

MICROSOFT

Microsoft QuickBASIC

Compiler



For IBM, Personal Computers
and Compatibles

Microsoft[®] QuickBASIC Compiler

Minimum (default) commands

bascom filename /o ;

/n6 filename ;

Microsoft Corporation

5	Linking and Running a Program	51
5.1	Linking a Program	53
5.2	Running the Linker	54
5.3	Linking with the BRUN10.LIB Library	59
5.4	Linking with the BCOM10.LIB Library	60
5.5	Running a Program	61
6	Using Metacommands	63
6.1	Metacommand Syntax	65
6.2	Source Listing Format: \$LIST	66
6.3	Object Code Listing Format: \$OCODE	66
6.4	Controlling the Listing Format	67
6.5	Processing Additional Source Files: \$INCLUDE	69
6.6	Dimensioned Array Allocation: \$STATIC and \$DYNAMIC	69
6.7	Changing Internal Module Names: \$MODULE	70
7	Creating Structured Programs	71
7.1	Structuring Programs	73
7.2	Creating Subprograms	74
7.3	Invoking BASIC Subprograms with CALL	75
7.4	Passing Parameters within a Module	78
7.5	Using a Library of Subprograms: Passing Parameters with COMMON	80
7.6	Preserving Subprogram Variable Values	81
7.7	Preserving Subprogram Array Values	82
7.8	Common Errors in Calling Subprograms	82
7.9	Using CALL or CHAIN	84

8	Using Assembly-Language Subroutines	85
8.1	Converting Interpreted Programs	87
8.2	Calling Assembly-Language Subroutines with CALL and CALLS	87
8.3	Coding Subroutines for the Compiler	91
8.4	The Run-Time Memory Maps	93
8.5	Segment Maps	96
8.6	The Run-Time Segment Maps	97
9	Microsoft QuickBASIC Compiler Reference	101
9.1	Compiler/Interpreter Differences	103
9.2	Reference Format	104
9.3	Reference Syntax Notation	105
9.4	Reference Input/Output Notation	106
	Appendixes	171
A	Summary of Commands	173
A.1	Microsoft QuickBASIC Metacommands	175
A.2	Compiler Switches	176
B	Questions Most Frequently Asked	177
C	Listing File Formats	181
C.1	Source Listing	183
C.2	Disassembly Listing	184
C.3	Linker Listing	185

D Using the Input Line Editor 187

E Microsoft QuickBASIC
Reserved Words 189

F Error Messages 191

F.1 Invocation Errors 193
F.2 Compile-Time Errors 194
F.3 Run-Time Errors 200
F.4 Microsoft LINK Errors 206

Glossary 209

Index 213

Figures

Figure 8.1	The Stack after CALL Executes	89
Figure 8.2	Run-Time Module Environment	94
Figure 8.3	Run-Time Library Environment	95

Tables

Table 1.1	Locating Information	4
Table 2.1	Disk Contents	12
Table 3.1	Error-Handling Exceptions	37
Table 6.1	Listing Format Commands	67
Table 8.1	Run-Time Segment Map with Run-Time Module	96
Table 8.2	Run-Time Segment Map without the Run-Time Module	97
Table 9.1	Functions and Statements in This Reference	103
Table 9.2	Colors in Medium Resolution	126
Table D.1	Input Line Editor Commands	187

Chapter 1

Introduction

- 1.1 Overview of Microsoft QuickBASIC 3
- 1.2 Using This Manual 4
- 1.3 Notational Conventions 6

1.1 Overview of Microsoft QuickBASIC

This manual documents differences between the Microsoft® QuickBASIC Compiler and the Microsoft BASIC 2.0 Interpreter. The interpreter is described in the IBM® Personal Computer BASICA reference manual and the *COMPAQ® Portable Computer BASIC Reference Guide*.

The Microsoft QuickBASIC Compiler is intended for users who are familiar with the Microsoft BASIC Interpreter, but want the size and speed advantages of compiled programs.

The following are features of the Microsoft QuickBASIC Compiler:

- **Flexible array dimensioning.** Dynamic arrays allow you to use variables to dimension the array. This has two advantages:
 1. Compatibility with a greater number of existing BASIC programs is provided.
 2. There is more efficient use of memory.
- **Support for 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 improves program readability, and can provide simpler debugging.
- **Enhanced graphics.** These are available through the Microsoft QuickBASIC Compiler graphics feature set.
- **Ability to trap many kinds of events.** The compiler's error-trapping features are explained in Chapter 3 of this manual.

The Microsoft QuickBASIC Compiler also features a run-time module that contains the run-time environment. This feature allows you to develop a system of related programs that will use the same run-time environment, without requiring that the run-time environment be saved as part of each executable program file. The run-time module is loaded when program execution begins; later execution of chained programs does not require reloading. For a system of four programs, this can save a substantial 100K of disk space.

1.2 Using This Manual

This manual assumes that you are familiar with the Microsoft BASIC language and with the MS-DOS® operating system, and that you are able to create and save a BASIC source file on your system. If you have questions about the BASIC language, refer to the BASIC reference in Chapter 9 of this manual, and the BASIC reference manual you received when you bought your computer. The Microsoft QuickBASIC reference in Chapter 9 includes only language features that are new or different from those of the Microsoft BASIC Interpreter.

Table 1.1 will help you find what you need to know to start using the Microsoft QuickBASIC Compiler.

Table 1.1

Locating Information

For This Information	See
How to install the compiler and linker software	Chapter 2, "Getting Started"
How to set environment variables for use with the compiler	Chapter 2, "Getting Started"
What changes you need to make to your interpreted programs before you can compile them	Chapter 3, "Developing a Program"
How to use the new language features of Microsoft QuickBASIC in your programs	Chapter 3, "Developing a Program"
How to use alphanumeric labels	Chapter 3, "Developing a Program"
How to run the compiler	Chapter 4, "Compiling a Program"
How to link and run a BASIC program	Chapter 5, "Linking and Running a Program"
What metacommands are and how to use them	Chapter 6, "Using Metacommands"
How to create modular programs	Chapter 7, "Creating Structured Programs"

Table 1.1 (continued)

For This Information	See
How to incorporate assembly-language routines into your BASIC programs	Chapter 8, "Using Assembly-Language Routines"
How to use the enhanced graphics features of Microsoft QuickBASIC	Chapter 9, "Microsoft QuickBASIC Compiler Reference"
A summary of all the Microsoft QuickBASIC Compiler switches	Appendix A, "Summary of Commands"
A summary of all the Microsoft QuickBASIC Compiler metacomands	Appendix A, "Summary of Commands"
A list of input line editor commands	Appendix D, "Using the Input Line Editor"
A list of Microsoft BASIC reserved words	Appendix E, "Microsoft QuickBASIC Reserved Words"
Explanations of error messages given by the compiler, the linker, and the run-time system	Appendix F, "Error Messages"
An explanation of the terms used in this manual	The glossary

Once you become more familiar with the Microsoft QuickBASIC Compiler, you may want to refer to Appendix B, "Questions Most Frequently Asked," and Appendix C, "Listing File Formats."

1.3 Notational Conventions

The following syntax notation is used in this manual:

Command names Bold typeface within the text indicates command names, as in the following sentence:

It is necessary to specify the library name on the link command line.

Bold type is also used occasionally in the text for emphasis.

Italics

Italics indicate places where specific terms, supplied by the user, will appear. For example, in *bascom filename*

filename is italicized to indicate that this is a general form, and that a specific filename will be substituted for *filename* in an actual command.

Italics are also used occasionally in the text for emphasis.

Screen text

This special typeface is used to look similar to the text on the screen or in a program. It used for sample programs, screen output, and command lines, as in the following:

```
10 print "This is a sample program line."
```

Input text

This special typeface is used to indicate input you enter in response to a prompt. In the following example, "sample" is entered in response to the "Source filename" prompt:

```
Source filename [.BAS]: sample
```

CAPITALS

Capital letters are used in the text to indicate Microsoft BASIC keywords. This is a convention for the documentation only; you don't have to enter these words in capital letters.

[Brackets]

Square brackets indicate that the enclosed entry is optional. For example, in

CALL *name* [*argument-list*]

the brackets around *argument-list* indicate you are not required to supply this information when you use the CALL statement.

{Braces}

Braces indicate that you have a choice of two or more entries. At least one of the entries enclosed in braces must be chosen unless the entries are also enclosed in square brackets. For example, in the statement

GOTO {*linenumber*{*linelabel*}}

you must specify either a line number or a line label as the object of the statement.

|

Vertical bars separate the choices within braces. At least one of the entries separated by bars must be chosen unless the entries are also enclosed in square brackets. For example, in the statement

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

the argument (enclosed in square brackets) is optional, but if you use an argument it must be either a line number or a line label.

Ellipses ...

Ellipses indicate that an entry may be repeated as many times as needed or desired. For example, in the statement

STATIC *variable* [,*variable*...]

the ellipses indicate that two or more *variables* are allowed.

Vertical ellipses are also used in program examples to indicate that a portion of the program is omitted. For instance, two statements are shown in the following example. The ellipses between the statements indicate that intervening program lines occur but are not shown:

```
sub prog2  
.  
.  
.  
end sub
```


Chapter 2

Getting Started

- 2.1 Backing Up Your Disks 11
- 2.2 Disk Contents 12
- 2.3 Setting Up the Environment 12
- 2.4 Setting Up Your Compiler:
Two Floppy-Disk Drives 13
- 2.5 Setting Up the Compiler:
Using a Hard Disk 14
- 2.6 Practice Session 15

This chapter tells you what you need to know to set up your compiling and linking environment to create programs quickly and efficiently. Section 2.4 explains how to set up the compiler and linker on a system with two floppy-disk drives; Section 2.5 explains setting up your compiler and linker on a hard-disk system. A practice session at the end of this chapter gives you the opportunity to test your setup and become familiar with the operation of the compiler.

If you have never used a compiler before, you may want to read Section 3.1, "Compilation versus Interpretation," first. Before you begin using the Microsoft QuickBASIC Compiler, you should follow the preliminary procedures described in this chapter. The preliminary procedures include the following:

- Copying (or "backing up") your system disks
- Organizing your compiler and linker software
- Setting up your MS-DOS environment to work with the compiler

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

2.1 Backing Up Your Disks

The first thing you should do when you have unwrapped your package is make a working copy of the compiler disk using the MS-DOS DISKCOPY utility. Put the original disk in a safe place and use the copy from now on. If the copy is ever damaged or destroyed, you can make new copies from the original disk.

2.2 Disk Contents

Your Microsoft QuickBASIC Compiler disk contains all the files listed in Table 2.1.

Table 2.1
Disk Contents

File	Description
BASCOM.EXE	The Microsoft QuickBASIC Compiler
BRUN10.EXE	The Microsoft QuickBASIC run-time module
BRUN10.LIB	The Microsoft QuickBASIC run-time module library
BCOM10.LIB	The Microsoft QuickBASIC alternate run-time library
README.DOC	List of recent software improvements; not always present

2.3 Setting Up the Environment

Before you can compile and link programs you must make sure that the compiler and linker can locate all the files they need to process your program. The required files are as follows:

File Type	Description
Executable	These are the files that are executed as your program is processed. The names of these files are BASCOM.EXE, LINK.EXE, and BRUN10.EXE.
Library	At link time, the linker LINK.EXE attempts to find the library file required by the object file and link it to your program. The names of the library files are BRUN10.LIB and BCOM10.LIB.

When you start the compiler or linker, it checks to see if you have defined certain "standard places" to search for the necessary files. You can define these places by using "environment variables." Environment variables are defined at the MS-DOS command level using the PATH and SET commands. (Environment variables can also be defined in the AUTOEXEC.BAT file.) They are called environment variables because they are effective throughout the environment in which a program is executed.

The PATH variable tells the compiler where to look for executable files, and the LIB environment variable tells the linker where to look for any libraries it needs.

Use the PATH command to define the PATH variable. The command has the following form:

```
PATH pathname [pathname; ...]
```

The PATH variable can contain more than one pathname. Pathnames must be separated by semicolons (;). For example, the following command tells 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 on Drive A:

```
path A:\bin:A:\src
set lib=A:\lib
```

Files that are linked with BRUN10.LIB require the presence of the run-time module file BRUN10.EXE when the program is run.

2.4 Setting Up Your Compiler: Two Floppy-Disk Drives

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 one drive contains the linker and the program you are working on. Because the compiler, linker, and libraries will not fit on one disk, it is easiest to copy all the files you need to compile your program onto one disk, and all the files you need to link and run your program onto another disk.

To set up your system you will need the following:

1. Two blank, formatted disks. One disk must be a "system" disk; that is, a disk formatted with the /S switch.
2. The backup disk of compiler software.
3. A computer with two floppy-disk drives.
4. An MS-DOS system disk.
5. A copy of LINK.EXE.
6. A text editor or Microsoft BASIC interpreter.

A sample setup is given below:

Disk 1: COMMAND.COM
BASCOM.EXE
BRUNIO.EXE
BCCMIO.LIB
BRUNIO.LIB

Disk 2: LINK.EXE
Text editor or BASIC Interpreter
Source program files

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:

```
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, listing file, map file, and executable program file) are written onto Disk 2.

2.5 Setting Up the Compiler: Using a Hard Disk

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. Refer to your MS-DOS documentation if you are unfamiliar with creating directories.

To set up your system you will need the following:

1. The backup compiler disks
2. A computer with a hard disk
3. A copy of Microsoft LINK (if it is not already on the hard disk)

The following is a sample hard-disk setup:

```
\BIN\BASCOM.EXE
\BIN\BRUN10.EXE
\BIN\LINK.EXE
```

```
\LIB\BRUN10.LIB
\LIB\BCOM10.LIB
```

Using the previous sample setup, your environment variables can be given the values shown below:

```
path C:\bin
set lib=C:\lib
```

2.6 Practice Session

This section shows you the steps involved in compiling and linking a program using the Microsoft QuickBASIC Compiler. By following these steps you can produce and run an executable program file.

The practice session assumes you are using the sample disk setup and environment described in Sections 2.3, 2.4, and 2.5. The first thing you should do is verify that the environment is set up correctly. Type

```
set
```

and press the Return key. The SET command lists all environment variables and their current settings. Make sure PATH and LIB are set appropriately for your system, as shown below:

Hard-Disk Settings

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

Floppy-Disk Settings

```
PATH A:\
LIB=A:\
```

Before you can begin compiling and linking, you must create a BASIC source file. Create your source file on the disk in Drive B with any available text editor, or with the BASIC interpreter. (If you create your program with the interpreter, be sure to save the program with the "A" option.) It is recommended that you write a very simple program, such as the one in the following example, so that you can concentrate on the compiling and linking procedures instead of debugging the program.

Enter the following program and save it in a file called SAMPLE.BAS:

```
for A = 1 to 10
print "This is a sample program."
next A
```

Once you have entered the program you are ready to begin processing your source file using the following 12 steps:

1. You are ready to begin compiling. Enter the command
bascom

The compiler will display prompts on the screen to guide you through the compiling process.

2. The first message to appear on your screen is

```
Microsoft Quick BASIC Compiler Version 1.0
(C) Copyright Microsoft Corporation 1984, 1985
Source filename [.BAS]:
```

Following the "Source filename" prompt, specify the name of the file to be compiled. If you don't include the filename extension, the extension ".BAS" is assumed.

In response to this prompt, type
sample

3. The next prompt is

```
Object filename [.OBJ]:
```

This prompt allows you to supply a name for the object file. Instead of entering a filename, just press Return. This tells the compiler to use the default response for the prompt, which is to name the file SAMPLE.OBJ.

4. The next prompt is

