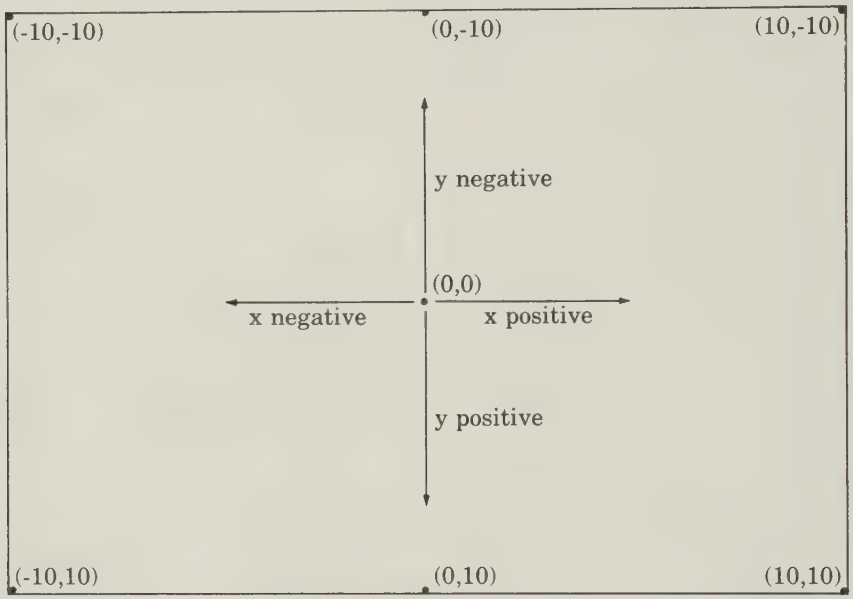


Figure 11.3 Coordinates After WINDOW SCREEN (–10,–10)-(10,10)

Example 1

The following example illustrates two lines with the same end-point coordinates. The first is drawn on the default screen, and the second is on a redefined window.

```
SCREEN 2
LINE (100,100) - (150,150), 1
LOCATE 2,20 : PRINT "The line on the default screen"
WINDOW SCREEN (100,100) - (200,200)
LINE (100,100) - (150,150), 1
LOCATE 8,18 : PRINT"& the same line on a redefined window"
```

Example 2

The following example shows how changing the window size changes the size of a figure drawn on the screen. The effect is one of "zooming" in and out: as the window gets smaller, the figure appears larger on the screen, until parts of it are finally clipped because they lie outside the window; as the window gets larger, the figure appears smaller on the screen.

```
PRINT "Press <RETURN> to start; press any other key to end";
INPUT;"",A$
SCREEN 1 : COLOR 7                  'Grey screen
X = 500 : XDELTA = 50
LOOP:
    WHILE X < 525 AND X > 50
        X = X + XDELTA              'Change window size
        GOSUB ZOOM
        FOR I = 1 TO 1000           'Delay loop
            A$ = INKEY$             'Stop if key pressed
            IF A$ <> "" THEN END
        NEXT
    WEND
    X = X - XDELTA
    XDELTA = -XDELTA                'Reverse size change
    GOTO LOOP

ZOOM:
    CLS
    WINDOW (-X,-X)-(X,X)            'Define new window
    LINE (-X,-X)-(X,X),1,B          'Draw window border
    CIRCLE (0,0),60,1,,,.5          'Draw ellipse with x-radius 60
    PAINT (0,0),1                   'Paint ellipse
RETURN
```

■ Syntax

WRITE [[*expressionlist*]]

■ Action

Outputs data to the screen

■ Remarks

If *expressionlist* is omitted, a blank line is output. If *expressionlist* is included, the values of the expressions are output to the screen. The expressions in the list can be numeric and/or string expressions. They must be separated by commas.

When the printed items are output, each item is separated from the last by a comma. Printed strings are delimited by quotation marks. After the last item in the list is printed, BASIC inserts a carriage-return–line-feed.

WRITE outputs numeric values using the same format as the **PRINT** statement.

■ See Also

PRINT

■ Example

This example shows the difference between the **PRINT** and **WRITE** statements:

```
A=80 : B=90 : C$="THAT'S ALL"
WRITE A,B,C$
PRINT A,B,C$
```

Output:

```
80, 90,"THAT'S ALL"
 80       90       THAT'S ALL
```

■ Syntax

WRITE# *filenumber,expressionlist*

■ Action

Writes data to a sequential file

■ Remarks

The *filenumber* is the number under which the file was opened for **OUTPUT** or **APPEND** in the **OPEN** statement. The expressions in the *expressionlist* are string or numeric expressions. They must be separated by commas.

The difference between **WRITE#** and **PRINT#** is that **WRITE#** inserts commas between the items as they are written to the file and delimits strings with quotation marks. Therefore, it is not necessary for you to put explicit delimiters in the list. A new line is inserted once the last item in the list has been written to the file.

If a **WRITE#** statement attempts to write data to a sequential file to which access has been restricted by a **LOCK** statement, two options are available. The first is to return control to the program immediately with an accompanying error message. All of BASIC's usual error-handling routines can trap and examine this error. If error trapping is not active the error message is:

```
Permission denied
```

■ See Also

OPEN, LOCK, PRINT#, WRITE

■ Example

The output from the following program illustrates the difference between the **WRITE#** and **PRINT#** statements:

```
A$ = "VCR, remote control" : B$ = "$399.00"
OPEN  "PRICES" FOR OUTPUT AS #1        'Open PRICES for writing
PRINT #1,A$,B$        'Store A$ and B$ in first record with PRINT#
WRITE #1,A$,B$        'Store A$ and B$ in second record with WRITE#
CLOSE #1
OPEN "PRICES" FOR INPUT AS #1          'Open PRICES for reading
INPUT #1, X$, Y$                       'Read the first record
PRINT "PRINT#" TAB(10) X$ TAB(40) Y$   'Print the first record
INPUT #1, X$, Y$                       'Read the second record
PRINT "WRITE#" TAB(10) X$ TAB(40) Y$   'Print the second record
CLOSE #1
```

Output:

```
PRINT#  VCR                    remote control          $399.00
WRITE#  VCR, remote control    $399.00
```

Part 3
Appendixes

A ASCII Character Codes 523

B QuickBASIC Reserved Words 525

C Differences between QuickBASIC Version 1.0 and Version 2.0 527

D Using the Separate Compilation Method 533

E Error Messages 549

Appendix A
ASCII Character Codes

Dec	Oct	Hex	Chr	Dec	Oct	Hex	Chr
000	000	00H	NUL	032	040	20H	SP
001	001	01H	SOH	033	041	21H	!
002	002	02H	STX	034	042	22H	"
003	003	03H	ETX	035	043	23H	#
004	004	04H	EOT	036	044	24H	$
005	005	05H	ENQ	037	045	25H	%
006	006	06H	ACK	038	046	26H	&
007	007	07H	BEL	039	047	27H	'
008	010	08H	BS	040	050	28H	(
009	011	09H	HT	041	051	29H	)
010	012	0AH	LF	042	052	2AH	*
011	013	0BH	VT	043	053	2BH	+
012	014	0CH	FF	044	054	2CH	,
013	015	0DH	CR	045	055	2DH	-
014	016	0EH	SO	046	056	2EH	.
015	017	0FH	SI	047	057	2FH	/
016	020	10H	DLE	048	060	30H	0
017	021	11H	DC1	049	061	31H	1
018	022	12H	DC2	050	062	32H	2
019	023	13H	DC3	051	063	33H	3
020	024	14H	DC4	052	064	34H	4
021	025	15H	NAK	053	065	35H	5
022	026	16H	SYN	054	066	36H	6
023	027	17H	ETB	055	067	37H	7
024	030	18H	CAN	056	070	38H	8
025	031	19H	EM	057	071	39H	9
026	032	1AH	SUB	058	072	3AH	:
027	033	1BH	ESC	059	073	3BH	;
028	034	1CH	FS	060	074	3CH	<
029	035	1DH	GS	061	075	3DH	=
030	036	1EH	RS	062	076	3EH	>
031	037	1FH	US	063	077	3FH	?

Dec=Decimal, Oct=Octal, Hex=Hexadecimal(H), Chr=Character, LF=Line feed
FF=Form feed, CR=Carriage return, DEL=Delete

Appendix A *(continued)*

Dec	Oct	Hex	Chr	Dec	Oct	Hex	Chr
064	100	40H	@	096	140	60H	`
065	101	41H	A	097	141	61H	a
066	102	42H	B	098	142	62H	b
067	103	43H	C	099	143	63H	c
068	104	44H	D	100	144	64H	d
069	105	45H	E	101	145	65H	e
070	106	46H	F	102	146	66H	f
071	107	47H	G	103	147	67H	g
072	110	48H	H	104	150	68H	h
073	111	49H	I	105	151	69H	i
074	112	4AH	J	106	152	6AH	j
075	113	4BH	K	107	153	6BH	k
076	114	4CH	L	108	154	6CH	l
077	115	4DH	M	109	155	6DH	m
078	116	4EH	N	110	156	6EH	n
079	117	4FH	O	111	157	6FH	o
080	120	50H	P	112	160	70H	p
081	121	51H	Q	113	161	71H	q
082	122	52H	R	114	162	72H	r
083	123	53H	S	115	163	73H	s
084	124	54H	T	116	164	74H	t
085	125	55H	U	117	165	75H	u
086	126	56H	V	118	166	76H	v
087	127	57H	W	119	167	77H	w
088	130	58H	X	120	170	78H	x
089	131	59H	Y	121	171	79H	y
090	132	5AH	Z	122	172	7AH	z
091	133	5BH	[	123	173	7BH	{
092	134	5CH	\	124	174	7CH	|
093	135	5DH	]	125	175	7DH	}
094	136	5EH	^	126	176	7EH	~
095	137	5FH	_	127	177	7FH	DEL

Dec=Decimal, Oct=Octal, Hex=Hexadecimal(H), Chr=Character, LF=Line feed
FF=Form feed, CR=Carriage return, DEL=Delete

Appendix B
QuickBASIC Reserved Words

The following is a list of Microsoft QuickBASIC reserved words:

ABS
ACCESS
AND
APPEND
AS
ASC
ATN
AUTO

BASE
BEEP
BLOAD
BSAVE

CALL
CALLS
CDBL
CHAIN
CHDIR
CHR$
CINT
CIRCLE
CLEAR
CLOSE
CLS
COLOR
COM
COMMAND$
COMMON
CONT
COS
CSNG
CSRLIN
CVD
CVI
CVS

DATA
DATE$
DEF FN
DEF SEG
DEF USR
DEFDBL
DEFINT
DEFSNG
DEFSTR
DELETE
DIM
DRAW

EDIT
ELSE
END
ENVIRON
ENVIRON$
EOF
EQV
ERASE
ERDEV
ERDEV$
ERL
ERR
ERROR
EXIT
EXP

FIELD
FILES
FIX
FOR
FRE

GET
GO
GOSUB
GOTO$

HEX

IF
IMP
INKEY$
INP
INPUT
INPUT#
INPUT$
INSTR
INT

KEY
KILL

LBOUND
LCOPY
LEFT$
LEN
LET
LINE
LIST
LLIST
LOAD
LOC
LOCAL
LOCATE
LOCK
LOF
LOG
LPOS

LPRINT
LSET

MERGE
MID$
MKD$
MKDIR
MKI$
MKS$
MOD
MOTOR

NAME
NEW
NEXT
NOISE
NOT
NULL

OCT$
ON
OPEN
OPEN COM
OPTION
OR
OUT
OUTPUT

PAINT
PALETTE
PALETTE USING
PEEK
PEN
PLAY
PMAP
POINT
POKE
POS
PRESET
PRINT
PRINT# USING
PSET
PUT

RANDOMIZE
READ
REDIM
REM
RENUM
RESET
RESTORE
RESUME
RETURN
RIGHT$
RMDIR
RND
RSET
RUN

SADD
SAVE
SCREEN
SEG
SGN
SHARED
SHELL
SIN
SOUND
SPACE
SPACE$
SPC
SQR
STATIC
STEP
STICK
STOP
STR$
STRIG$
STRING
SUB
SWAP
SYSTEM

TAB
TAN
THEN
TIME$
TIMER
TO
TROFF
TRON

UBOUND
UNLOCK
USING
USR
USR0
USR1
USR2
USR3
USR4
USR5
USR6
USR7
USR8
USR9

VAL
VARPTR
VARPTR$
VIEW

WAIT
WEND
WHILE
WIDTH
WINDOW
WRITE
WRITE#

XOR

Appendix C

Differences between QuickBASIC Version 1.0 and Version 2.0

C.1 Compatibility between Version 1.0 and Version 2.0 529

C.2 Command Line 529

C.3 Options 529

C.4 IBM Enhanced Graphics Support Adapter 530

C.5 New Language Features 530

C.6 Using Microsoft® Word Files 531

C.1 Compatibility between Version 1.0 and Version 2.0

QuickBASIC Versions 1.0 and 2.0 are source and binary compatible. This means that source files created for Version 1.0 will compile under Version 2.0. Also, object files created with Version 1.0 can be linked with Version 2.0 files, using the linker or the **buildlib** command.