```
Source listing [NUL.LST]:
```


4. The next prompt is

Source listing [NUL.LST]:

This prompt lets you create a listing of your source file, any error messages that are produced during compilation, and the machine instructions that correspond to the BASIC instructions in your program. In response to this prompt, type

sample

The compiler appends the default extension .LST and creates a listing named SAMPLE.LST.

5. Now the compiler begins to compile your program. If there are errors in your program, the compiler will display the error messages on your screen. For the demonstration program, the compiler should display no error messages. When the compiler has finished, it displays a message that looks something like this one:

23533 Bytes Available
17003 Bytes Free

0 Warning Error(s)
0 Severe Error(s)

The number of bytes available and bytes free varies with the specifics of your particular system.

You now have an object file named SAMPLE.OBJ and a listing file named SAMPLE.LST in your directory.

6. The next step is to create an executable program by linking routines in the run-time library BRUN10.LIB to the object file SAMPLE.OBJ. Start the linker by typing

link

The LINK command invokes the linker. You will see the following message on your screen:

Microsoft 8086 Object Linker

The message is followed by a version number and copyright notice.

7. The first prompt is

Object Modules [.OBJ]:

Enter the name of the object module you just created:

sample

Microsoft LINK automatically appends the .OBJ extension to the name you type.

8. The next prompt is

Run File [SAMPLE.EXE]:

This prompt lets you name the executable program file. Press Return in response to this prompt. The linker uses the default name shown in brackets for the executable file if you don't supply a different name.

9. The next prompt is

List File [NUL.MAP]:

If you give a filename following this prompt, the linker creates a map file listing all the external symbols in your program and their locations. Type the response

sample

This response tells the linker to create a listing file named SAMPLE.MAP. The .MAP extension is used because you did not supply your own extension.

10. The final prompt is

Libraries [.LIB]:

Several options are available at this point, but for this demonstration run, we will use the default library BRUN10.LIB. The LIB environment variable tells the linker where to find the library. Just press Return in response to this prompt.

11. Microsoft LINK now proceeds to link your file. If any errors are found, they are displayed on your screen. When the system prompt reappears, the linker has finished processing your file. You now have

an executable file named `SAMPLE.EXE` in your directory, plus a linker listing named `SAMPLE.MAP`.

You may want to examine the listing files `SAMPLE.LST` and `SAMPLE.MAP` to familiarize yourself with their formats. These files are especially useful for debugging programs. The listing file formats are explained in Appendix C, "Listing File Formats." However, the listing and map files are not required to run the program, so you can delete them.

12. The final step is running the program. Type

```
sample
```

and press Return.

If the program you are using is the `SAMPLE` program listed above, the following message will appear on your screen when you run the program:

```
This is a sample program.  
This is a sample program.  
This is a sample program.  
This is a sample program.  
This is a sample program.  
This is a sample program.  
This is a sample program.  
This is a sample program.  
This is a sample program.
```


Chapter 3

Developing a Program

3.1	Compilation versus Interpretation	23
3.2	Converting Interpreted Programs to Compiled Programs	24
3.2.1	Source File Format	25
3.2.2	Commands Not Accepted by the Compiler	25
3.2.3	Statements Requiring Modification	25
3.2.4	Operational Differences	26
3.3	Using New Language Features	26
3.3.1	New BASIC Statements and Functions	26
3.3.2	Enhancements to Existing BASIC	27
3.3.3	Creating Lines Longer than 254 Characters	28
3.3.4	Line Numbers and Alphanumeric Labels	28
3.3.5	Using Line Labels	29
3.3.6	Dynamic and Static Arrays	30
3.3.7	Referencing Arrays in COMMON Statements	32
3.3.8	File Locking	33
3.3.9	Using Multiline Functions	34
3.3.10	Error Handling	37
3.4	Debugging with the Interpreter	38

This chapter explains the differences between interpretation and compilation, and lists the differences between interpreted Microsoft BASIC and Microsoft QuickBASIC. In addition, it explains how to use features added to the compiled BASIC to create more efficient and useful programs. Even if you are an experienced BASIC programmer, before you begin to use the compiler, you should read this chapter to learn about the new features and to find out if you need to change source files created with the interpreter.

3.1 Compilation versus Interpretation

A microprocessor can execute only its own machine language instructions; it cannot directly execute BASIC statements. Before a program can be executed, the statements in a BASIC program must be translated to the machine language of the microprocessor. Compilers and interpreters are two types of programs that perform this translation. This discussion explains the difference between the two translation schemes, and explains why and when you should use the Microsoft QuickBASIC Compiler.

The Microsoft BASIC Interpreter analyzes each Microsoft BASIC statement, checks it for errors, then performs the Microsoft BASIC process requested. Statements that are executed repeatedly (inside a FOR...NEXT loop, for example) are translated before each execution of the statement. A compiler translates the entire source program into machine code *before* you run the compiled program. It doesn't translate your Microsoft BASIC source file during the execution of your program. The compiler then puts the translation into a new file called an object file.

The interpreter cannot look ahead in the program to see what's coming up, or analyze the whole program and perform actions related to such an analysis. The compiler, however, can examine the entire program and modify the file it produces to create a more efficient executable program.

Interpreted Microsoft BASIC programs are stored in memory as a list of numbered lines. When the interpreter encounters branches such as GOTOs and GOSUBs it examines the line numbers in the list, starting with the first, until the referenced line is found. In contrast, the compiler assigns absolute memory addresses to variables and target lines of GOTOs and GOSUBs. When the computer encounters branching statements, it goes directly to the memory address associated with that line and executes the statement. The computer doesn't have to search lists of variables or line numbers during execution of your program, and your program executes more quickly. Compiled programs occupy less space in memory, and execute faster than interpreted programs.

The interpreter also maintains a list of all variables. When a reference to a variable occurs in a Microsoft BASIC statement, the interpreter searches through this list until the variable is found. The compiler, on the other hand, assigns absolute memory addresses to variables. When the compiler encounters a variable, it goes directly to the absolute memory address associated with that variable.

Interpreted programs require only one file for execution; creating an executable program with the compiler requires two files in addition to the original source file: an object file and an "executable file." The object file must be linked to a "run-time library." The run-time library is a file that contains routines that perform services such as printing characters on the screen and accepting input from the keyboard. During the linking process, these routines are combined with the BASIC compiled object file to create an executable file. (Assembly-language subroutine object files can also be included at this time.) To run a compiled program, execute it like any other MS-DOS program.

The Microsoft QuickBASIC Compiler is an *optimizing* compiler. Optimization produces executable programs that are smaller and execute more quickly than programs that are not optimized. The compiler optimizes by reordering expressions and by eliminating redundancy, such as multiplication by 1.

The execution time you save with the compiler depends on the content of your program. In most cases, the computer executes compiled Microsoft QuickBASIC programs 3 to 10 times faster than the interpreter. If you make maximum use of integer variables, program execution can be up to 30 times faster. Programs with many input or output operations will not benefit as much because of speed limitations of file storage hardware.

3.2 Converting Interpreted Programs to Compiled Programs

The BASIC understood by the compiler is slightly different from interpreted BASIC. If you wish to compile programs originally developed for the interpreter, you may need to modify your source code. The following sections describe these language differences and some rules you should know about when preparing source files for the compiler.

3.2.1 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 Microsoft BASIC Interpreter, it must be saved with the /A option; otherwise, the interpreter encodes the text of your Microsoft BASIC program in a special format. The compiler cannot read this format. If this happens, reload the BASIC interpreter and resave the file in ASCII format, using the /A option.

3.2.2 Commands Not Accepted by the Compiler

You cannot use the following commands in a compiled program:

AUTO	DELETE	MERGE
BLOAD	EDIT	NEW
BSAVE	LIST	RENUM
CONT	LLIST	SAVE
DEFUSR	LOAD	USR

3.2.3 Statements Requiring Modification

If your interpreter program contains any of the following statements, you will probably need to modify the source code before you can compile the program. Refer to Chapter 9, "Microsoft QuickBASIC Compiler Reference," for details.

CALL
CALLS
CHAIN
DEF *type*
DRAW
PLAY
RESUME
RUN

3.2.4 Operational Differences

If your program contains any of the following commands, you must compile your program with the indicated switch. Using switches is explained in Section 4.2, "Using the Compiler Switches."

Command	Switch
ON ERROR GOTO	/E
ON <i>event</i> GOSUB	/E
KEY ON/OFF/STOP	/V or /W
PEN ON/OFF/STOP	/V or /W
PLAY ON/OFF/STOP	/V or /W
STRIG ON/OFF/STOP	/V or /W
TIMER ON/OFF/STOP	/V or /W
RESUME	/X
RESUME NEXT	/X
RESUME 0	/X
STRIG ON/OFF/STOP	/V or /W
TIMER ON/OFF/STOP	/V or /W
TRON/TROFF	/D

3.3 Using New Language Features

Microsoft QuickBASIC has many new features. The following sections discuss how to use these features.

3.3.1 New BASIC Statements and Functions

The following statements and functions are new to Microsoft QuickBASIC. Using these new features is explained in this chapter and in Chapter 9, "Microsoft QuickBASIC Compiler Reference."

Statement/Function	Effect
COMMAND\$	Returns the command line used to invoke the program
LBOUND/UBOUND	Returns the lower/upper bound of an array
LOCK...UNLOCK	Controls access by other processes to all or part of an opened file
REDIM	Changes the space allocated to dynamic arrays
SHARED	Within a single module, gives subprograms access to main program variables without requiring the variables to be passed as parameters
STATIC	Designates variables and arrays as local to a subprogram or function definition, and preserves subprogram variable values when the subprogram is exited and then reentered
SUB...END SUB/EXIT SUB	Marks beginning and end of a subprogram

3.3.2 Enhancements to Existing BASIC

Increased functionality has been added to the following functions and statements. See Chapter 9, "Microsoft QuickBASIC Compiler Reference," for details.

Statement/Function	Enhancement
COMMON	Allows data to be shared between program modules
DEF FN	Supports both single and multiline functions
END	Terminates program execution, closes all files, and returns control to the operating system
ERASE	Deallocates dynamic-array elements
FRE	Returns the size of the next free block in string space

Line number line labels can be any combination of digits, from 0 to 65529. The following are valid line numbers:

```
1
200
65000
```

Although line number 0 is allowed, error trapping interprets the presence of line number 0 to mean that error trapping is disabled. The following statement will not branch to line 0 if an error occurs:

```
on error goto 0
```

Also, RESUME 0 will not resume on line 0 if an error occurs. *Alphanumeric line labels* 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 number or label on a line.

3.3.5 Using Line Labels

Microsoft 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 Microsoft 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 goto IncomeData1
```

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

3.3.6 Dynamic and Static Arrays

The Microsoft 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 and array elements can be accessed faster, but dynamic arrays are more flexible.

The metacommands \$STATIC and \$DYNAMIC tell the compiler how to allocate memory for arrays that appear in DIM statements with integer-constant subscripts. (Metacommands are special instructions to the compiler that are placed in the source code.) Using the \$STATIC and \$DYNAMIC metacommands is explained in Chapter 6.

Arrays are static by default. If the \$DYNAMIC metacommand is present, all subsequent array declarations in a DIM statement are treated as dynamic arrays. If the \$STATIC metacommand is present, any array declaration with constant-integer bounds, such as `ARRAY1 (2, 12)`, is statically allocated.

Arrays are dimensioned with the DIM statement, and dynamic arrays are redimensioned with the REDIM statement. Dynamic-array memory space is deallocated with the ERASE statement. The syntax of the DIM, REDIM, and ERASE statements is explained in Chapter 9, "Microsoft QuickBASIC Compiler Reference." The rest of this section explains how to use these statements.

The following rules apply when using the DIM statement:

1. Statically allocated arrays can only be dimensioned once. Dynamic arrays can be redimensioned using the REDIM statement or by using an ERASE array DIM array sequence.
2. Because a DIM statement that allocates a dynamic array is considered an executable statement, it 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 unallocated, otherwise a "Redimensioned Array" error message appears. Dynamic arrays are deallocated with the ERASE 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. Static arrays cannot be reallocated.

The ERASE statement operates differently on static and dynamic arrays. On a static array, ERASE sets all elements to zero, or to null strings. On a dynamic array, ERASE deallocates the array elements. The dynamic array must be redimensioned with a DIM statement before it can be referenced again.

Examples

The following example allocates space for a static two-dimensional array:

```
rem $static
dim A(10,20)
```

This 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 erases 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)
```

Warning

Bounds checking is performed on static arrays only when the program is compiled with the /D (debug) switch. Unallocated arrays (arrays that have not been dimensioned with the DIM statement) are considered static even when the %DYNAMIC metacommand is in effect. Severe run-time errors may result when the bounds of an array are exceeded in programs compiled without the /D switch. The upper bound of an unallocated static array defaults to 10.

3.3.7 Referencing Arrays in COMMON Statements

COMMON statements reference arrays in two forms. The following form treats the array as a static array:

COMMON *variable*()

The array must be declared in a DIM statement before it is used in the above statement.

Example

```
dim p (10,20)
common p()
```

The following statement treats the array as a dynamic array:

COMMON *variable*(*integer*)

Integer is the number of dimensions in the array. The array elements are not allocated at compile time; only space for the dynamic-array descriptor is reserved in the COMMON area. The array must be allocated with a DIM or REDIM statement that refers to it. If the array is allocated in a chaining program, its element values are passed to the program that is the object of the CHAIN statement. A DIM statement in the chained-to program produces an error. A REDIM statement in the chained-to program erases the passed values.

Example

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,u)      'first allocation
```

The chained-to program reallocates array X, erasing its original contents:

```
common X(2)
redim X(o,p)    'reallocation
```

3.3.8 File Locking

The LOCK and UNLOCK statements restrict access by other processes to all or part of an opened file. This function is useful in a network environment where more than one process may be using a given file.

The LOCK statement restricts access to a range of records. It has the following form:

```
LOCK [#] filenum [, [{ record } [ start ] TO end ]]
```

where

<i>filenum</i>	is the file whose records are being locked.
<i>record</i>	is the number of the first record to be locked. LOCK with no arguments locks the entire file. If one <i>record</i> is specified, only that <i>record</i> is locked. If you specify a range of records and omit a starting record, (<i>start</i>) then all records from the first record to the end of the range (<i>end</i>) are locked.
<i>start TO end</i>	is an optional sequence designating the range of locked records.

All locked regions should be freed with the UNLOCK statement before the file is closed or the program terminates. Undefined results can occur when locks are not removed before closing.

The UNLOCK statement releases locks applied to an opened file. It has the following form:

```
UNLOCK [#] filename [, [record] [start] TO end]
```

The record-number range must match the record-number range given in the corresponding LOCK statement, or a "Permission Denied" error message is generated.

Examples

lock#1	'Locks the entire file
unlock#1	'Unlocks the entire file
lock#1,x	'Locks record x
unlock#1,x	'Unlocks record x
lock#2,to y	'Locks records 1 to y
unlock#2,to y	'Unlocks records 1 to y
lock#2,x to y	'Locks records x to y
unlock#2,x to y	'Unlocks records x to y

3.3.9 Using Multiline Functions

The Microsoft QuickBASIC Compiler adds to your programming capabilities by allowing you to define multiline functions.

A multiline-function definition consists of three parts:

- The DEF FN statement
- The function body
- The END DEF statement

The syntax for the DEF FN statement is as follows:

```
DEF FNname [(parameter-list,...)]
```

where

<i>name</i>	is any valid Microsoft BASIC identifier.
<i>parameter-list</i>	consists of variable names the function replaces when it is called.

Values in the body of a function are returned by assigning the result to a variable with the same name as the function and then exiting the function. The format for the function value assignment is as follows:

FNname = expression

Multiline functions are ended with an EXIT DEF or END DEF statement.

Example

The following function definition computes the total of the square of integers from 1 to *n*, where *n* is an argument passed to the function from the main program.

```
def fsumofsquares(n)
  if n <= 0 then fsumofsquares = -1 : exit def
  sum = 0
  for i = 1 to n
    sum = sum + i*i
  next i
  fsumofsquares = sum
end def
```

The statement

```
print fsumofsquares(3)
```

yields 14.