Executable files are not compatible between versions. Files created with one version cannot be chained to **.EXE** files created with the other version.

C.2 Command Line

QuickBASIC Version 2.0 supports the command line available in Version 1.0. You can compile Version 2.0 source files using the command line in the following form:

qb *filename* [[*options*]] **;**

The prompting sequence is not supported in Version 2.0; a command line without a semicolon at the end invokes the QuickBASIC editor. If this happens, choose the Compile command from the Run menu to process the program.

C.3 Options

The Version 1.0 **/A** (source listing) option is not supported in Version 2.0. If you use the **/A** option on the command line the compiler ignores it.

The following options are added to QuickBASIC Version 2.0:

Option	Description
/b	Use this option when you are using a monochrome monitor with a color graphics card, or if you don't want to display the QuickBASIC screen in color.
/g	Updates the screen at maximum speed.
/l [[*libraryfile*]]	Loads a user library file. If *libraryfile* is not present, the default library **USERLIB.EXE** is loaded.

These options are described in Section 4.1, "Starting QuickBASIC."

C.4 IBM Enhanced Graphics Support Adapter

QuickBASIC supports the screen modes made available by the IBM Enhanced Graphics Adapter. These screen modes are described in Part 2, "Reference."

C.5 New Language Features

The following statements and functions, described in Part 2, "Reference," are added to QuickBASIC Version 2.0:

BLOAD
BSAVE
PALETTE
PALETTE USING
PCOPY
SADD

QuickBASIC Version 2.0 also supports multiline **IF...THEN...ELSE** statements, and large numeric arrays (LNA). See the "Reference" for information about **IF...THEN...ELSE**. Large numeric arrays are explained in Chapter 10.

C.6 Using Microsoft® Word Files

Formatted Microsoft Word files cannot be compiled with QuickBASIC Version 2.0. If you want to compile formatted Word files with QuickBASIC, load the file into Word and resave it with the Unformatted option before loading it into QuickBASIC.

Appendix D
Using the Separate Compilation Method

D.1 Disk Setup 535
D.1.1 Two-Drive Setup 536
D.1.2 Hard-Disk Setup 537
D.2 Compiling 538
D.3 Linking 540
D.3.1 Linking with the BRUN20.LIB Library 541
D.3.2 Linking with the BCOM20.LIB Library 542
D.3.3 Linking with GWCOM.OBJ 543
D.3.4 Linking with SMALLERR.OBJ 543
D.4 Running a Program 544
D.5 The Run-Time Memory Maps 546

In addition to in-memory compilation, QuickBASIC lets you compile programs without entering the QuickBASIC editor. Programs compiled outside the editor must be explicitly linked.

The separate compilation method is useful for creating executable programs that use routines from the **BCOM20.LIB** alternate run-time library, and with batch processing.

To compile a program without using the QuickBASIC menus, use the **qb** command to create an object file, then use the **link** command to create an executable program. This method creates an executable file with the base name of the source file and the extension **.EXE**. The program is executed when the user enters the base name of the file.

You can also create executable programs with batch files created for use with QuickBASIC Version 1.0. The batch file does not need modification. However, QuickBASIC Version 2.0 does not support the **/A** listing option, and it is ignored if present.

This appendix describes

- Setting up your file system for use with the separate compilation method
- The **qb** command syntax
- The **link** command syntax
- The library options
- Using **SMALLER.OBJ** and **GWCOM.OBJ**
- Running the program

D.1 Disk Setup

The files listed in Table D.1 are required when using the separate compilation method.

Table D.1

Separate Compilation Method Required Files

File Names	Description
QB.EXE	The Microsoft QuickBASIC Compiler
BRUN20.EXE	The Microsoft QuickBASIC run-time module
BRUN20.LIB	The Microsoft QuickBASIC run-time module library
BCOM20.LIB	The Microsoft QuickBASIC alternate run-time library

Before you can compile and link programs, you must make sure the compiler and linker can locate all the files they need to process your program. The **PATH** environment variable tells DOS where to look for executable files, and the **LIB** environment variable tells DOS where to look for libraries. These variables are defined at the DOS command level using the **SET** command. See your DOS documentation for more information about the **SET** command.

Example

The following commands tell the compiler to search for executable files on Drive A in the directory named **BIN**, then, if necessary, in the directory **SRC**, and then to search for library files in the directory **LIB**:

```
SET PATH=A:\BIN;A:\SRC
SET LIB=A:\LIB
```

D.1.1 Two-Drive Setup

If you are using a computer with two floppy-disk drives, you will want to arrange your software so that one drive contains a disk that has the compiler, libraries, and run-time module on it, and the other drive contains the linker and the program you are working on.

A sample setup is given below:

Disk 1	Disk 2
COMMAND.COM	**LINK.EXE**
QB.EXE	Source program files
BRUN20.EXE	
BCOM20.LIB	
BRUN20.LIB	

Using this sample setup, your environment variables should be given the values shown below, assuming that Disk 1 will be in Drive A and Disk 2 will be in Drive B:

```
SET PATH=A:\
SET LIB=A:\
```

With the above setup, use Drive B as your default drive. Run the compiler from Drive B, so that the output files (the object file, and executable program file) are written onto Disk 2.

D.1.2 Hard-Disk Setup

If you have a computer with a hard disk, you will want to copy the compiler, linker, and library files onto the hard disk. It is recommended that you place the executable, library, and source files in separate directories and set your environment variables accordingly.

The following is a sample hard-disk setup:

\BIN\QB.EXE
\BIN\BRUN20.EXE
\BIN\LINK.EXE
\LIB\BRUN20.LIB
\LIB\BCOM20.LIB

Using the above setup, your environment variables can be given the values shown below:

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

D.2 Compiling

With the separate compilation method, the **qb** command has the following syntax:

qb *filename* [[*optionlist*]] **;**

Argument	Description
filename	Is the name of the file being compiled.
optionlist	Is one or more of the following options:

Option	Description
/c:*buffersize*	For each communications port, **/c** sets the size of the buffer receiving remote data using an asynchronous communications adapter. (The transmission buffer is allocated 128 bytes for each communications port and cannot be changed on the **qb** command line.) This option has no effect if the asynchronous communications card is not present. The default buffer size is 256 bytes total for both ports; the maximum size is 32,767 bytes. Note that if you are using the asynchronous communications adapter with programs compiled with the **/o** option, you must also link with **GWCOM.OBJ**, which contains the communications support routines.
/d	Generates debugging code for run-time error checking and enables CTRL-BREAK. This option is the same as the Debug option.
/e	Indicates the presence of **ON ERROR GOTO** with **RESUME** *linenumber* statements. This option is the same as the On Error option.

/l [*libraryfile*]	Use this option when the program contains calls to the user library. When no *libraryfile* is present, the default **USERLIB.EXE** is used. If /l is omitted, calls to the user library cause "Unresolved external reference" errors at link time. The user library must be in the search path when the program is compiled and run.
/o	Substitutes the **BCOM20.LIB** run-time library for **BRUN20.LIB**. See Sections D.3.1 and D.3.2 for more information about these libraries.
/q	Optimizes the program to be as fast as possible. If the option is not present, the program is optimized for size by default.
/r	Stores arrays in row order. This option is the same as Arrays In Row Order option.
/s	Writes quoted strings to the **.OBJ** file instead of the symbol table. Use this option when an "Out of memory" error occurs in a program that has many string constants. This option is the same as Minimize String Data option.
/v	Enables event trapping for communications (**COM**), lightpen (**PEN**), joystick (**STRIG**), timer (**TIMER**), and function keys (**KEY**). Use this option to check between statements for an occurrence of an event. This option is the same as the Checking Between Statements option.

/w	Enables event trapping for same statements as **/v**, but checks between lines for occurrence of an event. This option is the same as the Event Trapping option.
/x	Indicates presence of **ON ERROR GOTO** with **RESUME**, **RESUME NEXT**, or **RESUME 0**. This option is the same as the Resume Next option.

The options are explained in detail in Chapter 4, "Compiling and Running a Program."

Important

The semicolon (;) terminating the command line is required. If the semicolon is not present, the screen comes up in the QuickBASIC editor. Compilation is not performed. If this happens, choose Quit from the File menu and try again.

Example

The following command line compiles the file **SAMPLE.BAS** into **SAMPLE.OBJ**:

```
qb sample /v /d /o ;
```

D.3 Linking

The object file created by the compiler must be explicitly linked. Linking brings the compiled program together with the additional resources needed to run the program. The object file produced by the compiler contains calls to routines in libraries. The routines in these libraries are needed for the program to perform functions such as printing characters on the screen and accepting input from the keyboard.

The linker program, **LINK.EXE**, takes the **.OBJ** file produced by the compiler and links the appropriate routines from the library you specify. The result of this process is the executable (**.EXE**) file. The following sections explain how to link a program, and describe the two libraries available with the Microsoft QuickBASIC Compiler.

The **link** command has the form:

link *objectfiles*[,[*executable*] [,[*map*] [,[*library*]]]] [*option...*] [;]

It takes the following arguments:

Argument	Description
object	Is a list of object modules, separated by plus signs or blanks.
executable	Is the name of the file receiving the executable output. The default is the base name of the object file with the extension **.EXE**.
map	Is the name of the file to receive the map listing. The default is no map file.
library	Consists of the names of the libraries to be searched, separated by plus signs. The default is **BRUN20.LIB** or **BCOM20.LIB**, depending on the use of the **/o** option at compile time.
option...	May be any **link** command option except the **/DSALLOCATE** and **/HIGH** options. The default uses no options.

You can select the default for any argument except *object* by omitting the file name before the comma. You can select default responses by using the semicolon. The semicolon tells the linker to use the default items for all remaining arguments.

D.3.1 Linking with the BRUN20.LIB Library

Programs linked to **BRUN20.LIB** require the presence of the run-time module **BRUN20.EXE** at run time. The library **BRUN20.LIB** is the default library; no options are required at link time. Linking with the **BRUN20.LIB** run-time library provides the following advantages:

- Link time is shorter.
- **COMMON** and **CHAIN** statements can be used to support a system of programs with shared data. This can be valuable in business applications, where programs are often set up this way. With the alternate library **BCOM20.LIB**, **COMMON** is not supported; **CHAIN** is equivalent to **RUN**.
- The **BRUN20.EXE** run-time module resides in memory, so it does not need to be reloaded for each program in a system of chained programs.
- Since **BRUN20.LIB** contains only the code for reading in the **BRUN20.EXE** module, the actual routines from **BRUN20.EXE** are not incorporated into the **.EXE** file. Therefore, the **.EXE** file is much smaller. If several compiled programs are kept on the disk, considerable space is saved.
- Your compiler-generated code, using **BRUN20.LIB**, can be as much as 15 to 20 percent smaller than the code generated with **BCOM20.LIB**. This is because the run-time system uses software interrupts to invoke run-time routines. The **BCOM20.LIB** uses intersegment calls. (Note, however, that optimizing for speed will cancel this effect. Use the **/q** option to generate the fastest code possible.)

D.3.2 Linking with the BCOM20.LIB Library

When you link with **BCOM20.LIB**, selected routines are linked to the **.OBJ** file to create a single **.EXE** file that does *not* require the presence of the run-time module **BRUN20.EXE**. These programs are smaller and slightly faster, but require more disk space than those linked with **BRUN20.LIB**. Also, in programs linked with **BCOM20.LIB**, the **CHAIN** statement is identical to the **RUN** statement; i.e., **COMMON** variables and open files are not preserved.

To link to the **BCOM20.LIB** library, the **/o** option must be specified at compile time. When you specify **/o**, the alternate run-time library (**BCOM20.LIB**) is substituted for **BRUN20.LIB** as the default library to be searched at link time.

Linking with **BCOM20.LIB** provides the following advantages:

- You may save RAM space at run time by linking with **BCOM20.LIB**. This is most likely when you have small, simple programs that do not require all the routines in the run-time module.
- With **BCOM20.LIB**, execution of a compiled and linked **.EXE** file does not require that the run-time module be on the disk at run time. This feature works to your advantage when you write programs for unsophisticated users who may copy the program to another disk but may not copy the run-time module needed to execute the program.
- With **BCOM20.LIB**, programs execute slightly faster because the run-time routines are invoked through 8086 intersegment calls. With **BRUN20.LIB**, the run-time routines are invoked through software interrupts.

D.3.3 Linking with GWCOM.OBJ

The object file **GWCOM.OBJ**, provided on QuickBASIC product Disk 2, contains the run-time support routines required for remote communications. Note that linking with **GWCOM.OBJ** is required only when the program is compiled with the **/o** option.

D.3.4 Linking with SMALLERR.OBJ

The object file **SMALLERR.OBJ** maps all run-time error messages to one of the following:

```
Unprintable error
Internal error
Error occurred
```

Linking with **SMALLERR.OBJ** produces a smaller **.EXE** file. This option is useful only with programs compiled with the **/o** option.