The following rules apply to Microsoft QuickBASIC multiline functions:

1. User-defined functions must be defined before any references are made to the function.
2. Recursive function definitions are not allowed, i.e., functions cannot call themselves.
3. User-defined function definitions may not appear inside DEF FN...END DEF, IF...THEN...ELSE, FOR...NEXT, or WHILE...WEND blocks. User-defined function definitions are also not permitted in SUB...END SUB blocks.

Because multiline functions are very powerful, they must be used with care. You should note the following warnings before you use multiline functions in your programs:

- 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 switches /D, /E, /V, /X, or /W, you will either get a "RETURN without GOSUB" error or unpredictable results.
- 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 Microsoft 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 : l = fna : print l + i + i
```

Nesting run-time features can also cause side effects when using multiline functions. In the following function, for example,

```
A$ = "+ . " : B$ = " . - "
def fnb
    print using B$;Z
end def
print using A$;fnb.B.C
```

B and C will be printed using the next field description from B\$ instead of from A\$.

3.3.10 Error Handling

The compiler has several switches that allow a program to use the advanced error-handling features of Microsoft QuickBASIC:

- The /D switch generates special code that can trap certain errors and will generate error messages during run time.
- The /E and /X switches handle ON ERROR and RESUME statements in your program.

Using command-line switches is explained in Section 4.2, "Using the Compiler Switches." The compiler supports the following error-handling statements and functions. They work as described in your BASIC interpreter reference manual, with the exceptions indicated in Table 3.1.

Table 3.1
Error-Handling Exceptions

Statement or Function	Exception
ERL function	Only reports line numbers
ERR function	None
ERROR statement	None
ON ERROR GOTO statement	Can refer to line numbers or labels
RESUME statements	Can refer to line numbers or labels

3.4 Debugging with the Interpreter

Following the requirements explained in this manual, you can use the Microsoft BASIC Interpreter to create and debug source programs that have numbered lines. (You cannot use the interpreter to debug programs that have unnumbered lines or alphanumeric line labels.)

Creating programs with the interpreter allows you to correct errors as they occur. The interpreter stops execution of a program when an error is encountered, tells you the line where the error was produced, and places you in edit mode. When the error is corrected, you can run the program again and again until all errors are corrected.

When the program is complete and error-free, you can use the compiler and linker to create an executable file that runs faster and occupies less space in memory.

Chapter 4

Compiling a Program

4.1	Running the Compiler	41
4.1.1	Filename Conventions	42
4.1.2	Special Filenames	42
4.1.3	Source Filename Prompt	43
4.1.4	Object Filename Prompt	43
4.1.5	Source Listing Prompt	44
4.1.6	Selecting Default Responses	45
4.1.7	Using the Command-Line Method	45
4.2	Using the Compiler Switches	46
4.2.1	Event Trapping: Checking Between LINES	46
4.2.2	Event Trapping: Checking Between STATEMENTS	47
4.2.3	Using Error-Handling Statements	47
4.2.4	Generating Debugging and Error-Handling Messages	47
4.2.5	Generating Disassembled Object Code	48
4.2.6	Preparing to Link with BCOM10.LIB	49
4.2.7	Writing Quoted Strings to Disk Instead of Memory	49
4.2.8	Storing Arrays in Row Order	50
4.3	Using the Console for Input/Output	50

After you create a Microsoft BASIC source program, your next step is compilation. This chapter covers important aspects of compilation, including the following:

- Invoking the compiler
- Using the compiler switches
- Using input from the console

4.1 Running the Compiler

The Microsoft QuickBASIC Compiler requires two kinds of input: a command to start the compiler and responses to command-line prompts. Start the compiler by typing

```
bascom
```

at the MS-DOS command level. The compiler prompts you for the input it needs by displaying the following three lines, one at a time. The compiler waits for you to respond to each prompt before printing the next one:

```
Source filename [.BAS]:  
Object filename [.OBJ]:  
Source listing [NUL.LST]:
```

The responses you can make to each prompt are explained in Sections 4.1.3 through 4.1.6.

Once you understand the Microsoft QuickBASIC prompts and operations, you can use the command-line method of running the compiler; this alternate method is described in Section 4.1.7. The command-line method lets you type all commands, options, and filenames on the line used to start the compiler.

Before you use the command-line method you should understand how the compiler works and what your responses to its prompts mean. It is recommended that you allow the compiler to prompt you for responses until you are comfortable with its commands and operations.

4.1.1 Filename Conventions

The Microsoft QuickBASIC Compiler uses the default file extensions ".BAS", ".OBJ", and ".LST" for the source, object, and listing files, respectively, when you do not supply extensions with your filenames. You can override the default extension for a particular prompt by specifying a different extension. To enter a filename that has no extension, type the name followed by a period. For example, typing "ABC." in response to a prompt tells the compiler that the given name (in this case, ABC) has no extension, while typing just "ABC" tells the compiler to use the default extension for that prompt.

4.1.2 Special Filenames

You can use the following MS-DOS device names as filenames with the Microsoft QuickBASIC Compiler. These device names allow you to direct files to your terminal or to a printer. Note that you cannot use these names for ordinary filenames.

Device	Function
AUX	Refers to the communications port for auxiliary devices such as a printer or disk drive.
CON	Refers to your screen (for output) and keyboard (for input).
LST or PRN	Refers to the printer device.
NUL	Specifies a nonexistent output file. Giving NUL as a filename means no file is created.
USER	Refers to your screen. Same as CON, except you cannot direct output to the screen at run time with USER but must use CON instead.

Even if you add device designations or filename extensions to these special filenames, they remain associated with the devices listed above. For example, A:CON.XX still refers to the screen and is not the name of a file.

4.1.3 Source Filename Prompt

Following the "Source filename" prompt, give the name of the source file you want to compile. If you do not supply an extension, the compiler automatically looks for a file with the extension ".BAS".

Pathnames are allowed with the source filename. You can give the pathname of a source file in another directory or on another disk.

You can compile only one file at a time, so only one response to this prompt is allowed. There is no default response; the compiler displays an error message if you do not give a source filename.

Example

Both of the following responses compile the source file MYPROG.BAS:

```
Source filename [.BAS]: myprog.bas
```

```
Source filename [.BAS]: myprog
```

The following example compiles the file "MYPROG". Note that the filename has no extension.

```
Source filename [.BAS]: myprog.
```

4.1.4 Object Filename Prompt

Following the "Object filename" prompt you can give a name for the object file produced when the source file is compiled. By convention, object files have the basename of the source file and the extension ".OBJ", but you can choose any name you like.

Pathnames are allowed following this prompt. You can create an object file in another directory or on another disk by including the directory and drive specification in the object filename.

The default name supplied for the object file is the basename of the source file with an ".OBJ" extension. By default, the object file is created in the current working directory.

Example

The following response tells the compiler to create the object file MYPROG.OBJ:

```
Object filename: [MYPROG.OBJ]: myprog
```

The following example creates the object file MYPROG.OBJ in the directory OBJECTS on Drive C:

```
Object filename: [MYPROG.OBJ]: c:\objects\myprog
```

4.1.5 Source Listing Prompt

Following the "Source listing" prompt you can tell the compiler to create a source listing file for the compiled file. The source listing contains the memory address of each line in your source file, the text of the source file, its size, and any error messages produced during compilation.

If you give a filename following this prompt, the compiler creates a source listing using the filename you supply. By convention, these listings are given the extension ".LST", but you can choose any name you like. If you supply a name without an extension, the compiler automatically appends .LST to the listing filename.

Pathnames are allowed in response to this prompt. You can create a source listing file in another directory or on another disk by including the directory and drive specification in the source listing filename.

When you do not give a filename, the default is the special name NUL.LST, which tells the compiler *not* to create a listing.

There is a sample source listing in Appendix C, "Listing File Formats."

Example

The following example tells the compiler to create the source listing SAMPLE.LST:

```
Source listing [.LST]: sample
```

4.1.6 Selecting Default Responses

To select the default response to the current prompt, just press Return. The next prompt will appear.

There is no default for the first prompt, "Source filename". To select default responses for the remaining prompts, type a single semicolon (;) after the "Object filename" prompt. Once the semicolon is entered, you cannot respond to any of the remaining compiler prompts (they won't be printed on the screen). Use this option to save time when the default responses are acceptable.

The default for the "Object filename" prompt is the basename of the source file with an ".OBJ" extension. The default for the "Source listing" prompt is the special name NUL.LST, which tells the compiler not to create a source listing.

Example

The following response compiles the file MYPROG.BAS. The object file is named MYPROG.OBJ, and no listing file is created.

```
Source filename [.BAS]: myprog;
```

4.1.7 Using the Command-Line Method

Once you understand how the Microsoft QuickBASIC Compiler prompts and responses work, you can use the command-line method of running the compiler. With this method you type all the filenames on the same line as the command you use to start the compiler. This command line has the following form:

```
bascom source-file [, [object-file] [, [listing-file] ] ] [/switch...] [;]
```

You can include spaces before or after filenames, but not within them. Switches can follow any filename, but must appear before the semicolon. Switches are explained in Section 4.2, "Using the Compiler Switches."

4.1.6 Selecting Default Responses

To select the default response to the current prompt, just press Return. The next prompt will appear.

There is no default for the first prompt, "Source filename". To select default responses for the remaining prompts, type a single semicolon (;) after the "Object filename" prompt. Once the semicolon is entered, you cannot respond to any of the remaining compiler prompts (they won't be printed on the screen). Use this option to save time when the default responses are acceptable.

The default for the "Object filename" prompt is the basename of the source file with an ".OBJ" extension. The default for the "Source listing" prompt is the special name NUL.LST, which tells the compiler not to create a source listing.

Example

The following response compiles the file MYPROG.BAS. The object file is named MYPROG.OBJ, and no listing file is created.

```
Source filename [.BAS]: myprog;
```

4.1.7 Using the Command-Line Method

Once you understand how the Microsoft QuickBASIC Compiler prompts and responses work, you can use the command-line method of running the compiler. With this method you type all the filenames on the same line as the command you use to start the compiler. This command line has the following form:

```
bascom source-file [, [object-file] [, [listing-file] ] ] [/switch...] [;]
```

You can include spaces before or after filenames, but not within them. Switches can follow any filename, but must appear before the semicolon. Switches are explained in Section 4.2, "Using the Compiler Switches."

To select the default responses, leave the *object-file* and *listing-file* fields blank. For example, the line

```
bascom myprog. ,myprog.l ;
```

compiles the file MYPROG.BAS, and produces the object file MYPROG.OBJ and the listing file MYPROG.L.

The semicolon (;) has the same effect on the command line as it does on the compiler prompts. When the compiler encounters a semicolon on the command line, it uses the default responses for the remaining prompts. For example, the command

```
bascom myprog;
```

compiles the file MYPROG.BAS and produces the object file MYPROG.OBJ.

4.2 Using the Compiler Switches

You can direct the Microsoft QuickBASIC Compiler to perform additional or alternate functions by adding switches to the command line. Switches signal special instructions the compiler uses during compilation. The switch tells the compiler to "switch on" a special function or to alter a normal compiler function. You can use more than one switch, but each must begin with a slash (/).

If you are using the prompting method, you can specify switches after you give the filename response to the "Source filename" prompt, before you press Return. If you are using the command-line method and are using any default responses, you can place the switches before the semicolon (;) at the end of the line.

The following sections describe each switch. There is a summary of the switches in Appendix A, "Summary of Commands."

4.2.1 Event Trapping: Checking Between LINES

The /W switch enables event trapping for communications (COM), lightpen (PEN), joystick (STRIG), the 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.

4.2.2 Event Trapping: Checking Between STATEMENTS

Like the /W switch, the /V switch enables event trapping, but it causes checking after each statement. If there are several statements on a line, the /V switch checks between each statement to see whether an event has occurred.

4.2.3 Using Error-Handling Statements

The /E switch tells the compiler that the program contains an ON ERROR GOTO...RESUME *line number* construction. To trap errors properly, the compiler must generate extra code for the GOSUB and RETURN statements. The compiler also generates a line-number address table (one entry per line number) in the binary file, so each run-time error message can include the number of the line in which the error occurs. Do not use this switch unless the program contains an ON ERROR GOTO statement.

The /X switch tells the compiler that the program contains one or more RESUME, RESUME NEXT, or RESUME 0 statements. If a RESUME statement other than RESUME *line number* is used with the ON ERROR GOTO statement, use the /X instead of the /E switch.

4.2.4 Generating Debugging and Error-Handling Messages

The /D switch forces the compiler to include code that generates debugging and error-handling messages at run time. Use of /D allows use of TRON and TROFF in the compiled file. If /D is not set, TRON and TROFF are ignored and a warning is issued.

When the /D switch is used, 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 numbers

The generated binary code includes line numbers so that the run-time error listing can indicate on which line an error occurs.

- RETURN statements

Each RETURN statement is checked for a prior GOSUB statement.

- Control-Break

After each line the compiler checks to see if the user has pressed Control-Break. If Control-Break is pressed, the following message appears and program execution stops:

Break

STOP in line *n* of module *m* at address *segment:offset*

If you don't use the /D switch, array bound errors, RETURN without GOSUB errors, and arithmetic overflow errors do not generate error messages at compile time or run time. The results may be unpredictable.

If a program is not compiled with the /D switch, a user cannot exit the program by pressing Control-Break, except when entering data in response to an INPUT statement prompt. In this case, to exit the program you must restart your computer.

4.2.5 Generating Disassembled Object Code

The /A switch generates a listing of the disassembled object code for each source line and shows the assembly-language code generated by the compiler. Disassembly listing files are typically used to look for inefficient instruction sequences that can be improved by changing the source file.

The disassembly listing is placed in the source listing file, so if you use this switch you must also request a source listing file.

Using this switch can greatly increase the length of a listing. For this reason it should be used judiciously. The actual code generated by the compiler is not affected by the use of this switch.

Example

The following command generates a source code listing (MYPROG.LST) that includes the disassembly code for MYPROG.BAS:

```
bascom myprog. myprog /A
```

The disassembly listing format is explained in Appendix C, "Listing File Formats."

4.2.6 Preparing to Link with BCOM10.LIB

The /O switch tells the compiler to substitute the BCOM10.LIB run-time library for BRUN10.LIB as the default run-time library searched by the linker. Any .EXE files created by linking to BCOM10.LIB do not need the run-time module on disk at run time. Using the /O switch makes the .EXE file larger.

The advantages and disadvantages of linking with BCOM10.LIB are discussed in Section 5.1, "Linking a Program."

4.2.7 Writing Quoted Strings to Disk Instead of Memory

The /S switch forces the compiler to write quoted strings that exceed four characters in length to an .OBJ file on disk as they are encountered, instead of retaining them in memory during the compilation of the program. If this switch is not set, and the program contains a large number of long quoted strings, the program may run out of memory at compile time.

Although the /S switch reduces the amount of memory used at compile time, it may *increase* the amount of memory needed in the run-time environment, since multiple instances of identical strings will exist in the program. Without /S, references to multiple identical strings are combined so that only one instance of the string is necessary in the final compiled program.

4.2.8 Storing 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 `/R` switch instructs the compiler to store arrays in row order, where the element `ARRAY(2,1)` would be followed by `ARRAY(2,2)`. This switch is useful if you are using assembly-language routines that store arrays in row order.

The Microsoft BASIC Interpreter stores and accesses arrays in column order.

4.3 Using the Console for Input/Output

You can tell the compiler that the source file will come from, or the listing file will go to, the screen. Do this by answering `USER` to the "Source filename" and "Source listing" prompts.

If you specify `USER` as the source file, you will enter the source code directly from the keyboard.

If `USER` is specified as the source listing, the listing will appear on the screen. Do not use both `USER` and `CON` devices in a single compilation process.

Chapter 5

Linking and Running a Program

5.1	Linking a Program	53
5.2	Running the Linker	54
5.2.1	Filename Conventions	54
5.2.2	Object Modules Prompt	55
5.2.3	Run File Prompt	55
5.2.4	List File Prompt	56
5.2.5	Libraries Prompt	56
5.2.6	Separating Entries	57
5.2.7	Extending Lines	57
5.2.8	Selecting Default Responses	57
5.2.9	Using a Command Line	58
5.3	Linking with the BRUN10.LIB Library	59
5.4	Linking with the BCOM10.LIB Library	60
5.5	Running a Program	61

Linking is a process that 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, 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.

5.1 Linking a Program

The two ways to link an object file are as follows:

1. With the BRUN10.LIB library
2. With the BCOM10.LIB library

Linking to BRUN10.LIB requires the presence of the module BRUN10.EXE at run time. This is the default; in order to use the second method you must compile your program with the /O switch.

When you link with BCOM10.LIB, selected routines are linked to the .OBJ file to create a single .EXE file that does *not* need the run-time module; any program linked with BCOM10.LIB does not require the presence of the run-time module. These programs are smaller in memory and slightly faster, but require more disk space than those linked with BRUN10.LIB. Also, in BASIC programs linked with BCOM10.LIB, the CHAIN statement is identical to the RUN statement, i.e., COMMON variables and open files are not preserved.

The advantages and requirements for linking with each library are discussed in Section 5.3, "Linking with the BRUN10.LIB Library," and Section 5.4, "Linking with the BCOM10.LIB Library."

5.2 Running the Linker

Microsoft LINK has the same style of interface as the Microsoft QuickBASIC Compiler. The linker requires two kinds of input: a command to start the linker, and responses to command prompts. Start the linker by typing

```
link
```

at the MS-DOS command level. Microsoft LINK prompts you for the input it needs by displaying the following four lines, one at a time; it waits for you to respond to each prompt before printing the next one.