D.4 Running a Program

To run a compiled program, enter the file name without its **.EXE** extension. For example, to run the program **SAMPLE.EXE**, type

```
sample
```

and press ENTER. This command executes the program **SAMPLE.EXE**. If you have linked the program **SAMPLE.EXE** to **BRUN20.LIB**, then the computer loads the **BRUN20.EXE** run-time module. If you compiled the program with a user library, it must also be present.

The computer searches for the user library in the following order:

1. The current directory
2. The directories specified by the **LIB** environment variable

If the computer can't find the user library, the following message appears:

```
Cannot find user library. Enter new path and file name:
```

Enter the user library path specification.

The computer searches for the run-time module in the following order:

1. The current directory
2. The directories specified by the **PATH** environment variable

If the computer can't find the run-time module in the above locations, the following message appears:

```
Cannot find BRUN20.EXE. Enter new path:
```

At this point, enter the new path specification or a drive letter followed by a colon and press ENTER. A check is performed to ensure that the run-time module loaded is compatible with the version of the compiler you are using. Once the run-time module is loaded, execution of the file named **SAMPLE.EXE** begins.

You can also run an executable binary file from within a program. For example

```
RUN "PROG"
```

or

```
CHAIN "PROG"
```

Note that if you use **CHAIN**, the program must be another BASIC program. When **RUN** or **CHAIN** is used, the computer loads an executable binary file. The computer reloads the run-time module when you use **RUN**, but not when you use **CHAIN**. Most of the run-time environment is occupied by the run-time module, which is automatically loaded when you invoke an executable **.EXE** file requiring it. When you use **RUN** to start a program, the **.EXE** file is loaded into memory. The computer also loads the run-time module to create a fresh run-time environment. Both files reside in memory simultaneously. See Section D.5 for a diagram of the run-time libraries and memory maps.

D.5 The Run-Time Memory Maps

Section D.5 presents illustrations of two different run-time memory maps. Figure D.1 shows the run-time memory map as it appears when the run-time module **BRUN20.EXE** is used with the separate compilation method.

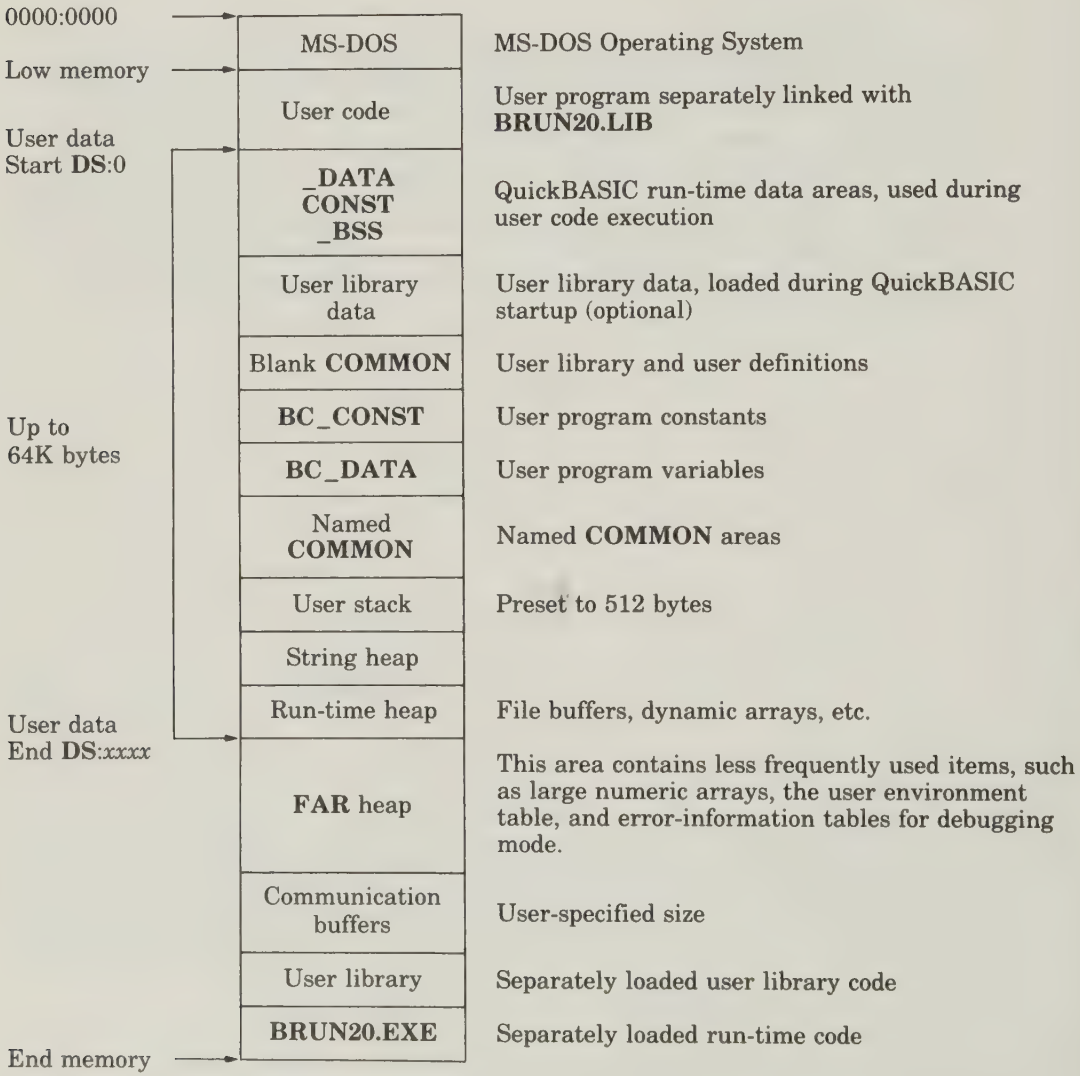


Figure D.1 The BRUN20.EXE Run-Time Module Environment

Figure D.2 shows the run-time memory map when the run-time library **BCOM20.LIB** is used.

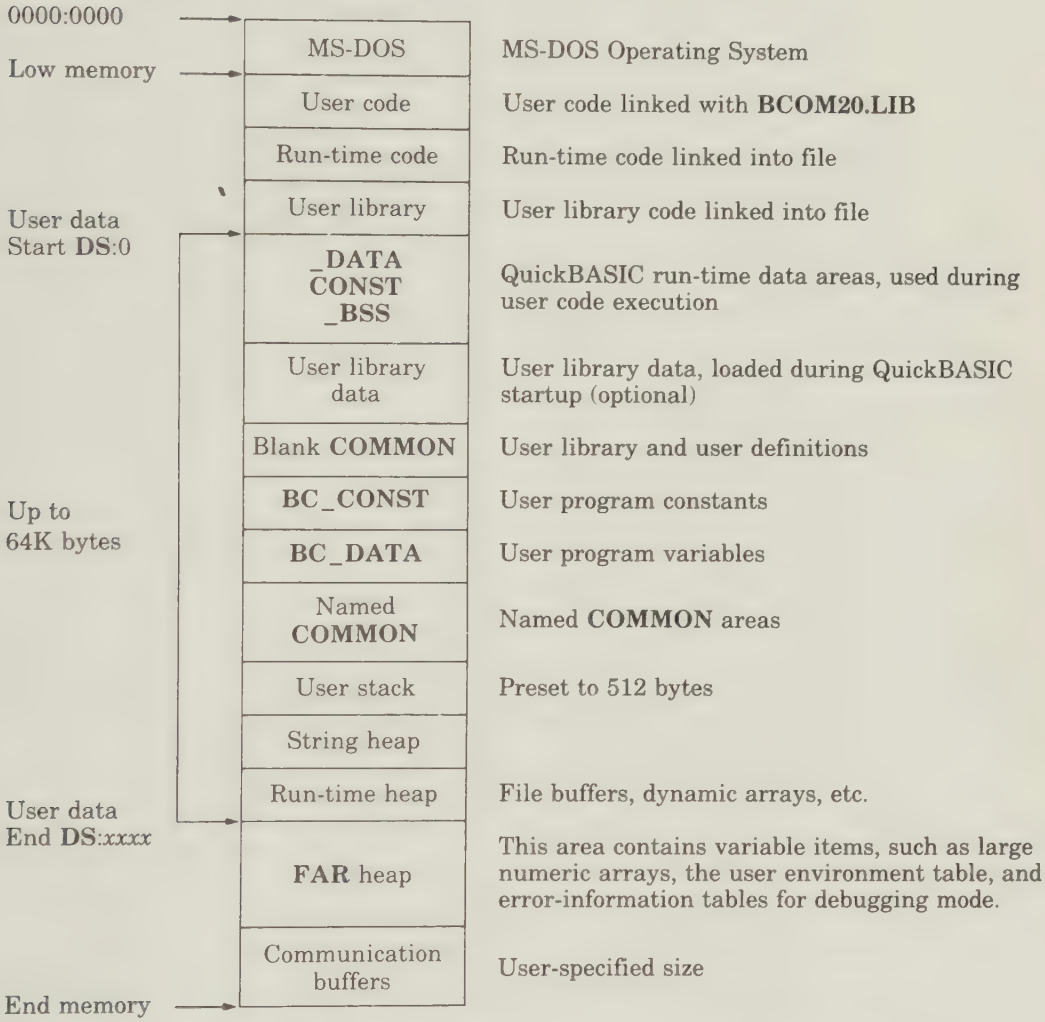


Figure D.2 The BCOM20.LIB Run-Time Library Environment

Appendix E
Error Messages

E.1 Invocation Errors 551
E.2 Compile-Time Errors 552
E.3 Run-Time Errors 562

During development of a BASIC program with the QuickBASIC compiler, three different kinds of errors can occur:

1. Invocation errors
2. Compile-time errors
3. 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, compile-time errors occur during compilation, and run-time errors occur when the QuickBASIC program is running.

This appendix lists alphabetically the error messages for each type of error, along with any error codes that are assigned.

If you are using the separate compilation method described in Appendix D, note that linker errors are documented in your DOS manual.

E.1 Invocation Errors

Invocation errors occur when you start QuickBASIC.

`Can't load source file`

QuickBASIC cannot find the source file in the location you specified. This error occurs only during batch processing.

`Error during QuickBASIC initialization`

Several conditions can cause this error. It is most commonly caused when an option is entered incorrectly. This error is also caused when there is not enough memory in the machine to load QuickBASIC. If you are loading a user library, try reducing the size of the library.

`Requires DOS 2.0 or later`

You are attempting to use QuickBASIC with an incorrect version of DOS.

`Unable to load specified user library`

The user library is not in the location you specified.

`User library is incompatible with QuickBASIC`

You have built the user library with an incompatible version of **QB.EXE**. Recompile the user library source files and rebuild the library with **buildlib**. This error should not occur in Version 2.0.

E.2 Compile-Time Errors

When errors occur while your program is compiling, the "Errors:" entry in the compile display box is incremented and compilation continues. Up to 25 compile errors can occur before compilation stops. When compilation is finished and you return to the editor, the cursor is positioned at the location of the first error. See Section 4.4.4, "Handling Compilation Errors," for more information about finding and correcting errors.

When system errors occur during compilation, compilation stops. When you press ENTER a message box opens containing the words "CRITICAL ERROR" and an error message. These errors are caused by circumstances unrelated to your BASIC program, such as pressing CTRL-BREAK during compilation, or by write-protected disk errors. The system error messages are also described in this section.

`Array already dimensioned`

This error can be caused by

- More than one **DIM** statement for same array.
- A **DIM** statement after initial use of array. Static arrays must be deallocated with the **ERASE** statement before they can be redimensioned; dynamic arrays are redimensioned with the **REDIM** statement.
- An **OPTION BASE** statement that occurs after the array it refers to is dimensioned.

`Array too big`

There is not enough user data space to accommodate the array declaration. Reduce the size of the array.

`Binary source file`

The file you have attempted to compile is not an ASCII file. All source files saved from within the BASIC interpreter should be saved with the **,A** option.

`Can't create file`

This error occurs on networked systems. The compiler attempted to write a **.OBJ** or **.EXE** file onto a write-protected disk.

`Can't create OBJ file`

QuickBASIC encountered a hardware or system error while trying to write the **.OBJ** file to disk. This error can be caused by opening the drive door while the file is being written to disk, or by opening too many files concurrently. Check the "FILES=" entry in your **CONFIG.SYS** file and try compiling the program again.

`COMMON array not dimensioned`

An array is declared in a **COMMON** statement without being dimensioned.

`COMMON out of order`

A **COMMON** statement is misplaced. **COMMON** declarations must appear before any executable statements. All BASIC statements are executable except the following:

- **COMMON**
- **DEF***type*
- **DIM** (for static arrays)
- **OPTION BASE**
- **REM**
- All metacommands

`Compiler out of memory`

The compiler needs more memory than is available. Try reducing the size of your DOS buffers, getting rid of any terminate-and-stay-resident programs, or eliminating some device drivers.

`Data memory overflow`

The program data is too big to fit in memory. This error is often caused by too many constants, or too much array data. Try turning off the debugging options. If memory is still exhausted, break your program into parts and use the **CHAIN** statement.

`Data type conflict`

The operand of an operation is an incorrect data type; for example, `V$=MID$(INT%,N)`.

`Disk full error`

There is not enough room on the disk for the compiler to write out an **.OBJ** or **.EXE** file.

`Divide by 0`

This message results from division by zero, division of the integer –32,768 by 1 or –1, or moding of the integer –32,768 by 1 or –1.

`Duplicate COMMON variable.`

A variable is declared twice in **COMMON** statements.

`Duplicate definition in user library`

There is a subprogram in the main program with the same name as a subprogram in the user library.

`Duplicate statement number`

Two program lines are assigned the same number or label. Each line number or label must be unique.

`Expected the end of a line`

The compiler expects the line to end here. This error occurs in **IF...THEN...ELSE** constructs, where a line break is required after the **THEN** statement.

`Expecting simple or array variable`

The compiler is expecting a variable argument.

`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 by the compiler. A large number of such strings can cause this error to occur. Try simplifying expressions, and assigning strings to variables.

`FOR without NEXT`

Each **FOR** statement must have a matching **NEXT** statement.

`FOR-loop index variable already in use`

This error occurs when an index variable is used more than once in nested **FOR** loops.

`Formal parameters not unique`

A function or subprogram declaration contains duplicate parameters, as in `SUB(A,B,C,A) STATIC`

`Function already defined`

This error occurs when a previously defined function is redefined.

`Function not defined`

Functions must be defined before they are used.

`IF without matching END IF`

The compiler encountered a block **IF...THEN...ELSE** statement without a terminating **END IF**.

`Illegal COMMON name`

The compiler encountered an illegal */blockname/* specification (for example, a *blockname* containing a parenthesis) in a named **COMMON**.

`Illegal DEF`*type* `character specification`

A **DEF***type* statement is entered incorrectly. **DEF** can only be followed by **DBL**, **INT**, **SNG**, **STR**, or (for user-defined functions) a blank space.

`Illegal dimension value`

An array declaration contains an illegal dimension value. Dimension values are limited to 32,767. This error can also be caused by an incorrect array specification in a **UBOUND** function.

`Illegal FOR-loop index variable`

Usually caused when an incorrect variable type is used in a **FOR**-loop index. A **FOR**-loop index variable must be an integer or real variable.

`Illegal formal parameter specification`

There is an error in a function or subprogram parameter list.

`Illegal function name`

A BASIC reserved word is used as a user-defined function name.

`Illegal separator`

There is an illegal delimiting character in a **PRINT USING** or **WRITE** statement. Use a semicolon or a comma as a statement delimiter.

`Illegal subprogram name`

The error is caused when a subprogram name is a BASIC reserved word, or a subprogram name is used twice.

`Illegal subscript syntax`

An array subscript contains a syntax error: for example, both string and integer data types.

`Illegal syntax`

Several conditions can cause this error. The most common cause is a mistyped BASIC keyword or argument.

`$INCLUDE file access error`

The include file named in the **$INCLUDE** metacommand cannot be located.

`Invalid character`

The compiler found an invalid character, such as a control character, in the source file.

linelabel: `Label not defined`

A line label is referenced (in a **GOTO** statement, for example), but does not occur in the program.

`Line too long`

Lines are limited to 255 characters.

`Math overflow`

The result of a calculation is too large to be represented in BASIC number format.

`$Metacommand error`

A metacommand is incorrect.

`Missing =`

The compiler expects an equal sign.

`Missing AS`

The compiler expects an **AS** keyword, as in `OPEN "FILENAME" FOR INPUT AS #1`.

`Missing BASE`

The compiler expected the keyword **BASE** here, as in **OPTION BASE**.

`Missing comma`

The compiler expects a comma.

`Missing equal sign`

The compiler expects an equal sign.

`Missing Event Trapping (/w) or Checking Between Statements (/v) option`

The program contains a statement requiring one of these options. See

Section 4.4.3, "Compile...", for an explanation of the options.

`Missing GOTO or GOSUB`

The compiler expects a **GOTO** or **GOSUB** statement.

`Missing IF`

There is no corresponding **IF** for an **ELSE**, **ELSEIF**, or **END IF** statement.

`Missing INPUT`

The compiler expects the keyword **INPUT**.

`Missing left parenthesis`

The compiler expects a left parenthesis.

`Missing line number or label`

There is a line number or label missing from a statement that requires one, for example **GOTO**.

`Missing minus sign`

The operand is missing from a subtraction operation.

variable: `Missing NEXT`

A **FOR** statement is missing a corresponding **NEXT** statement. The *variable* is the **FOR**-loop index variable.

`Missing On Error (/e) option`

Programs that contain **ON ERROR GOTO** or **ON** *event* **GOTO** statements must be compiled with the On Error (**/e**) option.

`Missing operand in expression`

An expression has no operand.

`Missing Resume Next (/x) option.`

Programs that contain **RESUME**, **RESUME NEXT**, and **RESUME 0** statements must be compiled with the Resume Next (**/x**) option.

`Missing right parenthesis`

The compiler expects a right (closing) parenthesis.

`Missing semicolon`

The compiler expects a semicolon.

`Missing STATIC on SUB`

A **SUB** statement has no **STATIC** attribute.

`Missing THEN`

The compiler expects a **THEN** keyword.

`Missing TO`

The compiler expects a **TO** keyword.

`Must be first item on the line`

In block **IF...THEN...ELSE** constructs, **ELSE**, **ELSEIF**, and **END IF** can only be preceded by a line number or label.

`Name too long`

BASIC identifiers, such as subprogram names, are limited to 40 characters.

`Nested function definition`

A function definition appears inside another function definition, or inside an **IF...THEN...ELSE** clause.

`NEXT without FOR`

Each **NEXT** statement must have a matching **FOR** statement.

`Program memory overflow`

You are attempting to compile a program whose code segment is larger than 64K. Try splitting the program into subprograms and placing them in the user library, or use the **CHAIN** statement.

`Simple variable already defined`

There is more than one **DIM** or **REDIM** statement for a simple variable, as in `DIM X`.

`Size of named COMMON in user library incorrect`

Named **COMMON** blocks that appear in both the main program and the user library must be the same size.

`String assignment required`

The string assignment is missing from an **LSET** or **RSET** statement.

`String constant too long`

Strings are limited to 32,767 characters.

`String expression required`

The statement requires a string expression argument.

`String variable required`

The statement requires a string variable argument.

`Subprogram error`

This is a subprogram definition error and is usually caused by one of the following:

- The subprogram is already defined.
- The program contains incorrectly nested **SUB...END SUB** statements.

`Too many arguments in function call`

Function calls are limited to 60 arguments.

`Too many dimensions`

Arrays are limited to 63 dimensions.

`Too many named COMMON blocks`

The maximum number of named **COMMON** blocks permitted is 126.

`Too many statement numbers`

The number of lines in the line list following an **ON...GOTO/GOSUB** statement exceeds 255.

`Too many variables for INPUT`

An **INPUT** statement is limited to 60 variables.

`Too many variables for LINE INPUT`

Only one variable is allowed in a **LINE INPUT** statement.

`Unimplemented command`

The command is not implemented in this version of BASIC.

`Unrecognizable statement`

You have probably mistyped a BASIC statement.

name: `Unresolved subprogram reference`

QuickBASIC could not find subprogram *name* in the program or in the user library.

`User interrupt encountered`

The CTRL-BREAK or CTRL-C key was pressed during compilation.

`Variable required here`

The compiler encountered an **INPUT**, **LET**, **READ**, or **SHARE** statement without a variable argument.

`Warning: Array not dimensioned`

An array is declared but not dimensioned. This is the only error that is not "fatal"; the program will execute, although program results may be incorrect.

`WEND without WHILE`

This error is caused when a **WEND** statement has no corresponding **WHILE** statement.

`WHILE without WEND`

This error is caused when a **WHILE** statement has no corresponding **WEND** statement.

`Wrong number of arguments`

You are using an incorrect number of arguments with a BASIC statement or function.

`Wrong number of subscripts`

The following situations cause this error:

- Illegal array dimension value
- Wrong number of subscripts

E.3 Run-Time Errors

The errors listed below can occur while the program is running. When run-time errors occur in programs that are compiled and run from within the QuickBASIC editor, the following message appears:

```
Runtime error. Press any key
```

When you press any key, the Error-Message window, which contains the run-time error message, opens at the bottom of the screen. See Section 4.4.5, "Handling Run-Time Errors," for more information.

In standalone 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 Debug (**/d**), On Error (**/e**), or Event Trapping (**/w**) option is specified in the compile command line. In those cases, the error message is also 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` *X* `in module` *module-name* `at address` *segment:offset*.

and

`Error` *X* `in line` *linenumber* `of module` *module-name* `at address` *segment:offset*.

An "**ERR** code" is listed for some errors. If an error occurs, the **ERR** variable is set to the appropriate code when an error-trapping subroutine is entered. (Error-trapping subroutines are entered via the **ON ERROR GOTO** *line* statement.) The **ERR** variable remains at this value until a

RESUME statement returns control to the main program. See Part 2, "Reference," for more information.

`Bad file mode`

An attempt is made to use **PUT** or **GET** with a sequential file, or to execute an **OPEN** with a file mode other than I, O, or R. This error also occurs when an attempt is made to read from a file opened for output or appending.

ERR code: 54

`Bad file name`

An illegal form is used for the file name with **LOAD**, **SAVE**, **KILL**, or **OPEN** (for example, a file name with too many characters).

ERR code: 64

`Bad file number`

A statement or command references a file with a file number that is not **OPEN** or is out of the range of file numbers specified at initialization.

ERR code: 52

`Break key pressed`

The BREAK key was pressed while the program was running.

`Bad record number`

In a **PUT** or **GET** statement, the record number is equal to zero.

ERR code: 63

`Cannot find BRUN20.EXE. Enter new path:`

This message appears if the run-time module is not found. "Enter new path" prompts for input of the correct path specification. This error is severe and cannot be trapped.

`Cannot find user library. Enter new path and file name:`

This message appears if the user library is not found. "Enter new path" prompts for input of the correct path specification and file name.

`Can't continue after SHELL`

Upon returning from a child process, the **SHELL** statement discovers that there is not enough memory for BASIC to continue. BASIC closes any open files and exits to DOS.

Communication buffer overflow

During remote communications, the receive buffer overflowed. The size of the receive buffer is set by the **/c** command line option. Try checking the buffer more frequently (with the **LOC** function), or emptying it more often (with the **INPUT$** function).

Device fault

A device has returned a hardware error. If this message occurs while data is 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.

Device I/O error

An I/O error occurred on a device I/O operation. The operating system cannot recover from the error.

ERR code: 57

Device timeout

The program did not receive information from an I/O device within a predetermined amount of time. Retry the operation.

ERR code: 24

Device unavailable

The device you are attempting to access is not on line or does not exist.

ERR code: 68

Disk full

All disk space has been allocated.

ERR code: 61

Disk media error

Disk-drive hardware has detected a physical flaw on the disk.

ERR code: 72

Disk not ready

The disk-drive door is open or no disk is in the drive.

ERR code: 71

Division by zero

A division by zero is encountered in an expression, or the operation of involution results in zero being raised to a negative power. Also occurs if the integer –32,768 is divided by 1 or –1, or if –32,768 is moded by 1 or –1.

ERR code: 11

Error in EXE file

This error occurs when a file is not of the correct type; it must be an executable file if it is to be executed with **RUN** or **CHAIN**. This error is severe and cannot be trapped.

FIELD overflow

A **FIELD** statement is attempting to allocate more bytes than were specified for the record length of a random file.

ERR code: 50

File already exists

The file name specified in a **NAME** statement is identical to a file name already in use on the disk.

ERR code: 58

File already open

A sequential output mode **OPEN** is issued for a file that is already open, or a **KILL** is given for a file that is open.

ERR code: 55

File not found

A **KILL**, **NAME**, **FILES**, or **OPEN** statement references a file that does not exist at the specified location.

ERR code: 53

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 **USR** function is called that has an undefined starting address.
- 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.
- Strings are concatenated to create a string greater than 32,767 characters in length.

ERR code: 5

`Input past end`

An **INPUT** statement reads from a null (empty) file or from a file in which all data has already been read. To avoid this error, use the **EOF** function to detect the end-of-file character.

ERR code: 62

`Internal error`

An internal malfunction occurred in the QuickBASIC compiler. Use the Software Problem Report form at the back of this manual to report to Microsoft the conditions under which the message appeared.

ERR code: 51

`Must link with BCOM20.LIB`

This error occurs only with executable programs created using the separate compilation method described in Appendix D. Files compiled with the **/o** option must be linked with the **BCOM20.LIB** alternate run-time library.