```
Object modules [.OBJ]:  
Run file [.EXE]:  
List file [NUL.MAP]:  
Libraries [.LIB]:
```

The responses you can make to each prompt are explained in Sections 5.2.2 through 5.2.5.

Once you understand the Microsoft LINK prompts and operations, you can use the command-line method of running the linker. This method is described in Section 5.2.9. The command-line method lets you type all commands, options, and filenames on the line used to start Microsoft LINK.

Before you use the command-line method, you should understand how Microsoft LINK works and what your responses to its prompts mean. It is recommended that you allow Microsoft LINK to prompt you for responses until you are comfortable with its commands and operations.

5.2.1 Filename Conventions

Microsoft LINK uses the default file extensions ".OBJ", ".EXE", ".MAP", and ".LIB" for the object, executable, map, and library filenames, respectively, when you do not supply extensions with your filenames. You can override the default extension for a particular prompt by specifying a different extension. To enter a filename that has no extension, type the name followed by a period. For example, typing "ABC." in response to a prompt tells Microsoft LINK that the given name (in this case, ABC) has no extension, while typing just "ABC" tells Microsoft LINK to use the default extension for that prompt.

5.2.2 Object Modules Prompt

At the "Object modules" prompt, list the names of the object files and assembler routines you want to link. (If you are linking assembly language object modules, you must give the BASIC module as the first object module to be loaded; otherwise segments may be ordered incorrectly.) You must respond to this prompt. There is no default.

Microsoft LINK automatically supplies the ".OBJ" extensions when you give a filename without an extension. If your object file has a different extension, you must give the full name, with the extension, for the file to be found.

Pathnames are allowed with the object filenames. This means that you can give Microsoft LINK the pathname of an object file in another directory or on another disk. If Microsoft LINK cannot find a given object file, it displays a message and waits for you to change disks.

Each object filename must be separated from the next by blank spaces or a plus sign (+). If you need to type more filenames than will fit on a single line, enter a plus sign as the last character on the line. The "Object modules" prompt appears on the next line, allowing you to give more object files.

Example

The following response links two object modules, MATH.OBJ and ADD.OBJ:

```
Object modules [.OBJ]: math add
```

5.2.3 Run File Prompt

The "Run File" prompt lets you supply a name for the executable program file. You can use any filename you like; however, it is recommended that you use an ".EXE" extension, since MS-DOS will only execute files that have an ".EXE" extension.

You can skip this prompt by pressing the Return key without supplying a name. By default the linker gives the executable file the basename of the first ".OBJ" file listed at the previous prompt, replacing the ".OBJ" extension with an ".EXE" extension.

Example

The following command produces an executable file named MATH.EXE:

```
Run File [.EXE]: math
```

5.2.4 List File Prompt

At the "List File" prompt you can tell Microsoft LINK to create a listing file. A listing file contains the names of all segments in order of their appearance in the load module. (If you link with the /MAP switch you can also list all external symbols and their addresses. The /MAP switch is discussed in your *Microsoft MS-DOS User's Guide*.)

If you use a filename without an extension, the linker supplies the ".MAP" extension. The .MAP extension is not required, so you can supply another extension if you like. Microsoft LINK creates the map file in the current directory unless you give a different pathname.

You can choose not to create a listing file by pressing Return without giving a filename. The default response is the special filename NUL.MAP, which tells the linker *not* to create a listing file.

Example

The following response produces a map file named MATH.MAP:

```
List file [.MAP]: math
```

5.2.5 Libraries Prompt

Following the "Libraries" prompt you can give zero or more entries, separated by blank spaces or plus signs (+). If you are linking object modules written only in Microsoft QuickBASIC, you don't need to enter a filename; just press the Return key. The information about which library to link with is in the object file. Your program will be linked with the library BRUN10.LIB, unless you have compiled the program with the /O switch, in which case your program will be linked with the library BCOM10.LIB.

When you give a library name, Microsoft LINK searches for the given library and links it with your program. If the library name includes a directory specification, Microsoft LINK searches only that directory for the

library. If just a library name is given (no directory specification), Microsoft LINK uses the following search path to find the library:

1. The current working directory.
2. The directories listed following the "Libraries" prompt, in the order in which they are listed.
3. The libraries specified by the LIB environment variable. (Environment variables are explained in Section 2.3, "Setting Up the Environment.")

5.2.6 Separating Entries

Use the plus sign (+) or one or more space characters to separate filename entries in a list of object files or libraries.

5.2.7 Extending Lines

To extend a line, type the plus sign (+) as the last character of a line to be continued. (This is valid only for the "Object Files" and "Libraries" prompts.) The prompt will reappear on the next line, and you can add more entries. Do not type the plus sign in the middle of a filename entry; the plus sign may only be used after complete filenames.

Example

```
Object modules [.OBJ]: FUN TEXT TABLE CARE+
Object modules [.OBJ]: YOYO+ FLIPFLOP+ JUNQUE
```

5.2.8 Selecting Default Responses

You select default responses with the linker in the same way you select default responses with the compiler. For your convenience, this information is repeated here.

To select the default response to the current prompt, press Return without specifying a filename. The next prompt will appear.

To select default responses to the current prompt and all remaining prompts, use a semicolon (;) followed by a carriage return. Once you have entered a semicolon, you cannot respond to the remaining prompts for the link session. (The prompts will not appear on the screen.) Use this option to

save time when the default responses are acceptable. Note that the semicolon is not allowed with the first prompt, "Object Modules," because there is no default response for that prompt.

Defaults for the linker prompts are as follows:

1. The default for the "Run File" prompt is the basename of the first object file submitted to the "Object Files" prompt. The .EXE extension replaces the .OBJ extension.
2. The default for the "List File" prompt is the special filename NUL.MAP, which tells the linker *not* to create a listing file.

To terminate a session with Microsoft LINK at any time, press Control-C. If you type an erroneous response you may have to press Control-C to exit Microsoft LINK, then restart the linker.

5.2.9 Using a Command Line

To invoke the linker on a single command line, give your responses to the command prompts (which were discussed in the previous section) on the same line, following the **link** command. The responses to the prompts must be separated by commas, as shown below:

```
link object[,executable][,map[[,library]]] [/switch...] [:]
```

where

<i>object</i>	is a list of object modules, separated by plus signs or blanks.
<i>executable</i>	is the name of the file to receive the executable output.
<i>map</i>	is the name of the file to receive the map listing.
<i>library</i>	consists of the names of libraries to be searched, separated by plus signs.
<i>switch</i>	may be any Microsoft LINK switch described in the <i>Microsoft MS-DOS User's Guide</i> except the /DSAL-LOCATE and /HIGH switches. Using the prompt method, you can place switches after any filename response. If you are using the command-line method, switches can appear anywhere before the semicolon.

You can select the default response for any prompt by omitting the file-name or list before the comma. You can also select default responses by using the semicolon. The semicolon tells LINK to use the default responses for all remaining prompts.

Example

In the following example, Microsoft LINK loads and links the object modules FUN.OBJ, TEXT.OBJ, TABLE.OBJ, and CAR.OBJ, searching for unresolved references in the library BRUN10.LIB. The executable file produced is the default, FUN.EXE (indicated by the blank space). A list file named FUNLIST.MAP is also produced.

```
link FUN+TEXT+TABLE+CAR, .FUNLIST:
```

5.3 Linking with the BRUN10.LIB Library

Linking with the BRUN10.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 BCOM10.LIB, COMMON is not supported; CHAIN is equivalent to RUN.
- The BRUN10.EXE run-time module resides in memory. Therefore, it does not need to be reloaded for each program in a system of chained programs.
- Since BRUN10.LIB contains only the code for reading in the BRUN10.EXE module, the actual routines from BRUN10.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 BRUN10.LIB can be as much as 15 to 20 percent smaller than the code generated with BCOM10.LIB. This is because the run-time system uses software interrupts to invoke run-time routines. The BCOM10.LIB uses intersegment calls.

5.4 Linking with the BCOM10.LIB Library

You link to the BCOM10.LIB library the same way you link to the BRUN10.LIB library. However, the two libraries themselves are very different. The /O switch must be specified at compile time if your program is to be linked to the BCOM10.LIB library. When you specify /O, the alternate run-time library (BCOM10.LIB) is substituted for BRUN10.LIB as the default library to be searched at link time.

Note that when BCOM10.LIB is selected as the library to be searched, the run-time module is not used by your program.

Linking with BCOM10.LIB provides the following advantages:

- You may save RAM space at run time by linking with BCOM10.LIB. This is most likely when you have small, simple programs that do not require all the routines in the run-time module.
- With BCOM10.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 BCOM10.LIB, programs execute slightly faster because the run-time routines are invoked through 8086 intersegment calls. With BRUN10.LIB, the run-time routines are invoked through software interrupts.

For more information on using CHAIN and COMMON with a system of programs, see Chapter 7, "Creating Structured Programs."

5.5 Running a Program

To run a compiled program, enter the filename without its .EXE filename extension. For example, to run the program SAMPLE.EXE, type

```
sample
```

and press Return. This command executes the program SAMPLE.EXE. If you have linked the program SAMPLE.EXE to BRUN10.LIB, then the computer loads the BRUN10.EXE run-time module. 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 BRUN10.EXE  
Enter new path specification:
```

At this point, enter the new path specification or a drive letter followed by a colon and press Return. 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 Microsoft QuickBASIC program. Chaining to non-Microsoft QuickBASIC programs will leave the other-language files open when execution returns to the chained-from 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 RUN 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 8.6, "The Run-Time Segment Maps," for a diagram of the run-time memory maps.

Chapter 6

Using Metacommands

- 6.1 Metacommand Syntax 65
- 6.2 Source Listing Format: \$LIST 66
- 6.3 Object Code Listing
Format: \$OCODE 66
- 6.4 Controlling the Listing Format 67
- 6.5 Processing Additional
Source Files: \$INCLUDE 69
- 6.6 Dimensioned Array Allocation:
\$STATIC and \$DYNAMIC 69
- 6.7 Changing Internal Module
Names: \$MODULE 70

Metacommands tell the compiler to perform certain actions while it is compiling the source file. Metacommands can do the following:

- Control the format of listing files
- Control what parts of the source file are included in a listing file
- Read in and compile other BASIC source files at specific points during compilation
- Control the allocation of dimensioned arrays

This chapter describes the metacommands available with the Microsoft QuickBASIC Compiler, and how to use them. There is a summary of each metacommand and its function in Appendix A, "Summary of Commands."

6.1 Metacommand Syntax

Metacommands begin with a dollar sign (\$) and are always enclosed in a program comment. More than one metacommand can be given in one comment. Multiple metacommands are separated by white-space characters: space, tab, or linefeed. Metacommands that take arguments have a colon between the metacommand and the argument:

REM \$METACOMMAND: *argument*

Arguments that are strings of characters are enclosed in single quotation marks. White space between the elements of a metacommand is ignored. The following are also valid forms for metacommands:

```
rem $metacommand1 $metacommand2
rem $metacommand1 : 'argument' $metacommand2
```

Note that *no space* may appear between the dollar sign and the rest of the metacommand.

To put 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
```

6.2 Source Listing Format: \$LIST

The \$LIST metacommand turns on and off source listing. Source code listing is turned on by default. To turn source code listing off at any point in the program, add the following line:

```
rem $list-
```

To turn source code listing back on, insert the following line:

```
rem $list+
```

The \$LIST metacommand has no effect on whether or not a source code listing is produced; it only affects what parts of the source code are placed in the listing. A source code listing is produced only if you request it when you start the compiler.

6.3 Object Code Listing Format: \$OCODE

The \$OCODE metacommand turns on and off disassembled object code listing. When the \$OCODE metacommand is used, disassembled object code is produced only for the portion of the source file between the \$OCODE+ and \$OCODE- metacommands.

The \$OCODE metacommand has no effect on whether or not a source code listing is produced; it only affects what parts of the source code a disassembly listing is produced for. To produce a listing you still must give a file name in response to the "Listing File" prompt.

There is a sample disassembly listing file in Appendix C, "Listing File Formats."

6.4 Controlling the Listing Format

The metacommands listed in Table 6.1 control the listing file format. Note that there must be a colon between the metacommand and its argument.

Table 6.1
Listing Format Commands

Metacommand	Effect
<code>\$TITLE: <i>title</i></code>	Sets listing title
<code>\$SUBTITLE: <i>subtitle</i></code>	Sets listing subtitle
<code>\$LINESIZE: <i>size</i></code>	Sets width of listing, in columns
<code>\$PAGESIZE: <i>size</i></code>	Sets length of listing, in lines
<code>\$PAGE</code>	Skips to next page
<code>\$PAGEIF: <i>number</i></code>	Skips to next page if there are <i>number</i> lines or less left on listing page
<code>\$SKIP[:<i>number</i>]</code>	Skips <i>number</i> lines or to end of page

Example