`Must link with BRUN20.LIB`

This error occurs only with executable programs created using the separate compilation method described in Appendix D. Files compiled without the **/o** option must be linked with the **BRUN20.LIB** run-time library.

`Must run under DOS 2.0 or later`

You are trying to run the program with the wrong version of DOS.

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.

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.

ERR code: 19

Out of data

A **READ** statement is executed when there are no **DATA** statements with unread data remaining in the program.

ERR code: 4

Out of memory

Not enough memory is available to allocate a file buffer.

ERR code: 7

Out of paper

The printer is out of paper, or is not turned on.

Out of string space

String variables exceed the allocated amount of string space.

ERR code: 14

Overflow

The result of a calculation is too large to be represented within the range allowed for floating-point numbers.

ERR code: 6

Path not found

During an **OPEN**, **MKDIR**, **CHDIR**, or **RMDIR** operation, DOS was unable to find the path specified. The operation is not completed.

ERR code: 76

`Path/file access error`

During an **OPEN**, **MKDIR**, **CHDIR**, or **RMDIR** operation, the operating system was unable to make a correct path-to-filename connection. The operation is not completed.

ERR code: 75

`Permission denied`

An attempt was made to write to a write-protected disk.

ERR code: 70

`Program too large`

Not enough memory is available to load **BRUN20.EXE**. This error is severe and cannot be trapped.

`Redimensioned array`

A second **DIM** statement was executed for an already dimensioned dynamic array.

ERR code: 10

`Rename across disks`

An attempt was made to rename a file with a new drive designation. This is not allowed.

ERR code: 74

`RESUME without error`

A **RESUME** statement is encountered before an error-trapping routine is entered.

ERR code: 20

`RETURN without GOSUB`

A **RETURN** statement is encountered for which there is no previous, unmatched **GOSUB** statement.

ERR code: 3

STOP statement executed

A **STOP** statement was encountered in the program.

String formula too complex

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.

ERR code: 16

String space corrupt during G.C. in [*linenumber* of] *module-name* at address *segment:offset*

This error occurs when an invalid string in string space is being deleted during heap compaction. The probable causes for either of the "String space corrupt" errors 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 Debug (/**d**) 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.

String space corrupt in [*linenumber* of] *module-name* at address *segment:offset*

This error occurs when an invalid string in string space is being deallocated, usually in a string assignment statement. See the listing following the error message "String space corrupt during G.C." for additional causes. This error is severe and cannot be trapped.

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 is generated only if the Debug (/**d**) option was specified at compile time.

ERR code: 9

`Syntax error`

A line is encountered that contains an incorrect sequence of characters in a **DATA** statement.

ERR code: 2

`Too many files`

The 255-file directory maximum is exceeded by an attempt to create a new file with a **SAVE** or **OPEN** statement.

ERR code: 67

`Type mismatch`

The argument types are not compatible.

ERR code: 13

`Unable to locate user library`

The user library is not in the current directory or the **LIB** environment variable search path.

`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.

`User library incompatible with user program`

The user library being accessed at run time is not the same as the user library used at compile time.

`Wrong version of runtime module`

This error occurs when the version of the **BRUN20.EXE** run-time module being used does not match the version of the QuickBASIC compiler used to generate the **.EXE** file. This error is severe and cannot be trapped.

Glossary

The definitions in this glossary are intended for use with this manual. Neither the individual definitions nor the list of terms is comprehensive.

Animate

To execute in slow motion. QuickBASIC's debugging mode has an Animate mode, which executes the program very slowly.

Base name

The portion of the file name that precedes the file name extension. For example, **SAMPLE** is the base name of the file **SAMPLE.BAS**.

Basic Input/Output System (BIOS)

The code built into system memory that provides the lowest level of functionality in a computer system. You can make BIOS calls using the **INT86**, **INT86X**, and **PTR86** routines.

BIOS

See "Basic Input/Output System (BIOS)."

Call by reference

See "Pass by reference."

Call by value

See "Pass by value."

Choose

To select a command from a menu, or an option from a dialog box.

Click

To press and quickly release one of the mouse buttons while pointing the mouse at an object on the screen.

Clipboard

A storage area that holds text you are copying or moving.

Cursor

The thin blinking line that represents the current location where QuickBASIC is ready to accept input.

Compile time

The time during which the compiler is compiling a BASIC source file.

Compiler

A set of programs that translates BASIC programs into a language understood by the computer.

Dialog Box

A box that appears when you select a menu command that requires additional information.

Double precision

A real value that is allocated 8 bytes of memory.

Drag

To point the mouse at an object on the screen and press a mouse button, then, while holding the button down, move the mouse.

Executable program

A file that contains executable program code. When the name of the file is typed at the system prompt, the statements in the file are executed.

External symbol

A symbol referenced in one assembly-language module, but defined (made **PUBLIC**) in another module. References to the symbol are resolved (filled in with the correct addresses) by the linker.

FLAGS

A register that controls the operation of many machine-level instructions.

Flashing underscore

See "Cursor."

Global variable

A variable that is available throughout the program.

Heap

An area of random-access memory that is used by BASIC as the data management area. Variables and arrays are stored here.

Highlight

The reverse-video line that marks the current selection on a menu and selected text in the work area or a text box.

Input Focus

When an item in a dialog box has the input focus, that item is the object of the next action. The item that has the input focus is highlighted either in intense video or reverse video.

Library

A directory that stores related modules of compiled code.

Line styling

A method of drawing dashed or dotted lines with the **LINE** graphics statement; this is done by specifying a 16-bit *style* mask in the **LINE** statement. In the binary representation of *style*, a "1" means "plot a point on the screen," while a "0" means "plot no point."

Link time

The time when the linker is executing, during which it collects and links relocatable object files and library files. *See also* "Compile time" and "Run time."

Linking

The process in which the linker loads modules into memory, computes absolute offset addresses for routines and variables in relocatable modules, and resolves all external references by searching the run-time library. After loading and linking, the linker saves the modules it has loaded into memory as a single executable file.

List box

A box listing the files on the disk or in the current directory. The list box appears in the Load... dialog box. *See also* "Dialog box."

Local variable

A variable that has value only within a particular unit of code, such as a main program, a subprogram, or a user-defined function.

Machine code

Instructions that a microprocessor can execute.

Main program

In a program that calls subroutines and subprograms, the calling program is the main program.

Memory map

A representation of where in memory the computer expects to find certain types of information.

Menu bar

The bar at the top of the QuickBASIC display containing the menu titles.

Metacommands

Metacommands are special commands enclosed in comments in the source file that tell the compiler to perform certain actions while it is compiling the program, for example, to produce a listing file in a certain format.

Module

A general term for a discrete unit of code.

Mouse

A small pointing device designed to fit comfortably under your hand. By moving it about on a flat surface, you can move the mouse pointer in the corresponding direction on the screen. *See alse* "Pointer."

Object file

A file that contains relocatable machine code.

Offset

The number of bytes from the beginning of a segment to a particular byte in that segment.

Pass by reference

A method of passing parameters in which the calling routine provides the called routine with the memory addresses of the parameters. This permits subroutines to change the values of the parameters.

Pass by value

A method of passing parameters in which the calling routine provides the called routine with the current value of the parameters. This prevents the possibility of the subroutine changing the value of the parameter; the subroutine only changes its copy of the value.

Physical coordinates

Pairs of numbers (x,y) that uniquely identify a pixel location on the screen. The physical coordinates for the upper-left corner of the screen are always 0,0; the maximum x (screen width), and y (screen length) values depend on the hardware configuration and the specifications set in the **SCREEN** statement.

Pointer

The reverse-video square that moves to indicate the current position of the mouse. The mouse pointer only appears if a mouse is installed. To select an item with the mouse, move the mouse until the pointer rests on the item, then press the left button.

Registers

The areas of memory where data can be stored during machine-level processing. The registers used with the 8086-family of processors are: **AX**, **BX**, **CX**, **DX**, **SP**, **SI**, **DI**, **DS**, **ES**, **SS**, **CS**, **IP**, and the **FLAGS** register. *See also* "FLAGS."

Relocatable

The term applied to a module when the code within it can be placed and run at different locations in memory. The relocatable modules

created by the compiler are an intermediate stage between source code and executable code; they are changed into executable modules by the linker and have **.OBJ** extensions.

Routine

Executable code residing in a module, and usually representing a particular feature or procedure. More than one routine may reside in a module. The run-time module contains the majority of the library routines needed to implement the BASIC language. A library routine usually corresponds to a feature or subfeature of the BASIC language.

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.

Run-time module

A module containing most of the routines needed to implement the BASIC language. It is a peculiarity of the run-time module that it is an executable **.EXE** file. The run-time module is named **BRUN20.EXE** and is, for the most part, a library of routines. It is made executable so that you can see the version number.

Scroll

To move text up and down, or left and right, to see parts that cannot fit on the screen.

Scroll bars

The bars that appear at the right side and bottom of the work area and the list box.

Select

To indicate the object the next command or option you choose will work on.

Subprogram

A unit of code delimited by the **SUB** and **END SUB** statements, and called with the **CALL** statement. *See also* "Main program."

Subroutine

A unit of BASIC code terminated by the **RETURN** statement. Program control is transferred to a subroutine by means of a **GOSUB** statement. *See also* "Main program."

Text box

A box where you type information needed to carry out a command. A text box appears within a dialog box. What you type appears in the box. The text box may be blank when the dialog box appears, or the box may contain text if there is a default option or you have selected something applicable to that command.

Tiling

A method that uses the graphics statement **PAINT** to fill a figure with a pattern, instead of a solid color.

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, ALT-F8 is a toggle that switches debugging between Step and Trace execution modes. You can press ALT-F8 to toggle between the two modes.

Unbound external

A symbol that is referenced in one assembly-language module, but is not made **PUBLIC** in another module that is linked with it. Unbound external references are usually caused by misspellings, or by omitting the name of the module containing the desired symbol from the link command line.

World coordinates

Pairs of numbers (x,y) that uniquely define any point in two-dimensional space. The numbers x and y can be any real numbers. A section of the "world-coordinate space" can be mapped to the screen with the graphics **WINDOW** statement.

Work area

The area of the screen where you enter and edit program text.

Index

,A option
  $INCLUDE files, 188
  saving interpreter files, 190
ABS function, 196
Aliasing, variable, 127
Alphanumeric line labels. *See* Line labels
Angle measurement
  converting degrees to radians, 233
  converting radians to degrees, 214
Animate defined, 571
Animate mode, 65, 112
Animation, 419
Apostrophe, entering, 9
Arctangent, 198
Arithmetic operators, 170
Arithmetic overflow, 90, 179
Array subscripts, 168
Array variables, 251
Arrays
  addressing, large numeric, 150
  assembly language subroutines, 140
  bound
    checking, 90
    error, cause, 90
  dimensioning, 168
  $DYNAMIC metacommand, 184
  elements, 168
  INT86, INT86X routines, 149
  large numeric, 186
  LBOUND function, 320
  lower-bound function, 120
  memory allocation, 188
  minimum value of subscripts, 377
  passing to assembly-language subroutines, 140
  passing with CALL, 119
  Row Order option, 91
  specifying maximum value for subscripts, 251
  specifying number of subscripts, 251
  $STATIC metacommand, 184
  subprograms parameter list form, 116
Arrays *(continued)*
  UBOUND function, 496
  unallocated, default to static, 186
  upper-bound function, 120
Arrays in Row Order option, 91
ASC function, 197
ASCII character codes, 198, 214, 523
ASCII files, 78
Assembly language, books about, 13
Assembly-language subroutines
  arguments
    location, calculating, 141
    passing requirements, 143
    referencing, 141
  arrays, passing, 140
  CALL statement, 139, 204
  coding restrictions, 143
  data group, 143
  double-precision numbers, passing, 145
  integer, passing, 144
  local variable location, 142
  name, PUBLIC symbol, 204
  PREFIX.OBJ, use, 139
  register contents, 143
  segment register values, 147
  single-precision numbers, passing, 145
  stack adjustment, 143
  stack cleanup, 142
  string data, accessing, 144
  string descriptor, 143
  string descriptor length, 205
  string, passing, 146
  string restrictions, 143
  using, 139
  values, passing to BASIC, 143
  VARPTR function, 500
ATN function, 198
AUTO, compiler prohibited, 192
Auto Save command, 74, 79