```

10 rem $title:'Sample listing'
20 rem $subtitle:'Hello program'
40 for A = 1 to 10
50 rem $page
60   print "Hello, world!"
70 next A

```

A listing file for the above program looks like this:

SAMPLE LISTING

PAGE1
01-01-85
10:15:46

Offset	Data	Source Line	Microsoft QuickBASIC Compiler Version 1.0
0030	0006	10 rem \$title:'Sample listing'	
0030	0006	20 rem \$subtitle:'Hello program'	
0030	0006	30 for A = 1 to 10	
0043	0006	40 rem \$page	

Second page:

Sample listing
Hello program

PAGE 2
01-01-85

10:15:46

Offset	Data	Source Line	Microsoft QuickBASIC Compiler Version 1.0
0043	0006	60 print "Hello world!"	
004F	0006	70 next A	
0068	000A		
006B	000A		

50450 Bytes Available
50076 Bytes Free

0 Warning Error(s)
0 Severe Error(s)

6.5 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 end-of-file of the included source 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 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.

6.6 Dimensioned Array Allocation: \$STATIC and \$DYNAMIC

The \$STATIC and \$DYNAMIC metacommands tell the compiler how to allocate memory for arrays that are dimensioned with integer-constant subscripts.

If the \$STATIC metacommand is present, all subsequent arrays declared with constant-integer upper bounds are statically allocated. Static arrays cannot be redimensioned with the REDIM or ERASE statements. The ERASE statement changes the value of the array elements to zero (or to a null string), but the array dimensions remain the same.

If the \$DYNAMIC metacommand is present, all subsequent arrays declared in a DIM statement are dynamically allocated. Dynamic arrays can be redimensioned with the DIM or REDIM statements, and dynamic-array memory can be reallocated with the ERASE statement.

Using \$STATIC and \$DYNAMIC with the DIM, REDIM, ERASE, and COMMON statements is discussed in detail in Chapter 3, "Developing a Program."

6.7 Changing Internal Module Names: \$MODULE

The \$MODULE metacommand allows you to change the internal module name that is passed to the linker. This is useful when you want the module name to be different from that of the source file.

If you use the \$MODULE metacommand, it must appear before the first executable statement.

If each of the Microsoft BASIC modules you link together does not have a unique module name, the results will be unpredictable.

Note

The segment names generated by the compiler have changed with this release. These changes should not affect any assembly-language routines linked with a Microsoft BASIC application, but if any problems arise, consult the run-time maps in Chapter 8, "Using Assembly-Language Subroutines."

Chapter 7

Creating Structured Programs

7.1	Structuring Programs	73
7.2	Creating Subprograms	74
7.3	Invoking BASIC Subprograms with CALL	75
7.3.1	Passing Variables	76
7.3.2	Passing Arrays	77
7.3.3	Passing Expressions	78
7.4	Passing Parameters within a Module	78
7.4.1	Passing Parameters with the SHARED Attribute	79
7.4.2	Passing Parameters with the SHARED Statement	80
7.5	Using a Library of Subprograms: Passing Parameters with COMMON	80
7.6	Preserving Subprogram Variable Values	81
7.7	Preserving Subprogram Array Values	82
7.8	Common Errors in Calling Subprograms	82
7.9	Using CALL or CHAIN	84

In interpreted Microsoft BASIC, programs consist of only one source file. For many programs, a single source file is adequate. However, in some cases you can develop large programs more efficiently as a system of individual source files that are compiled separately, then linked to form a single executable program. Using this approach, you can create separate source files that perform specific functions and use them over and over, in one program or in many different programs. Each independently compilable segment of code used by another program (called the "main program") is called a "subprogram." This method of programming is called "modular programming." Modular programming has three major benefits:

1. **Clarity.** Each subprogram has a specific task it performs on a small number of arguments or common variables. This can make the program much easier to understand.
2. **Independence.** Subprograms can be developed as independent units, permitting repeated use in many different applications.
3. **Flexibility.** Subprograms can be modified and rewritten without affecting other programs, as long as the interface is the same.

This chapter describes how to create subprograms and how to call them from the main program. The syntax of the SUB and CALL statements and how to pass values between subprograms are also explained.

7.1 Structuring Programs

A complete subprogram call looks like this:

```
(main program)
.
.
CALL subprogram (arguments)
.
.
(end of main program)

SUB global-name (parameter-list) STATIC
    (body of subprogram)

END SUB
```

Sections 7.2 and 7.3 describe the SUB and CALL statements.

There are two methods of creating structured programs:

1. Creating a file that contains the main program and all the subprograms needed to execute the main program.
2. Creating several files (each file is called a "module"), where each module contains a single subprogram. Each file can be compiled separately and called from the main program.

The first approach works best when the main program will use all or most of the subprograms in the file.

If you have programs that use different combinations of subprograms, you will probably want to use the second method to create a "library" of subprogram modules. When the object files are linked, the subprograms are combined with the main program when you specify the pathname of the module(s) on the **link** command line. (If you have the Microsoft Library Manager, MS-LIB, you can use it to process libraries of object modules. Then you only need to specify the library name on the **link** command line. The called object modules will be linked from the library.)

Each method of structuring programs has different parameter-passing requirements. You can always pass parameters with the **CALL** statement, but if you have many parameters to pass, this method is impractical. If you are using a single file that contains the main programs and all subprograms, you can pass parameters with the **SHARED** attribute and the **SHARED** statement, described in Section 7.4. If you are using a library of subprograms, you can pass parameters with the **COMMON** statement, described in Section 7.5.

7.2 Creating Subprograms

Subprograms are delimited by SUB and END SUB or EXIT SUB statements. The SUB statement has the following form:

```
SUB global-name [(parameter-list)] STATIC
```

where

global-name

is a variable name up to 31 characters long. The name cannot appear in any other SUB statement in the same program module.

<i>parameter-list</i>	contains simple 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 supported, and a warning error is generated if STATIC is omitted.

Variables in the formal parameter list are declared the same way they are declared in a program. Variable types must correspond to the variable type being passed from the main program. Arrays must be declared in the subprogram parameter list in the following form:

array-name (*dim-num*)

where

<i>array-name</i>	is the name of the array.
<i>dim-num</i>	is an integer representing the number of dimensions in the array.

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

- User-defined functions.
- A SUB...SUB END block. Subprograms cannot be nested.

7.3 Invoking BASIC Subprograms with CALL

The CALL statement invokes Microsoft QuickBASIC subprograms or assembly-language subroutines. This section and the following sections apply only to CALL when it is used with subprograms. Calling assembly-language routines is described in Chapter 8, "Using Assembly-Language Subroutines."

<i>parameter-list</i>	contains simple 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 supported, and a warning error is generated if STATIC is omitted.

Variables in the formal parameter list are declared the same way they are declared in a program. Variable types must correspond to the variable type being passed from the main program. Arrays must be declared in the subprogram parameter list in the following form:

array-name (*dim-num*)

where

<i>array-name</i>	is the name of the array.
<i>dim-num</i>	is an integer representing the number of dimensions in the array.

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

- User-defined functions.
- A SUB...SUB END block. Subprograms cannot be nested.

7.3 Invoking BASIC Subprograms with CALL

The CALL statement invokes Microsoft QuickBASIC subprograms or assembly-language subroutines. This section and the following sections apply only to CALL when it is used with subprograms. Calling assembly-language routines is described in Chapter 8, "Using Assembly-Language Subroutines."

The CALL statement has the following syntax:

CALL name [(argument-list)]

where

<i>name</i>	is the name of the called subprogram. The name is limited to 31 characters.
<i>argument-list</i>	is a list of parameters (variables, array elements) passed to the called subprogram.

The CALLS statement has the same syntax as the CALL statement, but passes segmented addresses of its arguments. (The CALL statement passes the unsegmented addresses of its arguments.) You should use CALLS only to call assembly-language subprograms.

7.3.1 Passing Variables

Variables are passed to subprograms by reference; i.e., their location in memory, not their value, is passed to the subprogram. The subprogram can change the value of the variable by assigning a new value to the variable's corresponding argument. The following example shows how a CALL statement invokes a subprogram and passes variables by reference:

```
A=5 : B=0
call square (A,B)
print A,B
end

sub square(X,Y) static
  Y=X*X
end sub
```

The previous example prints the results 5 and 25. When the subprogram SQUARE begins executing, the initial value of Y, which is passed to the subprogram from argument B of the CALL statement, is 0. The subprogram passes the computed value of parameter Y back to the main program.

To pass a simple 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 its value, which is 0. When variables are passed by value, the subprogram cannot change them. In the above example, subprogram

SQUARE cannot change the value of Y because it cannot change the value passed from argument B of the CALL statement. The PRINT statement prints the results 5 and 0.

7.3.2 Passing Arrays

You can pass arrays with the CALL statement. The following statement passes two-dimensional array1, array2, and total to the subprogram MATADD2:

```
call matadd2 (array1(), array2(), total())
```

Subprogram MATADD2 might begin like this:

```
sub matadd2 (a(2), b(2), c(2)) static
```

To pass the values of individual array elements, enclose their subscripts in parentheses. The following statement passes the value of the element in row 3, column 9, of array1 to MATADD2:

```
call matadd2 (array1((3,9)))
```

Because only one integer is being passed, the SUB statement only needs an integer parameter, such as:

```
sub matadd2 (a) static
```

You can also pass array elements by reference, but because dynamic array elements passed by reference do not pass the actual address of that array element, you may not be able to access other elements of the same array using the offset of the original array element.

The LBOUND and UBOUND functions are useful for determining the size of an array passed to a subprogram. LBOUND finds the lower bound of an array; UBOUND the upper bound. Each function has two syntaxes: a general syntax and an abbreviated syntax for one-dimensional arrays. LBOUND and UBOUND have the following form:

Function	Array Type
LBOUND(array)	One-dimensional arrays
LBOUND(array,n)	n-dimensional arrays
UBOUND(array)	One-dimensional arrays
UBOUND(array,n)	n-dimensional arrays

The MATADD2 subprogram in the previous example can use LBOUND and UBOUND instead of passing the upper bounds explicitly:

```
call matadd2(array1(),array2(),total())
.
.
sub matadd2(a(2),b(2),c(2)) static
  'assume arrays have same dimensions
  for i%=lbound(a,1) to ubound(a,1)
    for j%=lbound(a,2) to ubound(a,2)
      c(i%,j%)=a(i%,j%)+b(i%,j%)
    next j%
  next i%
end sub
```

7.3.3 Passing Expressions

You can also pass expressions as arguments to subprograms. An argument can be any valid Microsoft BASIC expression except simple variables and array-element references. When BASIC encounters an expression in the argument list of a CALL statement, it assigns a temporary variable to the expression. This temporary variable is passed by reference to the subprogram, and this is equivalent to passing the value of the expression.

7.4 Passing Parameters within a Module

Variables and arrays declared and referenced in subprograms are usually considered local to the subprogram. However, Microsoft QuickBASIC supports shared variables and provides two ways for their values to be preserved across subprogram calls within a single module. One method is to give variables declared in COMMON, DIM, and REDIM statements the SHARED attribute, which permits the use of the variables in all subprograms within the module. The second method is to use the SHARED statement to declare main variables used in a subprogram. The first method is most useful if all variables will be shared by all subprograms. The second method is most useful when different subprograms use different combinations of main program variables. Sections 7.4.1 and 7.4.2 discuss these two ways to pass parameters.

7.4.1 Passing Parameters with the SHARED Attribute

Variables and arrays declared and referenced in subprograms are usually considered local to the subprogram. However, Microsoft QuickBASIC supports shared variables within a module and preserves their values across subprogram calls.

You can access variables from the main program without passing them as parameters to the CALL statement by using the SHARED attribute with COMMON, DIM, or REDIM statements in the main program. You can also pass variables among all subprograms in a module with the SHARED statement.

To declare variables as shared by the main program and all subprograms in a module, place the SHARED attribute directly after the keyword in a COMMON, DIM, or REDIM statement.

Within a module, subprograms can use the main program variables declared with the SHARED attribute.

Example

The following statements declare variables shared across all subprograms within a module:

```
common shared a,b,c
dim shared x,y,array (10,10)
redim shared alpha(n%)
```

In the following module, the main program passes the value of A to subprogram PROG2. When the executable program is run, it prints the number "10."

```
common shared A
A=6+4
call prog2

sub prog2 static
  print A
end sub
```


7.4.2 Passing Parameters with the SHARED Statement

Within a single module, use of the SHARED statement allows subprograms to use main program variables that are not declared with the SHARED attribute. The SHARED statement has the form

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

where *variable* is the name of any variable declared in the main program.

Example

To use variables A and D declared in the main program, the subprogram must declare them in a SHARED statement before they are used, as in the following example:

```
A=4:D=10
call prog2

sub prog2 static
  shared A,D
  X=A+D
  print X
end sub
```

7.5 Using a Library of Subprograms: Passing Parameters with COMMON

If the subprograms called by the main program are in one or more separate source files, you must pass variables and arrays with the COMMON statement. The COMMON statement has the form

COMMON [SHARED] [/blockname/] *item-list*

where

SHARED

is an optional attribute. You use the SHARED attribute when you want a subprogram in the same module to access the variable declared in the COMMON statement.

7.4.2 Passing Parameters with the SHARED Statement

Within a single module, use of the SHARED statement allows subprograms to use main program variables that are not declared with the SHARED attribute. The SHARED statement has the form

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

where *variable* is the name of any variable declared in the main program.

Example

To use variables A and D declared in the main program, the subprogram must declare them in a SHARED statement before they are used, as in the following example:

```
A=4:D=10
call prog2

sub prog2 static
  shared A,D
  X=A+D
  print X
end sub
```

7.5 Using a Library of Subprograms: Passing Parameters with COMMON

If the subprograms called by the main program are in one or more separate source files, you must pass variables and arrays with the COMMON statement. The COMMON statement has the form

COMMON [SHARED] [/blockname/] *item-list*

where

SHARED

is an optional attribute. You use the SHARED attribute when you want a subprogram in the same module to access the variable declared in the COMMON statement.

<i>blockname</i>	is any valid BASIC identifier up to 31 characters long. Use <i>blockname</i> when every subprogram does not share every variable in the <i>item-list</i> .
<i>item-list</i>	is a list of variables and arrays that will be used by the subprograms.

If the programs in the following example are placed in three separately compiled files, COMMON statements must be used, such as:

```
'main program file:
common a,d
a=4:d=6
call prog2
call prog3
```

```
'prog 2:
common a,d
sub prog2 static
  shared a,d
  e=a*d
  print e
end sub
```

```
'prog3:
common a,d
sub prog3 static
  shared a,d
  e=a*d
  print e
end sub
```

7.6 Preserving Subprogram Variable Values

BASIC assumes initial values of zero for numeric variables and null for string variables. However, if a subprogram is exited and then reentered, the values retained for the variables in the subprogram depend on the individual system. Declaring the variables in a STATIC statement guarantees that the previous values are retained. For example, the following statement declares the integer variables I and J as local, preserving their values when the subprogram is exited:

```
static I%,J%
```

Do not confuse the `STATIC` statement with the `$STATIC` metacommand. The `$STATIC` metacommand affects the way space is allocated for arrays, while the `STATIC` statement preserves subprogram variable and array values.

7.7 Preserving Subprogram Array Values

You can preserve the values of arrays in subprograms by declaring the array in a `STATIC` statement. `STATIC` arrays are not allocated until they are dimensioned (with the `DIM` statement) or redimensioned (with the `REDIM` statement). For example, the following `STATIC` statement declares arrays `X`, `Y`, and `Z` as local. The number in parentheses is the number of dimensions in the array. The actual dimensions are declared in the following `REDIM` statement:

```
static X(2), Y(2), Z(1)
redim X(I%, J%), Y(10, 20), Z(3)
```

7.8 Common Errors in Calling Subprograms

Two errors are commonly made when calling subprograms:

- Mismatched argument and parameter lists
- Variable aliasing

Mismatched argument and parameter list errors are caused when the order, type, or number of arguments passed to a BASIC subprogram do not correspond to the parameters in the subprogram. The Microsoft 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

```

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. In structured programs, variable aliasing occurs when an argument passed to a subprogram can be referenced in the subprogram in more than one way. This often happens when the same variable is used twice as a parameter to CALL, 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 and make minimal use of SHARED.

Example

The following is a simple example that 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, this 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.

7.9 Using CALL or CHAIN

While programs that contain CHAIN statements without line number arguments will compile correctly, unless you are compiling a very large program on a computer with a small amount of memory, it is suggested that you use the CALL statement instead of the CHAIN statement.

The objective of a well-structured program is to break individual tasks into separate modules of code, then bring the modules into the main program as they are required. The CALL statement has been created specifically for this purpose. After a called program is finished executing, control returns to the main program. On the other hand, CHAIN does not return control to the chained-from program. CALL also allows you to use assembly-language routines, providing greater flexibility in your programs. Finally, because CHAIN requires disk I/O to find and read the target file, it is significantly slower than CALL.

Chapter 8

Using Assembly-Language Subroutines

8.1	Converting Interpreted Programs	87
8.2	Calling Assembly-Language Subroutines with CALL and CALLS	87
8.3	Coding Subroutines for the Compiler	91
8.4	The Run-Time Memory Maps	93
8.4.1	The BRUN10.EXE Run-Time Module Environment	94
8.4.2	The BRUN10.LIB Run-Time Library Environment	95
8.5	Segment Maps	96
8.6	The Run-Time Segment Maps	97

Microsoft QuickBASIC interfaces with assembly-language subroutines through the CALL statement. All you need to do to use assembly-language subroutines with the compiler is call them with the correct number and type of parameters, then 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.

This chapter explains how to call assembly-language subroutines from Microsoft QuickBASIC programs, and what you need to know to convert your interpreted programs that call assembly-language subroutines so that they can be used with the compiler. This chapter also describes the rules for coding assembly-language subroutines called by compiled programs.

8.1 Converting Interpreted Programs

You must make the following modifications to your interpreter programs before they can be compiled:

1. Change CONST to CSEG; change DATA to DSEG.
2. You cannot use FAC, MAKINT, and FRCINT with the Microsoft QuickBASIC Compiler.

8.2 Calling Assembly-Language Subroutines with CALL and CALLS

The CALL statement calls an assembly-language subroutine and passes the routine the unsegmented addresses of the statement arguments. The CALLS statement has the same syntax as the CALL statement, but passes the segmented addresses of its arguments.

The CALL statement has the following syntax:

CALL *name* [(*argument-list*)]

where

<i>name</i>	is the name of the called subroutine. The name is limited to 31 characters and, for an assembly-language subroutine, must be a PUBLIC symbol.
<i>argument-list</i>	is a list of parameters (variables, array elements) passed to the called subroutine.

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 pointer to the element onto the stack. Note that elements adjacent to the pointer are not guaranteed to contain the values of adjacent elements of the array.
3. BASIC pushes the 2-byte offset of each nonarray argument's location in the data segment (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.

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

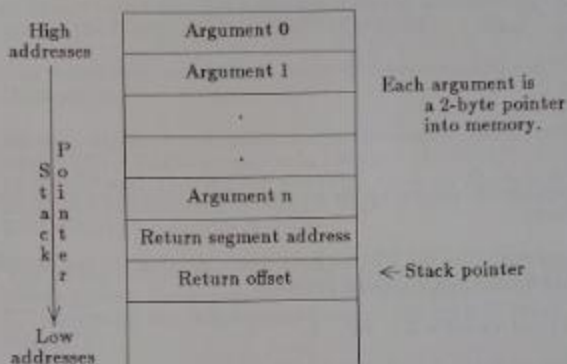


Figure 8.1 The Stack after CALL Executes

The assembly-language subroutine now has control. Arguments are referenced by saving the contents of BP, then moving the stack pointer (SP) to the base pointer (BP) and adding a positive offset to BP, as the following example shows:

```
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— k , for example—by using the following formula, where k is the argument and n is the total number of arguments made by the program:

$$\text{Location } k = BP + 6 + (2 * (n - k))$$

The following formula gives you the location of argument 0:

$$\text{Location of argument 0} = BP + 6 + 2n$$

If you have local variables, you should begin the assembly-language program with language 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:

$$\text{Location of local variable 0} = \text{BP} - 2$$

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. You can use the following formula to calculate the location of the *k*th local variable:

$$\text{Location of local variable} = \text{BP} - 2(k + 1)$$

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
```

8.3 Coding Subroutines for the Compiler

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

1. The called routine must not destroy the contents of the BP, SS, and DS registers.
2. All data declared and referenced in the routine should be in the segment DSEG. DSEG should be 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 RET *n* (where *n* is two times the number of parameters in the argument list) to adjust the stack to the start of the calling sequence.
5. The routine must return values to BASIC. It does this when you include the variable name in the argument list that will receive the result.
6. The routine *must not* change the length of any strings. Microsoft BASIC 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 32767). Bytes 2 and 3, respectively, are the lower and upper 8 bits of the string's starting address in string space. Changing the length of the string or modifying the string descriptor will cause unpredictable results, probably a "String Space Corrupt" error message.

Examples

Passing an integer

The following program demonstrates how to call an assembly-language sub-routine, 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 program, 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

incr    public  incr
incr    proc    far
        push    bp
        mov     bp,sp
        mov     bx,[bp]+6
        mov     ax,storage
        add     [bx],ax
        pop     bp
        ret     2
incr    endp

code    ends

        end
```

:routine is a FAR procedure
:save BP
:set up to address of BP
:fetch address of param 0
:add value to param 0
:restore BP
:return and remove 1 param from stack

Passing a string

The following example demonstrates how to access string arguments passed and store a return result in a variable. The following statement calls the routine:

```
call entry (B$)
```

The assembly-language procedure can appear as follows:

```
entry proc far
    push bp
    mov bp, sp           ;get current stack position bp
    mov bx, 6[bp]        ;get address of B$ in descriptor
    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
```

8.4 The Run-Time Memory Maps

Figures 8.2 and 8.3 in this section show run-time memory maps for programs linked to the run-time libraries BRUN10.LIB and BCOM10.LIB.

8.4.1 The BRUN10.EXE Run-Time Module Environment

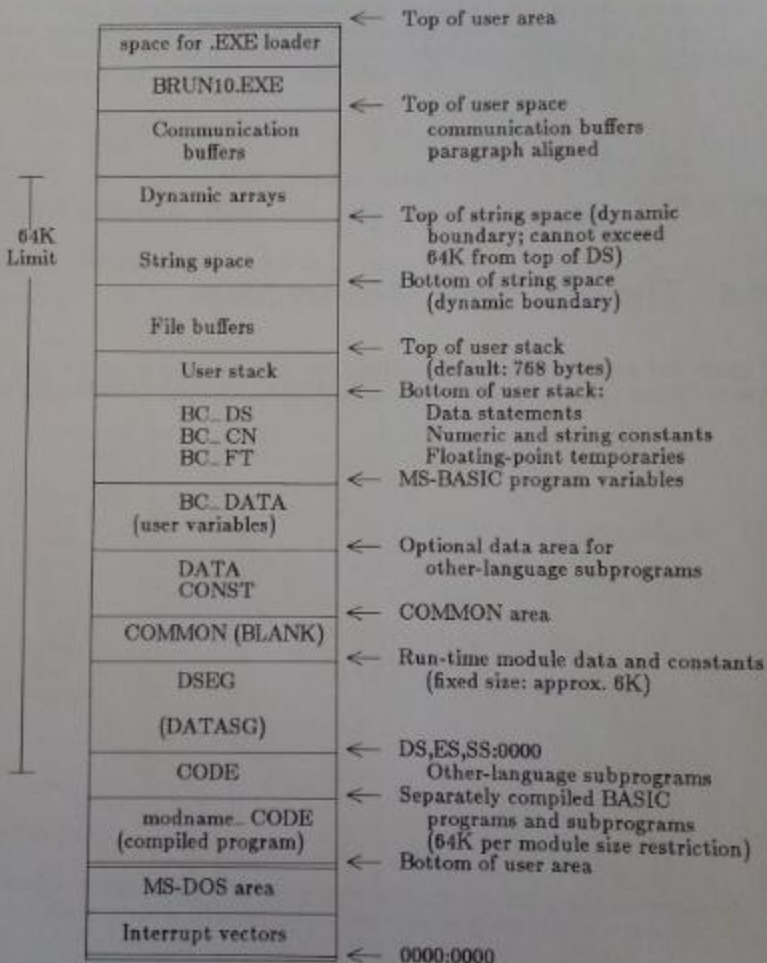


Figure 8.2 Run-Time Module Environment

8.4.2 The BCOM10.LIB Run-Time Library Environment

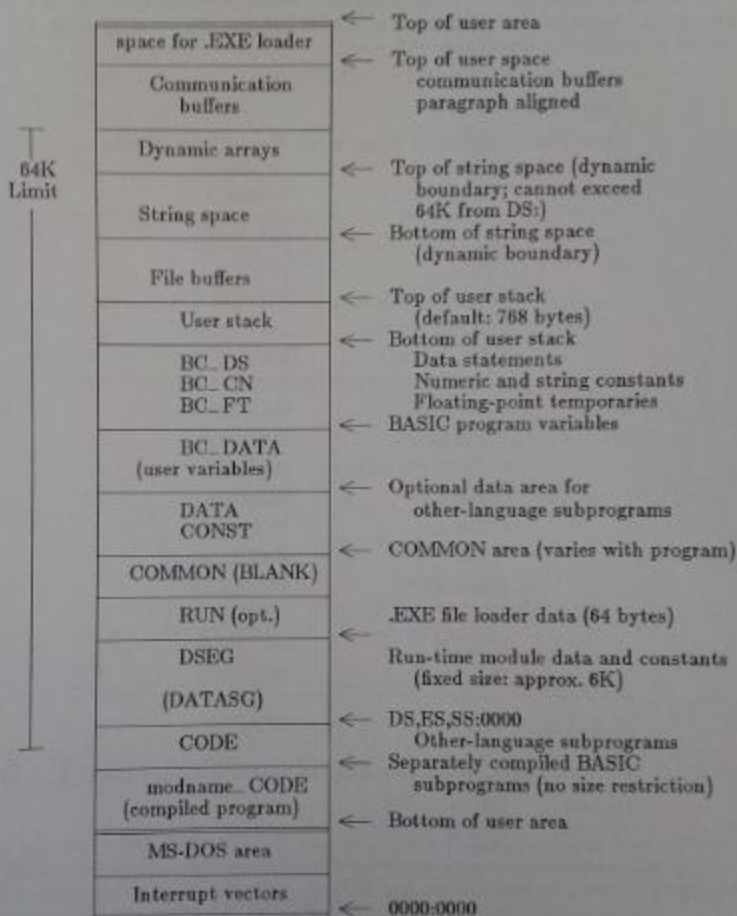


Figure 8.3 Run-Time Library Environment

8.5 Segment Maps

The segment maps for compiled programs under Microsoft BASIC are almost the same for versions with and without the run-time module, as shown in Table 8.1 and Table 8.2.

Table 8.1

Run-Time Segment Map with Run-Time Module

Address	Segment	Class
Low	modname_ CODE	BC_ CODE
CS	CODE	CODE
	BC_ ICN_ CODE	INIT_ CODE
	BC_ IDS_ CODE	INIT_ CODE
	INIT_ CODE	INIT_ CODE
Low	DSEG	DATASG
DS	RTMLOAD	DATASG
	PSEUDOCOMMON	DATASG
	COMMON	BLANK
	CONST	CONST
	DATA	DATA
	BC_ DATA	BC_ VARS
	BC_ FT	BC_ SEGS
	BC_ CN	BC_ SEGS
	BC_ DS	BC_ SEGS
High	STACK	STACK
DS		
Highest memory	BRUN10.EXE Run-time module code	

Table 8.2

Run-Time Segment Map
without the Run-Time Module

Address	Segment	Class
Low	modname_ CODE	BC_ CODE
CS	CODE	CODE
	CSEG	CODESG
	SHELL	CODESG
	CLEAR	CODESG
	BC_ ICN_ CODE	INIT_ CODE
	BC_ IDS_ CODE	INIT_ CODE
	INIT_ CODE	INIT_ CODE
Low DS	DSEG	DATASG
	RTMLOAD	DATASG
	PSEUDOCOMMON	DATASG
	COMMON	BLANK
	CONST	CONST
	DATA	DATA
	BC_ DATA	BC_ VARS
	BC_ FT	BC_ SEGS
	BC_ CN	BC_ SEGS
	BC_ DS	BC_ SEGS
High DS	STACK	STACK

8.6 The Run-Time Segment Maps

The segments BC_ ICN_ CODE and BC_ IDS_ CODE are block transferred to the segments BC_ CN and BC_ DS at program initialization. Just before the user program itself executes, the DS segment is moved down in physical memory over the segments of class INIT_ CODE.

All the classes and segments in the data segment (DS) are in the group DGROUP.

The contents of the segments are as follows:

Segment Name	Contents
BC_ CODE	Compiled user program
modname_ CODE	Separately compiled BASIC programs
CSEG	BASIC run-time routines
CODE	Other-language subprograms
BC_ ICN_ CODE	User-program constants (moved to BC_ CN)
BC_ IDS_ CODE	User-program data statements (moved to BC_ DS)
INIT_ CODE	Disposable run-time initialization code
DSEG	Run-time uninitialized data values
RTMLOAD	Relocatable data segment to be used by RUN statement
PSEUDOCOMMON	User-defined softkey string descriptors
COMMON	User-program COMMON area
CONST	Initialized data for use by other-language subprograms
DATA	Uninitialized data for use by other-language subprograms
BC_ DATA	User-program data variables
BC_ FT	User-program floating-point temporaries
BC_ CN	User-program constants
BC_ DS	User-program data statements
STACK	Stack segment required by loader (not used)

BASIC sets up string space and dynamic numeric array space at initialization time. The string and heap are allocated the balance of available memory, with the array heap occupying the memory between the end of string space and the run-time module. The string space is used for allocations to the small heap, which contains file buffers and dynamic string arrays. All communication buffers and dynamic numeric (integer and floating-point) arrays are stored in the dynamic array heap. The array heap is allocated

All the classes and segments in the data segment (DS) are in the group DGROUP.

The contents of the segments are as follows:

Segment Name	Contents
BC_CODE	Compiled user program
<i>modname</i> _CODE	Separately compiled BASIC programs
CSEG	BASIC run-time routines
CODE	Other-language subprograms
BC_ICN_CODE	User-program constants (moved to BC_CN)
BC_IDS_CODE	User-program data statements (moved to BC_DS)
INIT_CODE	Disposable run-time initialization code
DSEG	Run-time uninitialized data values
RTMLOAD	Relocatable data segment to be used by RUN statement
PSEUDOCOMMON	User-defined softkey string descriptors
COMMON	User-program COMMON area
CONST	Initialized data for use by other-language subprograms
DATA	Uninitialized data for use by other-language subprograms
BC_DATA	User-program data variables
BC_FT	User-program floating-point temporaries
BC_CN	User-program constants
BC_DS	User-program data statements
STACK	Stack segment required by loader (not used)

BASIC sets up string space and dynamic numeric array space at initialization time. The string and heap are allocated the balance of available memory, with the array heap occupying the memory between the end of string space and the run-time module. The string space is used for allocations to the small heap, which contains file buffers and dynamic string arrays. All communication buffers and dynamic numeric (integer and floating-point) arrays are stored in the dynamic array heap. The array heap is allocated

from string space as needed and occupies all available memory between the end of the data segment and the run-time module.

All STATIC arrays are allocated in BC_ DATA.

In general, while a Microsoft BASIC Compiler program is running, the segment registers (DS, ES, and SS) are the same. Register CS varies depending on the BASIC program or run-time code that is executing.

Chapter 9

Microsoft QuickBASIC Compiler Reference

9.1	Compiler/Interpreter Differences	103
9.2	Reference Format	104
9.3	Reference Syntax Notation	105
9.4	Reference Input/Output Notation	106

Table 9.1 lists the commands, functions, and statements covered in the alphabetical reference that follows Section 9.4. This is not a complete reference to the Microsoft BASIC language; instead, it focuses on compiler enhancements, new compiler statements, and statements that behave differently in the compiler, and so may require some changes to programs written with the Microsoft BASIC interpreter.

Table 9.1

Functions and Statements in This Reference

Enhanced in Compiler	New to Compiler†	May Require Modifying of Interpreter Programs
COMMON	COMMAND\$	CALL
DEF FN	LBOUND	CALLS
END	LOCK	CHAIN
ERASE	REDIM	DEF type
FOR...NEXT	SHARED	DRAW
FRE	STATIC	PLAY
GOSUB	SUB...END SUB	RESUME
GOTO	UBOUND	RUN
RETURN	UNLOCK	
WIDTH		

† For information on the compiler metacommands (%INCLUDE,%DYNAMIC, etc.) see Chapter 6, "Using Metacommands," as well as Section A.1, "Microsoft QuickBASIC Metacommands."

See the reference manual that came with your BASIC interpreter for a complete reference to the functions and statements that work identically in both interpreted and compiled programs.

9.1 Compiler/Interpreter Differences

The BASIC understood by your QuickBASIC Compiler is slightly different from interpreted BASIC. You may find it necessary to modify the source code in your interpreted BASIC program for one or more of the following reasons:

1. Your program would work more efficiently with some of the new or enhanced commands and statements supported by the QuickBASIC Compiler. (See Table 9.1.)

Table 9.1 lists the commands, functions, and statements covered in the alphabetical reference that follows Section 9.4. This is not a complete reference to the Microsoft BASIC language; instead, it focuses on compiler enhancements, new compiler statements, and statements that behave differently in the compiler, and so may require some changes to programs written with the Microsoft BASIC interpreter.

Table 9.1
Functions and Statements in This Reference

Enhanced in Compiler	New to Compiler†	May Require Modifying of Interpreter Programs
COMMON	COMMANDS	CALL
DEF FN	LBOUND	CALLS
END	LOCK	CHAIN
ERASE	REDIM	DEF type
FOR...NEXT	SHARED	DRAW
PRE	STATIC	PLAY
GOSUB	SUB...END SUB	RESUME
GOTO	UBOUND	RUN
RETURN	UNLOCK	
WIDTH		

† For information on the compiler metacommands (\$INCLUDE,\$DYNAMIC, etc.) see Chapter 6, "Using Metacommands," as well as Section A.1, "Microsoft QuickBASIC Metacommands."

See the reference manual that came with your BASIC interpreter for a complete reference to the functions and statements that work identically in both interpreted and compiled programs.

9.1 Compiler/Interpreter Differences

The BASIC understood by your QuickBASIC Compiler is slightly different from interpreted BASIC. You may find it necessary to modify the source code in your interpreted BASIC program for one or more of the following reasons:

1. Your program would work more efficiently with some of the new or enhanced commands and statements supported by the QuickBASIC Compiler. (See Table 9.1.)

2. Your program has certain commands and statements that work differently in the compiler and the interpreter. (See Table 9.1.)
3. Your program contains certain commands or statements, such as ON ERROR GOTO, that require it be compiled with a special option. (See Section 3.2.4, "Operational Differences," and Appendix A.2, "Compiler Switches.")
4. Your interpreted Microsoft BASIC program uses one of the following commands or statements not supported by the compiler:

AUTO	DEF USR	LLIST	RENUM
BLOAD	DELETE	LOAD	SAVE
BSAVE	EDIT	MERGE	USR
CONT	LIST	NEW	

9.2 Reference Format

Each command, function, or statement in the alphabetical reference that follows is described using the following format:

Heading	Function
Syntax	Shows the correct syntax for the statement or function. There are two kinds of syntaxes: one for functions and one for statements. All functions return a value of a particular type and can be used wherever an expression can be used. Unlike functions, statements must appear alone 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.
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.

Note	Points out an important fact or feature. This is an optional section that does not appear with every reference entry.
Warning	Alerts the user to problems or dangers associated with the use of the given statement or function. This is an optional section that does not appear with every reference entry.

9.3 Reference Syntax Notation

The following syntax notation is used in the reference manual:

CAPITALS	Items in capital letters indicate BASIC keywords. These keywords must be part of the statement syntax, unless they are enclosed in brackets. In an actual program, you can enter these BASIC keywords in either uppercase (capital) letters or lowercase letters.
<i>Italics</i>	Items in italics represent information that you must supply; italics are also used occasionally in the text for emphasis.
[Brackets]	Items inside square brackets are optional.
Ellipses ...	Items followed by horizontal ellipses can be repeated. Vertical ellipses are used in syntax lines and program examples to show that a portion of the program has been omitted.
{ Braces }	Braces indicate that you have a choice between two or more items. You must choose one of them, unless all of the items are also enclosed in square brackets.
	Vertical bars separate the different choices inside braces.

You must enter all punctuation, including commas, parentheses, semicolons, hyphens, and equal signs, exactly as shown.

Example

In the following syntax line, "LOCK" and "TO" are BASIC keywords; you cannot leave them out of the syntax. However, you can omit the "#" symbol and everything inside the second set of brackets (provided you want to lock an entire file). If you want to lock just certain records, you have a choice: one record (*record*); or a range of records ([*start*] TO *end*). Note that if you specify a range, the starting record is optional. The *filenum*, the *record*, the *start*, and the *end* are all names that you must supply.