/b option, 72
Background color option, 83
Background music, 362, 392, 397
.BAS file execution by compiler, 193
Base name defined, 571
BASIC run-time errors. *See* Errors
BASIC, books about, 12
Batch files, QuickBASIC Version 1.0, 535
BCOM20.LIB file
  advantages, 543
  described, 18
  invocation, 543
  memory map, 547
BEEP statement, 199
Bibliography, 12
BIN directory, 537
Binary files, 78
Binary files, executing within a program, 545
BIOS
  calling, 147
  defined, 571
Blank COMMON statement, 228
BLOAD statement, 193, 200
Block IF...THEN...ELSE, 296
Boolean expressions, 295, 296
Boolean operations, 174
Border, screen, controlling, 83
Bounds checking, arrays, 185
BP register, 141, 143
BRUN20.EXE file
  described, 18
  memory map, 546
  Obj(BRUN.LIB) option, use with, 93
BRUN20.LIB file
  advantages, 541
  described, 18
  invocation, 543
  run-time module library, 18
BSAVE statement, 193, 202
BUG.BAS file, 18
Buildlib command, 66, 133
BUILDLIB.EXE file, 18, 19
Built-in functions, 176
Buttons
  Cancel
    Change... command, 108
    described, 108
  Cancel *(continued)*
    Find... command, 106
    New command, 75
    Print... command, 80
  Change, 108
  Change All, 108
  circular
    moving between, 33
    selecting, 33
  Command, 33
  mouse, 25
  Next Error, 84
  No New command, 75
  Save, 60
  Skip, 108
  square, 33

/c option, 72
CALL ABSOLUTE statement, 205
CALL statement, 204
  assembly-language subroutines, use with, 139
  BASIC subprograms, 117
  compiler/interpreter differences, 193
  INT86 routine, 148
  INT86X routine, 148
  PTR86 routine, 150
CALLS statement, 205, 207
CALLSHAP.BAS file, 18
Cancel command button, 108
  Change... command, 108
  Find... command, 106
  New command, 75
  Print... command, 80
Caption bar
  controlling, 83
  defined, 28
Cardioid, 461
Carriage return, 301, 327, 329, 518
Case
  Find... command, 106
  line labels, 159
Cassette BASIC, 190
CDBL function, 208
CHAIN statement
  BASIC programs only, 545
  BCOM20.LIB, 542
  BRUN20.LIB, 542

CHAIN statement *(continued)*
compiler/interpreter differences, 193
described, 209
RUN, used with, 441
separate compilation, used with, 545
Change All command button, 108
Change command button, 108
Change... command
All option, 108
Cancel, 108
Change, 108
Change All, 108
described, 105
dialog box, 107
error, correcting, 108
Match Upper/Lowercase option, 107
Search menu, in, 51
Skip, 108
Whole Word option, 107
Change To text box, 107
Characters recognized by BASIC, 157
CHDIR statement, 212
Check boxes
moving between, 33
selecting, 33
Checking Between Statements option, 92
Choose defined, 571
Choosing commands, 28
CHR$ function, 213
CINT function, 214
CIRCLE statement, 215
Circular buttons
moving between, 33
selecting, 33
CLEAR statement, 218
Click
defined, 26, 571
double click, defined, 26
Clipboard
copying text into, 44
defined, 102, 572
inserting text from, 45
CLOSE statement, 220
CLS statement, 221
/cmd option, 73, 186
Color. *See* Display
Color. *See* Screen
COLOR statement, 222
COM statements, 92, 225
Command button
moving between, 33
selecting, 33
COMMAND$ function, 73, 186, 226
Command line arguments passing to BASIC program, 226
Commands
Auto Save, 74, 79
buildlib, 66, 133
canceling, 29, 33
Change... command, 51
All option, 108
described, 105, 107
choosing from menus, 28
Compile, 39, 85
Compile... command, 85
Copy, 44, 102
Cut, 48, 102
DISKCOPY, 17
Errors, 82, 84
Find... command, 104, 105
Label, 105, 108
link, 541
Load... command, 37, 74
menu. *See also* Menu; specific menu names
canceling, 29
choosing, 28, 29
displaying, 29
shortcut keys, 30
MODE, 80
New, 74, 75
Next Error, 58, 84, 105, 109
Options... command, 82
Paste, 45, 102, 103
Print... command, 74
qb, 21, 71
Quit, 21, 81
Quit command, 74
Repeat Last Find, 104, 106
Save... command, 59
Save, 74
Search menu, 104
Selected Text, 104, 106
SET, 536
Shell, 74, 81
Start, 42, 85
starting QuickBASIC, 71

Comments
  introducing, 160
COMMON statement
  BCOM20.LIB, 542
  blank COMMON, 228
  BRUN20.LIB, 542
  compiler/interpreter differences, 193, 194
  described, 185, 228
  DIM, restriction with, 184
  named COMMON, 126, 228
  order of variables, 230
  SHARED attribute, 122
  user library subprogram, 125
Communications event trapping See Event trapping, 92
Compilation errors, 94
Compile command, 85
Compile... command
  Arrays in Row Order option, 91
  Debug option, 90
  dialog box, 89
  Event Trapping option, 92
  Exe option, 93
  Memory option, 92
  Minimize String Data option, 91
  Obj(BCOM.LIB) option, 93
  Obj(BRUN.LIB) option, 66, 93
  object file options, 93
  On Error option, 91
  optimization options, 93, 94
  option settings, default, 87
  options, 88
  output options, 93
  Resume Next option, 92
  shortcut key, 31
  Size option, 94
  Speed option, 93
Compile time
  defined, 572
  error messages, 552
Compiler
  defined, 572
  errors, 552
  installation. *See* setup
  options, 88
  setup
    files required, 19
    hard disk, 20
  setup *(continued)*
    one floppy-disk drive, 20
    two floppy-disk drives, 19
  starting, 35
Compiler/interpreter comparison, 189
Compiler/interpreter differences
  CALL statement, 205
  CALL ABSOLUTE statement, 205
  CALLS statement, 207
  CHAIN statement, 209
  CLEAR statement, 219
  COM statement, 225
  DEF SEG statement, 247
  DEFtype statements, 249
  DIM statement, 251
  DRAW statement, 255
  ERL function, 267
  FIELD statement, 274
  FOR...NEXT statements, 281
  IF...THEN...ELSE statement, 296
  ON COM statement, 352
  ON ERROR GOTO statement, 353
  ON KEY statement, 359
  ON PEN statement, 361
  ON STRIG statement, 365
  ON TIMER statement, 367
  PRINT statement, 407
  RESUME statement, 433
  STOP statement, 472
  STRIG function, 476
  STRIG statement, 476
  SYSTEM statement, 485
  TAN function, 488
  TROFF statement, 494
  TRON statement, 494
  VARPTR$ function, 503
Compiling
  into memory, 92
  with option, 88
  without options, 38, 85
CONT, compiler prohibited, 192
Converting interpreted programs, 189
Copy command
  described, 102
  how to use, 44
  shortcut key, 31
Copying program lines, 43
COS function, 233
Critical errors, 552

CSNG function, 234
CSRLIN function, 235
CTRL-BREAK
  checking, 90
  Debug option, 90
  enabling, 91
Cursor
  defined, 27, 572
  moving, 100
  tab control, 83
Cut command
  described, 102
  how to use, 44
  shortcut key, 31
CVD function, 236
CVI function, 236
CVS function, 236

/d option, 73
DATA statement, 238, 430
Data types
  double-precision floating-point numeric, 162
  integer numeric, 161
  single-precision floating-point numeric, 161
  string, 161
DATE$ function, 240
DATE$ statement, 241
Debug option
  code size, effect on, 90
  purpose, 90
  required for debugging, 109
  run-time error information, 95
  static array bounds checking, 185
  use, 90
Debugging, 494
Debugging mode
  Animate mode, 65, 112
  Debug option required, 109
  display, 62, 110, 112
  exiting, 65, 111
  Help line, 110, 112
  program execution, 63, 65
  purpose, 109
  source lines, number displayed, 112
  starting, 61, 109
  Step mode, 63, 112
Debugging mode *(continued)*
  subprogram tracing, 64
  Trace mode, 64, 112
  TRON,TROFF, 109
Declaring variable types, 166
DEF FN statement
  compiler/interpreter differences, 194
  described, 242
  exit from, 270
DEF SEG statement, 247
DEF*type* statements
  compiler/interpreter differences
    DEF USR statement, 192
    DEFDBL statement, 193
    DEFINT statement, 167, 193
    DEFSNG statement, 193
    DEFSTR statement, 193
  described, 248
DELETE command, 192
Deleting text, 48
DEMO.BAS, 18, 36
Device status information, 266
DGROUP group, 143
Dialog box
  Change..., 107
  Compile..., 89
  defined, 31, 572
  Find..., 105
  Load..., 76
  Options..., 82
  Print..., 80
  Save..., 79
  selecting items, 33
  text box defined, 32
DIM statement
  array dimensioning, 184
  COMMON, restriction with, 184
  compiler/interpreter differences, 193
  described, 251
  SHARED attribute, 122
  use, 184
DIRECTION keys options, moving between, 33
Directory, changing with list box, 78
Disabling metacommands, 187
Disk contents, listing, 77
DISKCOPY command, 17
Disks, backing up, 17
Display Options, 83

Display
  changing, 82
  color
    options, 83
    selecting, 83
  memory, PCOPY, 383
  monochrome, 72, 83, 446
Division by zero, 172
Double click defined, 26
Double precision
  converting to, 208
  defined, 572
  PRINT, used with, 406
Double-precision numbers, 164
Drag defined, 26, 572
DRAW statement, 193, 253
DS register, 140, 143
Dynamic arrays *See also* Arrays
  defined, 184
  $DYNAMIC metacommand, 188
  ERASE statement, 184, 265
  memory space allocation, 184
  REDIM statement, 184, 425
  redimensioning, 184
$DYNAMIC metacommand, 184, 188
  See also Metacommands

/e option, 73
EDIT command, 192
Edit menu
  clipboard, 102
  Copy command, 102
  Cut command, 102
  Paste command, 102
  shortcut keys, 102
Editing
  keyboard commands, 101
  practice session, 43
END statement, 188, 194, 258, 289
END SUB statement, 115, 480
ENVIRON$ function, 260
ENVIRON statement, 262
Environment variables
  creating, 536
  defined, 536
  hard disk, 537
  LIB, 536
  PATH, 536
Environment variables *(continued)*
  two floppy drives, 537
Environment-string table, 260, 262
EOF function, 264
ERASE statement
  compiler/interpreter differences, 194
  deallocating with, 185
  described, 265
  dynamic array, effect on, 184
  static array, effect on, 184
ERDEV function, 266
ERDEV$ function, 266
ERL function
  compiler/interpreter differences, 192
  described, 267
  line labels, 160
ERR function, 267
ERR function, interpreter differences, 192
ERROR statement, 268
ERROR statement, interpreter differences, 192
Error
  codes, 267, 268
  command, 94
  handling, 353
    compiler/interpreter differences, 191
    messages, 90
    options, 91
    routines, subprograms, 128
    statements, compile option, 91
  location
    Debug option required, 90
    On Error option required, 91
  messages, 84, 551
  trapping
    Compile Options, 89
    Debug option, 90
    ERROR statement, 262
    effect on code size, 91
    line 0, 159
    RESUME, used with, 432
Error-handling
  error code, 267
  error line number, 267
Error-Message window, controlling, 84, 94
Errors

Errors *(continued)*
command, 82, 84
compilation, 94
compilation, finding, 109
compile-time, 552
correcting, 94
display during compilation, 41
finding, correcting, 55
invocation, 551
run-time, 95, 562
ESC key, closing a menu, 29
Event handling routines, subprograms, 128
Event trapping
between lines, 92
between statements, 92
compile option, 191
EX.BAS, 18
.EXE files, 441, 442
Exe option, 93
Executable files, 78, 93
Executable program defined, 572
Executing a program, 41, 85, 544
Execution, continuous, 65, 112
Execution time. *See* Program execution
EXIT statement
described, 270
use in multiline functions, 243
use in subprograms, 481
Exiting QuickBASIC, 21, 60, 81
EXP function, 271
Expressions
conversion of operands, 178
definition, 169
External symbol defined, 572

FIELD statement, 161, 273
File access
LOCK statement, 335
UNLOCK statement, 335, 498
File menu. *See also* Menus
Auto Save command, 74, 79
described, 74
displaying, 36
Load... command, 74, 75
New, 74
Print... command, 74, 79
Quit command, 74, 81
File menu. *See also* Menus *(continued)*
Save command button, 78
Save... command, 74
Shell command, 74, 81
File name changing, 79
FILES statement, 276
Files
ASCII format, 190
backing up, 17
$INCLUDE, use restrictions, 188
loading, 77
loading, ASCII files only, 78
object, defined, 575
printing, 79
random, 318, 331
random-access, 274, 284, 343, 348, 368, 417, 440
saving, 74
saving, automatic, 74
separate compilation method, setup, 537
sequential, 264, 304, 318, 329, 331, 338, 368, 409, 519
Find and Verify button, 108
Find... command
canceling, 106
described, 104, 105
dialog box, 105
Find Next option, 106
Find What text box, 106
Match Upper/Lowercase option, 106
Whole Word option, 106
Find Next option, 106
Find What text box
Change... command, 107
Find... command, 106
Finding text, 105
FIX function, 277
Fixed-point constants, 163
FLAGS register defined, 572
Floating-point constants
double-precision, 164
single-precision, 164
Floating-point numbers
allowable range
double precision, 162
single precision, 161
exponent, 162
mantissa, 162