```
LOCK [#] filenum [, [record] [start] TO end]
```

9.4 Reference Input/Output Notation

Program examples, user input, and program output are shown throughout this chapter. Typically, program lines are shown first, followed by descriptive text showing program output. In cases where there is user input, it is shown in a different, darker type to distinguish it from the program listing or output from BASIC. The following example illustrates these conventions:

```
input x
print x "squared is" x^2
end
```

Output for this program, along with user input (231), follows:

```
? 231
231 squared is 53361
```

Syntax

CALL *name*[(*argument-list*)]

Action

Calls and transfers control to a subprogram.

Remarks

In programs compiled with the QuickBASIC Compiler, CALL can be used to invoke compiled BASIC subprograms. CALL can also be used the way it is used in the interpreter; namely, to call assembly-language subroutines.

The syntax for CALL statements is identical in interpreted and compiled programs. However, the parameters have slightly different meanings in the two programs.

In interpreted programs, *name* is a variable containing the segment offset that is the starting address in memory of the subroutine being called. In programs compiled with the QuickBASIC Compiler, *name* is simply the name of the subroutine being called. The QuickBASIC Compiler allows subroutine names up to 31 characters long. If CALL invokes an assembly-language subroutine, then the *name* must be a PUBLIC symbol in that subroutine.

For both interpreted and compiled programs, *argument-list* contains the variables or constants that CALL passes to the subroutine. In the compiler, variables or constants can also be passed to a compiled BASIC subroutine with the SHARED statement inside the subroutine, or with the SHARED attribute to COMMON, DIM, or REDIM in the main program.

Since the compiler allows strings of up to 32767 characters in length, the string descriptor requires 4 bytes, rather than 3 as in the interpreter. (The 4 bytes are low byte, high byte of the length, followed by low byte, high byte of the address.) If the called assembly-language subroutine uses string arguments, you may need to recode the subroutine to take this difference into account.

CALL Statement

See Also

Chapter 7, "Creating Structured Programs"

Chapter 8, "Using Assembly-Language Subroutines"

COMMON, DIM, REDIM, SHARED

Example

This program, "combine.bas", calls three BASIC subprograms, "split", "strip", and "printout". The split subprogram breaks the command line input (command\$) into two separate filenames and stores them in the array, "file\$()". Next, the strip subprogram strips leading blanks from the second filename. Finally, the printout subprogram writes out the contents of these two files. The DOS output redirection symbol (>) on the command line sends this output to a third file.

If two file names are not given on the command line, the result is the BASIC error "Illegal function call".

```
dim file$(2)
cmd$ = command$
call split(cmd$, file$())           'separate command line
call strip(file$(2))               'strip leading blanks
call printout(file$())             'send both files
end

sub split(c$, f$(1)) static
    mark = instr(c$, " ")           'find first blank
    f$(1) = left$(c$, mark - 1)     'everything up to first blank
    f$(2) = mid$(c$, mark + 1)      'everything after first blank
end sub

sub strip(f$) static
    first$ = left$(f$, 1)
    while first$ = " "
        lng = len(f$)               'look for first nonblank
        f$ = right$(f$, lng - 1)    'character in file$(2)
        first$ = left$(f$, 1)
    wend
end sub
```



```

sub printout(f$(1)) static
  for file% = 1 to 2
    open "1".#1, f$(file %)
    while not eof(1)
      line input #1, temp$
      print temp$
    wend
    close #1
  next
end sub

```

'loop executes twice:
 '1st time for file\$(1)
 '2nd time for file\$(2)
 'read file
 'write file to standard
 'output

Sample command line:

```
combine main.bas sub.bas > prog.bas
```

After this executes, the contents of "main.bas" and "sub.bas" will be combined and stored in "prog.bas".

CALLS Statement

Syntax

CALLS *name* [(*argument-list*)]

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 *argument-list*, while **CALL** passes unsegmented addresses.

See Also

CALL

SyntaxCHAIN "*filename*"**Action**

Transfers control from current program to another program

Remarks

The *filename* argument is a string expression that identifies the program to which control is passed.

The QuickBASIC Compiler does not support the ALL, MERGE, DELETE, or *line-number* CHAIN options available in the interpreter. For this reason, you should use COMMON to pass variables from one program to another. In addition, you should be careful when a chained-to program chains back to the chaining program; since you cannot specify a *line-number* 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.)

The BCOM10.LIB does not support the use of CHAIN with COMMON. Therefore, both the chaining and chained-to programs must use the default BRUN10.LIB; that is, the chaining and chained-to programs must be compiled *without* the /O switch.

When programs are compiled with BRUN10.LIB, files are left open during chaining.

See Also

CALL, COMMON

Example

This example converts a number in any base to a decimal number. The base and the number are input in the main program, "main.bas", which then chains to a second program, "digit.bas". This second program splits the number from the main program into separate digits, converts those string values to numeric values, and stores the numeric values in an array, a(). Control then chains to a third program, "dec.bas", which actually changes

the number to a decimal number, deallocates (with ERASE) the array in which the digits were 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 in all three programs to share variables.

```
rem ** This program is main.bas **
common a(1),n$,b,ln
input "base,number: " b,n$
print
while b
    ln = len(n$)
    dim a(ln)
    chain "digit"
wend
end
```

```
rem ** This program is digit.bas **
common a(1),n$,b,ln
m = ln - 1
for j = 0 to m
    a$ = mid$(n$,j+1,1)
    if a$ < "A" then a(m-j) = val(a$) -
    else a(m-j) = asc(a$) - 55
next
chain "dec"
```

```
rem ** This program is dec.bas **
common a(1),n$,b,ln
dec = 0
for i = 0 to (ln-1)
    dec = dec + a(i)*b^i
next
erase a
print "Decimal # = ":dec : print
print "Input 0 for base to end program."
chain "main"
```

Sample output:

base,number: 16,43E

Decimal # = 1086

Input 0 for base to end program.
base,number: 0,

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, including any optional parameters, after first stripping all leading blanks from the command line and converting all letters to uppercase (capital letters).

Example

The following program computes the logarithm to the base 10 of any number greater than zero that is entered after "log" (the name of the executable file) on the command line:

```
bs = 10
power = 1
sign = 1
lg = 0
x = val(command$)
test1: if x > 0 then goto test2 _
      else _
          print "log(";command$;) not defined."
          print "Input must be greater than zero."
          end
test2: if x >= 1 then goto test3 _
      else _
          x = 1/x
          sign = -1
test3: if x < 100 then goto main _
      else _
          while x >= 100
              x = x/10
              lg = lg + 1
          wend
```

COMMAND\$ Function

```
main:
  while abs(bs - 1) > .0000001
    if bs > x then goto newval _
    else _
      x = x/bs
      lg = lg + power
    newval:
      bs = sqr(bs)
      power = power/2
  wend
  print "log(";command$;") = ";lg*sign
```

Command line and output:

log 6.78

log 6.78 = .8312299

Command line and output:

log -1

log(-1) not defined.

Input must be greater than zero.

Syntax

COMMON [SHARED] [/blockname/] *variable-list*

Action

Passes variables to a chained program or subprogram

Remarks

This is a statement that requires modification of interpreted BASIC programs when used with the QuickBASIC Compiler.

With the BASIC interpreter, you may put COMMON statements anywhere in a program. With the compiler, however, the COMMON statement must appear in a program before any executable statements. All statements are executable, except the following:

- COMMON
- DEFtype
- DIM (for static arrays)
- OPTION BASE
- REM
- \$name (all compiler metacommands)

If you use static array variables in a COMMON statement, then you must declare them in a preceding DIM statement. If an array is dynamic, the DIM statement follows the COMMON statement, since a DIM statement is an executable statement when the array it dimensions is dynamic.

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 also need the SHARED attribute with COMMON when the subprogram being called is in a separately compiled module.

The *blockname* is any valid BASIC identifier up to 31 characters long. Use *blockname* when not every subprogram shares all variables in the *variable-list*. Items in a named COMMON statement are not preserved across a chain to a new program.

The *variable-list* is 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* array variables are specified by appending "()" to the variable name; *dynamic* array variables are specified by appending "{num}" to the variable name, where *num* is an integer constant indicating the number of dimensions in the array.

When you use COMMON with CHAIN, you must use the BRUN10.EXE runtime module (this means you should compile your program without the /O switch). Also, both the chaining program and the chained-to program require a COMMON statement. With the QuickBASIC Compiler, 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 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, the additional COMMON variables are initialized to zeros and null strings.

To ensure that programs can share common areas, place COMMON declarations in a single "include" file and use the \$INCLUDE statement in each program. (See Chapter 6, "Using Metacommands," for a discussion of the \$INCLUDE statement.)

See Also

Chapter 6, "Using Metacommands"

CHAIN, SHARED, SUB...END SUB

Example

This program fragment shows the use of an "include" file to share COMMON statements among programs:

```
rem ** This file is menu.bas **
rem $include:'comdef.bas'
.
.
.
chain "progl"
end

rem ** This file is progl.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 **
```

DEF FN Statement

Syntax 1

DEF FN*name*[(*parameter-list*)] = *function-definition*

Syntax 2

DEF FN*name*[(*parameter-list*)]

.

.

.

FN*name* = *expression*

.

.

END DEF

Action

Defines and names a function.

Remarks

The QuickBASIC Compiler supports both of the preceding syntaxes (single-line functions and multiline functions), whereas interpreted BASIC supports only the first (single-line functions).

The *name* must be a legal variable name. This name, preceded by FN, becomes the name of the function.

The *parameter-list* is a list of variable names, separated by commas. When the function is called, it replaces these variables on a one-to-one basis with the values the program supplies.

The *function-definition* in the first syntax 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 or the optional EXIT DEF statement.

Argument variables or values that appear in the function call replace the variables in the parameter list on a one-to-one basis.

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 *function-definition* 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 *function-definition* or *expression*.

Warning

Your program must define a function with a DEF FN statement before it can call the function. If your program calls a function before it is defined, an "Undefined user function" error occurs.

User-defined functions cannot appear inside other multiline functions, nor can they appear inside IF...THEN...ELSE, FOR...NEXT, or WHILE...WEND blocks.

Your program cannot contain recursive function definitions; that is, a function cannot be defined in terms of itself.

The RETURN statement is *not* equivalent to END DEF or EXIT DEF. Using a RETURN statement to exit a multiline function will cause a severe unrecoverable error.

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)
  pi = 3.14159263
  d = d + m/60 + s/3600
  fndegrad = d * (pi/180)
end def
deg = 45 : min = 10
print tab(5):"Angle measurement";tab(38);"SINE"
print
for sec = 10 to 50 step 10
  print tab(5) deg chr$(248) min "" sec chr$(34):
  rad = fndegrad(deg,min,sec)
  print tab(35) sin(rad)
next
end
```

Output:

Angle measurement	SINE
45° 10' 10"	.7091949
45° 10' 20"	.7092291
45° 10' 30"	.7092632
45° 10' 40"	.7092974
45° 10' 50"	.7093316

SyntaxDEFINT *letter-range*DEFSNG *letter-range*DEFDBL *letter-range*DEFSTR *letter-range***Action**

Declares variables as integer, single precision, double precision, or string

Remarks

The interpreter and the compiler process DEFtype 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, based on the DEFINT statement in line 10. 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  defang i : 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 DEFtype statement by directing program flow around it.