Floating-point numbers *(continued)*
most significant bit, 161
Foreground color option, 83
Formatting output
numbers, 412
strings, 411
FOR...NEXT statement, 279
FOR...NEXT statement,
compiler/interpreter differences, 194
FRE function, 283
FRE function, interpreter differences, 194
Function-key event trapping. *See* Event trapping
Function variables, declaring as local, 244
Functional operators, 176
Functions. *See also* specific function name
contrasted with statements, 195
enhanced in compiler, 193
intrinsic, 176
new. *See* New statements/functions
prohibited by compiler, 192
user-defined, 177, 242

/g option, 72
GET statements
File I/O, 274, 284
graphics, 286
Global variable defined, 573
GOSUB...RETURN statements
compiler/interpreter differences, 194
described, 288
subprogram, use in, 128
GOTO statement
compiler/interpreter differences, 194
$INCLUDE, used with, 188
required with line label, 160
subprogram, use in, 128
subroutines, used with, 288
syntax, 292
Graphics, 286, 418
Graphics macro language, 253
GWCOM.OBJ, 18, 543

Heap defined, 573
Help window, 21
HEX$ function, 294
Hexadecimal, 294
Hierarchy of operations, 169
Highlight defined, 573

IF...THEN...ELSE statement
compiler/interpreter differences, 194
described, 295
GOTO, when required, 160
line labels, 160
$INCLUDE file, saving, 86
$INCLUDE metacommand, 188
Information line, debugging mode, 62, 110
INKEY$ function, 298
INP function, 300
Input focus
changing, 32
defined, 31, 573
INPUT$ function, 305
INPUT statement, 273, 301
INPUT# statement, 304
Inserting text, 49, 103
Installing the compiler, 19
INSTR function, 307
INT function, 309
INT86 interrupt routine, 135, 147
INT86X interrupt routine, 135, 147
Integer constants
decimal, 163
FIX, used with, 277
hexadecimal, 163
octal, 163
Integer division, 171
Integer, converting to, 214, 277, 309
Interpreter programs, converting, 189
Interpreter/compiler comparison, 189
Interrupt support routines, 135
Intrinsic functions, 176
IOCTL$ function, 310
IOCTL statement, 311

Joystick event trapping. *See* Event trapping

KEY(*n*) statements
  Checking Between Statements option, 191
  described, 314
  Event Trapping option, 92, 191
KEY statements, soft key, 312
Keyboard scan codes, 316
KILL statement, 318

/l option, qb command, 72, 134
Label command, 105, 108
Large numeric arrays, 186
LBOUND function, 120, 320
LBOUND/UBOUND, 120, 186, 320
Leaving QuickBASIC, 21, 60, 81
LEFT$ function, 322
LEN function, 144, 323
LET statement, 273, 324
LIB directory, 537
LIB environment variable
  hard disk, 537
  two floppy drives, 537
  user library search path, 134
Library defined, 573
Library, user. *See* User library
Lightpen event trapping. *See* Event trapping
Line 0, effect on error trapping, 159
Line editor commands, INPUT statement, 302
Line feed, 301, 327, 329, 518
LINE INPUT statement, 327
LINE INPUT# statement, 329
Line labels
  alphanumeric, 158, 159
  case, significance, 159
  finding, 108
  RESUME statement, 160
  use of, 158, 160
  use with GOTO, 160
Line length restrictions, 161
Line numbers
  checking, 90
  example of, 159
  in error messages, On Error option required, 91
  restrictions, 159
  RESUME statement, 160
Line numbers *(continued)*
  use of, 158
Line printer, 341, 342
LINE statement, 325
Line styling, 325, 573
Line width, 194
link command, 541
Link time, 573
LINK.EXE file, 18
Linking
  BCOM20.LIB, 542
  BRUN20.LIB, 541
  defined, 573
List box defined, 574
  See also Load... command
LIST command, 192
LLIST command, 192
LNA, 186
LOAD command, 192
Load... command
  choosing, 37
  description, 74, 75
  file type restrictions, 78
  list box
    listing files with, 77
    loading files with, 77
Load... dialog box, 77
Loading a program, 36, 74, 75, 77
LOC function, 331
Local variable defined, 574
LOCATE statement, 332
Locating errors, 94
LOCK...UNLOCK statements, 186, 335
LOF function, 338
LOG function, 340
Logical operators, 173
Loops, 279, 508
LPOS function, 341
LPRINT statement, 342
LPRINT USING statement, 342
LPT line width, 194
LPT1 device, 74, 80
LSET statement, 343, 440

Machine code defined, 574
Main program defined, 574
Map, memory

Map, memory *(continued)*
BCOM20.LIB run-time library environment, 546
BRUN20.EXE run-time module environment, 546
in-memory execution, 152
separate compilation, 546
Match Upper/Lowercase option
Change... command, 107
Find... command, 106
MDPA, 446
Memory map
BCOM20.LIB run-time library environment, 546
BRUN20.EXE run-time module environment, 546
defined, 574
Memory
clearing, 74, 75
option, 92
requirements, 6
run-time memory map, 152
Menu bar
controlling, 83
defined, 28, 574
Menu command
canceling, 29
choosing, 28
shortcut keys, 30
Menus. *See also* specific menu name
Menus
closing, 29
displaying, 29
Edit, 101
File, 74
Run, 39
Search, 104
View, 82
MERGE, compiler prohibited, 192
Metacommands. *See also* specific metacommand
defined, 187, 574
disabling, 187
$DYNAMIC, 184, 188
$INCLUDE, 188
purpose, 187
$STATIC, 184, 188
syntax, 187
MID$ function, 345
MID$ statement, 346
Minimize String Data option, 91
MKD$ function, 348
MKDIR statement, 347
MKI$ function, 348
MKS$ function, 348
MODE command, 80
Module
defined, 574
relocatable, 575
run-time. *See* Run-time
Modulo Arithmetic, 172
Monochrome display, 446
Monochrome monitor
color graphics card, used with, 72
display color, 83
MOTOR statement, 192
Mouse
button, 25
changing directories with, 78
commands
canceling with, 30
choosing with, 30
compatible types, 26
compiling with, 40
copying text, 44
cursor, moving, 101
defined, 574
dialog box, using with, 33
file, loading with, 38
loading a file with, 77
pointer defined, 26
running a program, 42
Save... command, using with, 60
scrolling with, 34
selecting text, 101
selecting with, 33, 34
terms defined, 26
using, 26
MS-DOS, books about, 13
Multiline functions, run-time considerations, nesting, 244
Music, background, 362, 363

NAME statement, 349
Naming files, 74
New command, 74, 75
NEW command, 192

New statements/functions
COMMAND$, 186
LBOUND/UBOUND, 186
LOCK...UNLOCK, 186
PALETTE, 186
PCOPY, 186
REDIM, 186
SADD, 186
SHARED, 186
STATIC, 186
SUB...END SUB, 186
Next Error command
button, 84
choosing, 58
described, 105, 109
handling compilation errors, 94
shortcut key, 31
No more errors message, 94, 109
Notational conventions, 11
Numeric constants, 163
Numeric conversions
double-precision, 208
integer, 214, 277, 309
single-precision, 234

/o option, 73, 542
Obj(BRUN.LIB) option, 93
Object files. *See also* User library
defined, 575
producing, 93
OCT$ function, 350
Octal conversion, 350
Offset defined, 575
ON COM statement, 351
On Error compile option, 91, 191
ON ERROR GOTO statement
compiler/interpreter differences, 192
described, 353
line 0, 159
On Error option required, 91, 191
option with RESUME statement, 92
ON ERROR GOTO...RESUME construction, 91
ON *event* GOSUB, Event Trapping option required, 191
ON *event* GOTO, line 0, 159
ON GOSUB statement, 355
ON GOTO statement, 355
ON KEY statement, 357
ON PEN statement, 360
ON PLAY statement, 362
ON STRIG statement, 364
ON TIMER statement, 366
OPEN statement, 194, 368
OPEN COM statement, 374
Operators
arithmetic, 170
functional, 176
hierarchy, 169
logical, 173
relational, 172
string, 177
Optimization options, 93, 94
OPTION BASE statement, 377
Options... command
border option, 83
caption bar option, 83
color options, 83
described, 82
dialog box, 82
menu bar option, 83
scroll bars option, 83
tab stops option, 83
Options
Compile... command, 88
Debug, 185
selecting from dialog box, 33
OUT statement, 378
Output line width, 510
Overflow, 90, 172, 271, 488

PAINT statement, 379
PALETTE statement, 186, 383
PALETTE USING statement, 383
Parameter passing
by reference, 118
by value, 118
SHARED attribute, 122
SHARED statement, 124
single module, SHARED attribute, 122
Pass by reference defined, 575
Pass by value defined, 575
Passing parameters. *See* Parameter passing
Paste command, 31, 45, 102, 103

PATH environment variable
  hard disk, 20, 537
  linker, use with, 536
  two floppy drives, 537
PCOPY statement, 186, 387
PEEK function, 388, 402
PEN function, 389
PEN ON/OFF/STOP statements
  Checking Between Statements option required, 191
  described, 391
  Event Trapping option required, 92, 191
Physical coordinates
  defined, 504, 575
  mapping to world coordinates, 398
Physical viewport, 504
PLAY function, 392
PLAY ON/OFF/STOP statements
  Checking Between Statements option required, 191
  described, 397
  Event Trapping option required, 191
PLAY
  compiler/interpreter differences, 193
  described, 393
PMAP function, 398
POINT function, 400
Pointer
  defined, 575
  mouse, defined, 26
Pointing defined, 26
POKE statement, 388, 402
POS function, 403
Practice session, 35
PREFIX.ASM file, 18
PREFIX.OBJ file, 18, 19, 139
PRESET statement, 404
Press defined, 26
Print... command
  described, 74, 79
  dialog box, 79
  output device, 80
  Print File option, 80
  Print Selected Text option, 80
  stopping, 80
Print File option, 80
Print Selected Text option, 80
PRINT statement, 406
PRINT# statement, 409
PRINT# USING statement, 409
PRINT USING statement, 411
Printer output device, 80
Printing a file, 79
Program
  clearing from memory, 75
  compiling, 38
  converting to compiled, 189
  executable, defined, 572
  executing, 85
  execution, speeding up, 190
  loading, 36, 75
  main, defined, 574
  overflow error, 91
  running, 41, 85, 544
  saving, 59, 78
  termination, 258
PSET statement, 415
PTR86 interrupt routine, 135, 150
PUBLIC symbol, assembly-language subroutine name, 204
PUT statements
  File I/O, 273, 417
  Graphics, 418

/q option, 73
qb command, 21
  arguments, 72
  /b option, 72
  /c option, 72
  /cmd option, 73
  compile options, 73
  /d option, 73
  /e option, 73
  /l option, 72, 134
  /o option, 73
  options, 72
  /q option, 73
  /r option, 73
  /s option, 73
  syntax, 71
  /v option, 73
  /w option, 73
  /x option, 73
QB.EXE file, 18, 19
QuickBASIC compiler. *See* Compiler
QuickBASIC Version 1.0 batch files, 535

Quit command, 21, 74, 81
Quitting QuickBASIC, 81

/r option, 73
RAM required, 6
Random-access
  file buffers, 343, 440
  files, 273, 284, 348, 368, 417
Random data files, 318
Random numbers, 422, 438
RANDOMIZE statement, 422, 438
READ statement, 424, 430
README.DOC, 18
REDIM statement. *See also* New statements/functions
  described, 425
  dynamic arrays, use with, 184
  SHARED attribute, 122
  static array, effect on, 184
Register values, INT86, INT86X, 149
Registers. *See also* specific register name
  defined, 575
  INT86,INT86X, 149
Relational operators, 172
Relocatable defined, 575
REM statement, 428
Renaming files, 78
RENUM command, 192
Repeat Last Find command, 31, 104, 106
Replacing text, 46, 107
Reserved words, 525
RESET statement, 429
RESTORE statement, 430
RESUME 0 statement
  line 0, 159
  Resume Next compile option, 92, 191
  with ON ERROR GOTO, 92
Resume Next option, 92
RESUME NEXT statement, 92, 160, 191
RESUME statement
  alphanumeric line labels, 160
  compiler/interpreter differences, 192, 193, 194
  described, 432
  Resume Next compile option, 92
RESUME statement *(continued)*
  Resume Next option required, 191
RET *n*, far, 143
RETURN statement
  Debug option, 90
  described, 434
  GOSUB, checking for, 90
  not equivalent to EXIT statement, 244, 481
  subprogram, use in, 128
  syntax, 288
RETURN without GOSUB error, cause, 90
RIGHT$ function, 435
RMDIR statement, 437
RND function, 422, 438
Routine defined, 576
RSET statement, 343, 440
Run menu
  Compile... command, 88
  Compile command, 85
  displaying, 39
  shortcut keys, 85
  Start command, 85
RUN statement, 193, 441, 545
Running a program, 41, 85, 544
Run-time module defined, 576
Run-time
  defined, 576
  environment, 545
  errors, 95
  memory map, 152
  module
    loading, 545
    search order, 544
  routines, invoking, 93
Runtime
  error messages, 562

/s option, 73
SADD function, 144, 146, 186, 443
Sample compilation, 35
Save... command
  described, 74
  dialog box, 78
  file name, changing, 79
  saving a program, 59
SAVE command, 190, 192

Saving a program
  ASCII format, 190
  Auto Save command, 74
  Save... command, 59, 78
Scan codes. *See* Keyboard scan codes
SCREEN function, 445
SCREEN statement, 446
Screen. *See also* Display
  description, 26
  elements, controlling, 83
SCRN line width, 194
Scroll bars
  controlling, 83
  defined, 28, 576
Scroll box defined, 34
Scroll defined, 576
Scrolling, 34, 100
Search menu
  Change... command, 105
  described, 104
  Find... command, 104, 105
  Label command, 105, 108
  Next Error command, 105, 109
  Repeat Last Find command, 104, 106
  Selected Text command, 104, 106
  shortcut keys, 105
Searching and replacing, 51
Select defined, 576
Selected Text command
  described, 104, 106
  shortcut key, 31
Selecting text, 33, 43, 101
Separate compilation method
  files required, 536
  setup
    hard disk, 537
    two floppy drives, 536
  user library requirements, 544
Sequential files, 264, 304, 318, 329, 331, 338, 368, 409, 519
SET command, 536
Setting up the compiler, 19
SGN function, 455
SHARED attribute, 122
SHARED statement. *See also* New statements/functions
  described, 456
  form, 124
  single module, 124
Shell command
  described, 74, 81
  exiting, 81
SHELL statement, 458
SHIFT-TAB key, dialog box, use in, 33
Shortcut keys
  described, 30
  Edit menu, 102
  Run menu, 85
  Search menu, 105
SIN function, 461
Single precision, 406
Single-precision numbers, 164
Size option, 94
Skip command button, 108
SMALLERR.OBJ, 18, 543
SOUND statement, 462
Source file format, 190
Source line, debugging mode, 62, 110
SPACE$ function, 464
SPC function, 465
Special characters, 157
Speed option, 93
Spiral of Archimedes, 233
SQR function, 466
SQUARE.BAS file, 18
SS register, 140, 143
Stack. *See* Assembly language subroutines
Start command, 31, 42, 85
Starting QuickBASIC, 21, 71
Statements enhanced in compiler
  COMMON, 194
  DEF FN, 194
  END, 194
  ERASE, 194
  FOR...NEXT, 194
  GOSUB...RETURN, 194
  GOTO, 194
  IF...THEN...ELSE, 194
  OPEN, 194
  RESUME, 194
  WIDTH, 194
Statements. *See also* specific statement name
  compiler prohibited, 192
  contrasted with functions, 195
  enhanced in compiler, 193
  executable, 160

Statements. *See also* specific statement name *(continued)*
execution
compiler, 190
interpreter, 190
new. *See* New statements/functions
nonexecutable, 160
prohibited by compiler, 192
requiring modification in compiler, 193
Static arrays. *See also* Array
bounds checking, 185
defined, 184
ERASE, 265
ERASE, effect, 184
REDIM, effect, 184
$STATIC metacommand, 188
STATIC attribute, 115
$STATIC metacommand. *See also* Metacommands
array memory allocation, 188
metacommand, 188
STATIC statement, 186, 467
Step mode, 63, 112
STICK function, 471
STOP statement, 289, 472
STR$ function, 474
STRIG function, 475
STRIG ON/OFF/STOP statements
Checking Between Statements option required, 191
described, 477
Event Trapping option required, 92, 191
String constants, 162
String descriptor length, 205
String expression address, 186
STRING$ function, 478
String functions, 236, 307, 322, 323, 345, 435, 474, 499
String literals, 91
String operators, 177
String space compression, 283
String variables, 327, 329
Strings
assembly-language subroutines, 143
concatenation, 177
descriptor, 143
length, LEN function, 144
Strings *(continued)*
location, SADD function, 144
SUB statement, formal parameter list, 116
SUB...END SUB statements. *See also* New statements/functions
described, 480
exit from, 270
STATIC attribute, used with, 115
subprogram delimiter, 115
subprogram variables, 456, 467
Subprograms
argument, parameter list errors, 126
arrays, passing, 116
beginning and end marking, 480
CALL, 204
CALL ABSOLUTE, 204
CALLS, 207
creating, example, 47
defined, 115, 576
entering, 128
error-handling routines, 128
event-handling routines, 128
exiting, 128
format, 115
invoking with CALL, 117
prohibited expressions, 116
user library
example, 232
naming restrictions, 125
parameter passing, 125
variable aliasing errors, 126
variables, 118, 479, 491
Subroutines
branching and returning, 288
branching to a specified line, 355
defined, 577
return to execution control, 434
tracing, 112
Subscripts
maximum value of, 251
minimum value of, 377
specifying number of, 251
SWAP statement, 483
Symbol, external, defined, 572
Syntax notation, 9
System calls, 147
System errors, 552
System requirements, 6

SYSTEM statement, 485

TAB function, 486
TAB key dialog box, use in, 33
Tab stops, controlling, 83
TAN function, 488
Text box
  defined, 32, 577
  errors, correcting, 32
Text Not Found box, 106
Text
  finding, 105
  replacing, 107
  selecting, 101
Tiling, 380, 577
TIME$ function, 489
TIME$ statement, 490
Timer event trapping. *See* Event trapping
TIMER function, 491
TIMER ON/OFF/STOP statements
  Checking Between Statements option required, 191
  described, 492
  Event Trapping option required, 92, 191
Toggle defined, 577
Trace mode, 64, 112
TRIANGLE.BAS file, 18
TROFF statement, 109, 494
  Debug compile option required, 90, 191
  debugging mode, 109
  described, 494
TRON statement, 109, 494
  Debug compile option required, 90, 191
  debugging mode, 109
  described, 494
Type conversion, 178
Type mismatch error message, 178

UBOUND function, 120, 186, 496
Unbound external defined, 577
Undo command
  described, 102
  shortcut key, 31, 102
UNLOCK statement, 186, 335, 498
Unresolved external symbol, 577
UPDATE.DOC file, 18
User library
  assembly language subroutines
    files required, 19
  building, 66
  building, files required, 19
  buildlib command, 133
  creating, 133
  defined, 65
  file naming, 134
  INT86, 135
  INT86X, 135
  loading, 134
  location, 134
  modifying, 133
  Obj(BRUN.LIB) option, use with, 93
  object files, creating, 65
  practice session, 65
  PTR86, 135
  search order, 544
  separate compilation, use with, 544
  specifying, 134
  USERLIB.EXE, 134
  using, 66
User-defined functions, 177
USERLIB.EXE file, 19, 134
USERLIB.OBJ file, 18, 135
USR, compiler prohibited, 192

/v option, 73
VAL function, 499
Variables
  aliasing, 127
  array, 251
  data type, 248
  described, 165
  global
    defined, 573
    function definitions, 186, 244, 467
    subprograms, 186, 456, 467
  local
    defined, 574
    function definitions, 186, 244, 467
    subprograms, 186, 456, 467
    use in assembly-language subroutines, 142

Variables *(continued)*
passing to subprograms
by reference, 118
single module, 122, 124
SHARED attribute, 122
SHARED statement, 124
String, 327, 329
subprogram aliasing errors, 126
VARPTR$ function, 502
DRAW, use with, 193
PLAY, use with, 193
VARPTR function
converting to segment/offset value, 150
described, 500
View menu
described, 82
Errors command, 84
Options... command, 82
VIEW PRINT statement, 506
VIEW statement, 504

/w option, 73
WAIT statement, 507
WEND statement, 508
WHILE statement, 508
Whole Word option
Change... command, 107
Find... command, 106
WIDTH statement
compiler/interpreter differences, 194
described, 510
WINDOW statement, 513
Work area defined, 28, 577
World coordinates
defined, 577
mapping to physical coordinates, 398
WINDOW, used with, 513
WRITE statement, 518
WRITE# statement, 519

/x option, qb command, 73

Documentation Feedback

Please help us make our documentation better by completing and returning this card. Your feedback will help us improve future products.

Name _____

Address _____

City _____ State _____ Zip _____

Product _____ Version no. _____

Operating system _____

Computer _____ Memory (K) _____

RAM disk ☐ yes ☐ no Hard disk ☐ yes ☐ no

Programming experience: _____ years; in this language: _____years

Please rate the documentation on a scale of 1 to 5.

	5	4	3	2	1	
Complete	☐	☐	☐	☐	☐	Incomplete
Accurate	☐	☐	☐	☐	☐	Inaccurate
Organized	☐	☐	☐	☐	☐	Disorganized
Usable	☐	☐	☐	☐	☐	Unusable
Readable	☐	☐	☐	☐	☐	Unreadable
Understandable	☐	☐	☐	☐	☐	Unintelligible
Attractive	☐	☐	☐	☐	☐	Unattractive
Excellent	☐	☐	☐	☐	☐	Poor

Was installation easy to understand? ☐ yes ☐ no

Did you work through the demonstration session? ☐ yes ☐ no

If yes, was it useful? ☐ yes ☐ no

Is information easy to find? ☐ yes ☐ no

Do you use your documentation mainly for:
☐ reference? ☐ learning to program?
☐ learning to operate the software?

Did you find factual errors in the documentation? ☐ yes ☐ no
If yes, please give manual name, page number, and description of the error.

Is more information needed on how to program? ☐ yes ☐ no
If yes, on what topics?

Is the documentation intimidating? ☐ yes ☐ no

Are the manuals:
☐ too long? ☐ too short? ☐ about the right length?

Other suggestions or comments? (Use back of card if necessary.)

Comments and suggestions become the property of Microsoft Corporation.

detach here

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 108 BELLEVUE, WA U.S.A.

POSTAGE WILL BE PAID BY ADDRESSEE

MICROSOFT®
16011 NE 36th Way
Box 97017
Redmond, WA 98073-9717

Attn: Manager, Languages User Education
Bldg. 2

tape here

MICROSOFT®

04/86

MICROSOFT®

Microsoft® QuickBASIC

Compiler

Get performance beyond BASICA.

- Compile BASIC Interpreter programs and execute them up to ten times faster.
- Develop stand-alone programs that can be distributed without source code.

Take advantage of a complete programming environment that lets you get your programs up and running quickly.

- Use the built-in editor to write programs, and then to locate and correct errors.
- Compile faster with Microsoft QuickBASIC's ability to compile entirely in memory.
- Trace execution at the source level.

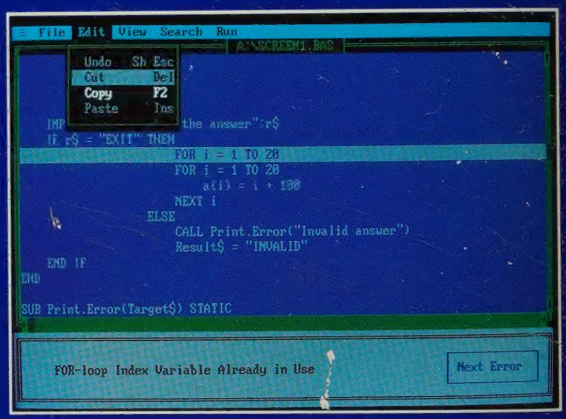
Create structured and modular programs that can be updated and maintained effortlessly.

- Control your program's flow with multi-line block IF/THEN/ELSE statements.
- Use alphanumeric labels — line numbers are optional — that make your programs easier to read and maintain.
- Create programs larger than 64K by compiling individual modules separately and linking them together.
- Create subprograms that use local and global variables.

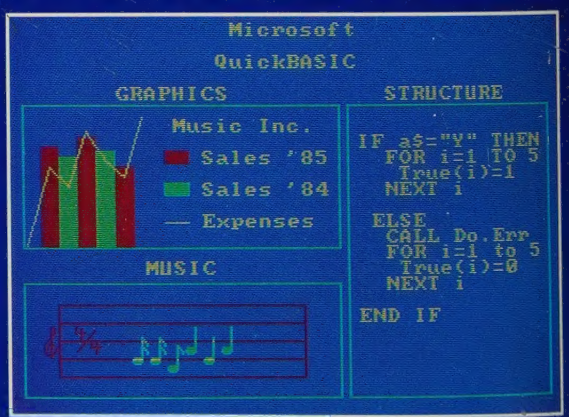
System Requirements

256K memory, DOS 2.0 or higher, one double-sided disk drive

Note: Use of the Microsoft® Mouse is optional.



Programming is easier with the built-in editor that searches for errors, letting you correct them and recompile without leaving the programming environment.



With Microsoft QuickBASIC you can create structured programs, as well as programs that use graphics, music, and sound effects.

Microsoft is a registered trademark of Microsoft Corporation.
© Copyright 1985, 1986. Microsoft Corporation.
All rights reserved. 0586 Part No. 007-114-009