You can see these differences in the output from the program in the example in this section.

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, that is, 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, L, M, X, Y, or Z are designated integer variables:

```
defint K-M,X-Z
```

A type declaration character such as %, !, #, or \$ always takes precedence over a DEFtype statement.

Note

II, I#, I\$, and I% are all separate and distinct variables; each one can hold a different value. The effects of this are illustrated in the following example.

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 occurrences of "t" variables 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     defeng 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
```

DRAW Statement

Syntax

DRAW "*subcommand-string*"

Action

Draws an object defined by *subcommand-string*

Remarks

This statement requires modification of interpreted BASIC programs when used with the QuickBASIC 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. (See also the discussion under "X" + VARPTR\$(*string-expression*) below.)

The DRAW statement combines many of the capabilities of the other graphics statements into a graphics macro language, as described below under *Prefixed*, *Cursor Movement*, and *Other Commands*. The graphics macro language defines a set of characteristics that comprehensively describe a particular image. In this case, the characteristics include motion (up, down, left, right), color, angle rotation, and scale factor.

Each of the following subcommands 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.

Prefixed

The prefix commands listed below may precede any of the movement commands:

- | | |
|---|--|
| B | Move but don't plot any points |
| N | Move but return to original position when done |

Cursor Movement

The following commands specify movement in units. The default unit size is one point; this unit size can be modified by the *S* command. If no argument is supplied, the cursor is moved one unit.

U [n]	Move up scale factor n points
D [n]	Move down
L [n]	Move left
R [n]	Move right
E [n]	Move diagonally up and right
F [n]	Move diagonally down and right
G [n]	Move diagonally down and left
H [n]	Move diagonally up and left

Other Commands

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 the coordinates of the current graphics position and connected to that position with 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.
A n	Set angle n. The value of n may range from 0 to 3, where 0 is 0°, 1 is 90°, 2 is 180°, and 3 is 270°. Figures rotated 90° or 270° on a monitor screen with the standard aspect ratio of 4/3 are scaled so they will have the same size they had before they were rotated.
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="+varptrd(d)+"NUSO"
next d
```

C *n*

Set color *n*. The value of *n* can range from 0 to 3 in medium resolution (screen 1), and from 0 to 1 in high resolution (screen 2).

In medium resolution, *n* selects the color attribute from the current palette chosen in the COLOR statement; the default is Palette 1. (See Table 9.2.)

Table 9.2

Colors in Medium Resolution

Color Attribute	Palette 0	Palette 1
0	†	†
1	Green	Cyan
2	Red	Magenta
3	Brown	White

† Zero (0) is always the attribute for the current background.

S *n*

Set scale factor *n*, which may range from 1 to 255. The scale factor multiplied by the distances given with U, D, L, R, or relative M commands gives the actual distance traveled.

"X" + VARPTR\$(*string-expression*)

This powerful command allows you to execute a second substring from a string. You can have one string execute another, which executes a third, and so on.

Numeric arguments can be constants such as "123" or variable names.

The QuickBASIC 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 works with the interpreter, the second with the compiler.

```
draw "xa$"
```

```
draw "x" + varptr$(a$)
```

P *paintcolor*, *bordercolor* The *paintcolor* is the paint attribute for a figure's interior, while the *bordercolor* is the paint attribute for the figure's border. The colors selected by *paintcolor* and *bordercolor* depend on the palette chosen in the COLOR statement. (See Table 9.2.) "Tile" painting is not supported in DRAW.

Example

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
input "Press return to end".end$
```

DRAW Statement

Example

The following program displays a stylized clock face in the high-resolution graphics mode (screen 2); the clock shows the time returned by the `time$` function. This program should be compiled with the `/D` switch to enable Control-Break.

```
screen 2
loop = 1
while loop
  gosub split
  check = min
  while check = min
    gosub split
    gosub face
  wend
  cls
wend
end
split: rem ** This splits time$ into numeric values **
  let tm$ = time$
  hr$ = left$(tm$,2) : min$ = mid$(tm$,4,2)
  hr = val(hr$) : min = val(min$)
return
face: rem ** This draws the clock face **
  circle (320,100),175
  little = 360 - (30*hr + min/2)
  big = 360 - (6*min)
  draw "ta=" + varptr$(little) + "nu45"
  draw "ta=" + varptr$(big) + "nu70"
  locate 2,37 : print time$
  locate 23,24 : print "Press Ctrl-Break to return to system"
return
```

Syntax

END [**DEF** | **SUB**]

Action

Ends a BASIC program, multiline function definition, *or* subroutine

Remarks

END DEF ends a multiline function definition; you must use END DEF with DEF FN. END SUB ends a BASIC subroutine; you must use END SUB with SUB. END DEF and END SUB are supported only in the compiler.

The END statement in the compiler has the same effect as the sequence END:SYSTEM in the interpreter: it 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 may place END statements anywhere in the program to terminate its execution.

See Also

DEF FN, SUB...END SUB/EXIT SUB

Example

The following ends program execution if variable K's value is greater than 1000; otherwise, program execution continues at line "compute".

```
if K>1000 then end _  
    else goto compute
```

Syntax

ERASE *arrayname* [*arrayname*...]

Action

Resets the elements of static arrays; deallocates dynamic arrays

Remarks

In the QuickBASIC Compiler, ERASE has the same effect as in the interpreter; namely, it 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 without first erasing it, a "Duplicate definition" error will occur.

See Also

Section 6.6, "Dimensioned Array Allocation: \$STATIC and \$DYNAMIC"

CHAIN (Example), DIM, REDIM

Example

```
rem $dynamic
dim a(100,100)
.
.
.
erase a           ' This deallocates array a.
redim a(5,5)
rem $static
dim b(50,50)
.
.
.
erase b           ' This sets all elements of b equal to zero;
                  ' b still has the dimensions assigned to it
                  ' above.
```

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 QuickBASIC Compiler supports double-precision control values (*start*, *end*, and *counter*) in its FOR...NEXT loops.

The FOR statement uses *start*, *end*, and *increment* as fixed numeric expressions, and *counter* as a counter. The expression *start* is the initial value of the counter. The expression *end* is the final value of the counter. 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*, then BASIC branches back to the statement after the FOR statement and the process is repeated. If *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.

It is a good idea not to change the loop variable within the loop, since doing so can make the program more difficult to debug.

FOR...NEXT Statements

Nested Loops

You may nest FOR...NEXT loops; that is, you may 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 sequence of statements

```
next k
next j
next i
```

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.

Example 1

```

for i= 5 to 1 step -1
  for j= 1 to 1
    print " ";
  next j
  print
next i

```

Output:

```

*****
*****
***
**
*

```

Example 2

The following example prints a sine curve:

```

cls
locate 12,1
print string$(80,95)
for i! = 0 to 6.3 step .07875
  column% = 12.65*i! + 1
  row% = 11*(1 - sin(i!)) + 1
  locate row%,column%
  print chr$(222)
next

```

FRE Function

Syntax

`FRE(number)`

`FRE(string-exp)`

Action

Returns the size of free string space.

Remarks

This function has a different use in programs compiled with the Quick-BASIC Compiler, as opposed to interpreted BASIC programs.

In interpreted BASIC programs, FRE with a numeric argument returns the number of bytes of memory not being used by BASIC. In compiled programs, FRE with a numeric argument returns the size of the next free block of string space.

In both compiled and interpreted BASIC programs, FRE with a string argument returns the number of bytes in BASIC's memory space that are not being used; however, before FRE returns the number of free bytes, it firsts compacts free string space into a single block.

Example

```
print fre(0)
```

Sample output:

59530

The actual value returned by FRE on your computer may differ from this example.

Syntax

```
GOSUB { line-number1 | line-label1 }
:
:
RETURN [ { line-number2 | line-label2 } ]
```

Action

Branches to, and returns from, a subroutine

Remarks

In addition to the simple RETURN statement, the compiler supports RETURN *line-number2* (or RETURN *line-label2*). This allows a RETURN from a GOSUB statement 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, however, because any other GOSUBs, WHILEs, or FORs 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 *line-number2* option unless only one event can occur, because an event occurring during an event-trapping routine will leave confusing information on the stack, and this will result in undefined program behavior.

The first line of the subroutine is *line-number1* in the GOSUB statement.

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.

A subroutine may contain more than one RETURN statement. Simple RETURN statements (without the *line-number2* option) in a subroutine cause Microsoft QuickBASIC 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.

Note

The preceding discussion of subroutines applies only to the targets of GOSUB statements, *not* subroutines delimited by SUB...END SUB.

Example

This example computes arcsine values for arguments between -1 and 1. This 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
guess2 = pi/2 : guess1 = 3 * (pi/2)
while abs(guess1 - guess2) > .0000005
  gosub newval
wend
print "arcsin ":x:" = ":temp
end
```

```
newval:
  temp = (guess1 + guess2)/2
  if sin(temp) > x then guess2 = temp
  else guess1 = temp
return
```

Sample output:

```
sine? 3
Illegal sine value.
Sine must be >= -1 and <= 1.
sine? .43
arcsin .43 = 2.6971
```

That is, the angle whose sine is .43 has a measure of 2.6971 radians.

Syntax

GOTO [*linenumber*] [*linelabel*]

Action

Branches unconditionally to the line specified by *linenumber* or *linelabel*

Remarks

In the QuickBASIC Compiler, the GOTO statement allows branching to lines identified by either line numbers or line labels.

If the statement that has *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 at 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.

See Also

Section 3.3.4, "Line Numbers and Alphanumeric Labels"

GOTO Statement

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  
goto start
```

Sample output:

```
Input 0 to end.  
? 5  
Area = 78.5  
? 7  
Area = 153.86  
? 12  
Area = 452.16  
? 0
```

Syntax

LBOUND(*array* [, *dimension*])

Action

Returns the lower bound (smallest available subscript) for the indicated dimension of an array.

Remarks

This function is supported only by the compiler.

The argument *dimension* is 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, 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.

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

LBOUND Function

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
```


Syntax

```
LOCK [#] filename [, record | [start] TO end] ]
.
.
UNLOCK [#] filename [, record | [start] TO end] ]
```

Action

Controls access by other processes to all or part of an opened file

Remarks

These statements are supported only by the compiler and are used in networked environments where several users might be working simultaneously on the same file.

The *filename* is the number with which the file was opened.

If you specify just one record (*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*.

Note

Be sure to remove all locks with an UNLOCK statement before closing a file or terminating your program. Otherwise, undefined results will occur.

LOCK and UNLOCK must match exactly.

LOCK...UNLOCK Statements

Example 1

The following locks the entire file opened as number 2:

```
lock #2
```

The following locks only record 32 in file number 2:

```
lock #2, 32
```

The following locks records 1 through 32 in file number 2:

```
lock #2, to 32
```

The following locks records 9 through 32 in file number 2:

```
lock #2, 9 to 32
```

Example 2

The following UNLOCK would be legal:

```
lock #1, 1 to 4  
lock #1, 5 to 8  
unlock #1, 1 to 4  
unlock #1, 5 to 8
```

The following UNLOCK would be illegal, since the range in an UNLOCK statement must match the range in the corresponding LOCK statement exactly:

```
lock #1, 1 to 4  
lock #1, 5 to 8  
unlock #1, 1 to 8
```

Possible error messages:

```
Bad record number  
Permission denied
```

Example 3

The following fragment opens a file and allows an operator to lock an individual record before updating the information in that record. When the operator is done, the program unlocks the locked record. (Unlocking the locked records allows other people to work with the file.)

```
open "monitor" 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
```

PLAY Statement

Syntax

PLAY "*subcommand-string*"

Action

Plays music as specified by *subcommand-string*

Remarks

The *subcommand-string* is one or more of the subcommands listed below.

PLAY uses a concept similar to that in 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 programs compiled with the QuickBASIC Compiler, you should use the VARPTR\$(*variable*) form for variables. For example, you should change interpreter statements such as PLAY "XA\$" and PLAY "O=1" to PLAY "X" + VARPTR\$(A\$) and PLAY "O=" + VARPTR\$(I) in the compiler.

Change Octave

- > Increases octave by 1. Octave will not advance beyond 6.
- < Decreases octave by 1. Octave will not drop below 0.

Tone

- O *n* Sets the current octave. There are seven octaves, numbered 0 through 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 through 84 (in the seven possible octaves, there are 84 notes); *n* = 0 means a rest.

Duration

- 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 through 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

- P n** Specifies a pause, ranging from 1 to 64. This option corresponds to the length of each note, set with L *n*.
- T n** Sets the "tempo," or the number of L 4s in one minute. The range for *n* is 32 to 255. The default for *n* is 120.

Operation

- 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 number of notes that can be played in the background at one time varies according to the particular machine.

PLAY Statement

Substring

"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).

Suffizes

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.

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$(scales$)
for i = 1 to 6
  play ">x" + varptr$(scales$)
next
play "o6 x" + varptr$(scales$)
for i = 1 to 6
  play "<x" + varptr$(scales$)
next
```

Example 2

This example will play the beginning of the first movement of Beethoven's Fifth Symphony:

```
let listen$ = "t180 o2 p2 p8 L8 ggg L2 e-"  
let fate$ = "p24 p8 L8 fff L2 d"  
play listen$ + fate$
```

Syntax

REDIM [SHARED] *arrayname*(*subscripts*) [*arrayname* (*subscripts*)...]

Action

Changes the space allocated to an array that has been declared dynamic.

Remarks

The REDIM statement is supported only by the QuickBASIC Compiler.

The optional SHARED attribute in a REDIM statement in the main program allows you to pass variables among all the subprograms in a module. (See Section 7.4.1, "Passing Parameters with the SHARED Attribute.")

The argument *arrayname* is the name of the array you want to redimension.

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 the 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.

See Also

Section 3.3.6, "Dynamic and Static Arrays"

Section 6.6, "Dimensioned Array Allocation: \$STATIC and \$DYNAMIC"

Section 7.4.1, "Passing Parameters with the SHARED Attribute"

DIM, ERASE

Example

The following program generates "total" different random numbers in the specified range. The "redim range(m)" and "erase range" statements allow you to repeat the program with a different range and a different value for total.

```
rem $dynamic
r$ = "n"
while r$ = "n"
  input "Range (low,high): ", low, high
  input "Total to choose: ", total
  m = high - low + 1
  redim range(m)
  for k = 1 to m
    range(k) = k + low - 1
  next
  randomize timer/10
  for i = 1 to total
    x = int(m*rnd + 1)
    print range(x),
    range(x) = range(m)
    m = m - 1
  next
  print
  erase range
  input "End program (y or n)": r$
wend
end
```

Sample output:

Range (low,high): 50,100

Total to choose: 10

58	100	76	71	99
64	87	85	77	69

End program (y or n)? n

Range (low,high): -5,5

Total to choose: 4

-4	1	-3	5
----	---	----	---

End program (y or n)? y

RESUME Statement

Syntax

RESUME [*linenumber*] [*linelabel*]

RESUME [0]

RESUME NEXT

Action

Continues program execution after an error-recovery procedure has been performed.

Remarks

This is a statement that may require modification of interpreted BASIC programs when used with the QuickBASIC Compiler. 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.

Depending on where execution is to resume, any one of the three syntaxes shown above may be used, as shown 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 1.)
RESUME <i>linenumber</i>	At <i>linenumber</i> .
RESUME <i>linelabel</i>	At <i>linelabel</i> .

A RESUME statement that is not in an error-handling routine causes a "RESUME without error" message to be printed.

Note

Statements containing error-handling routines should be compiled with either the /X switch or the /E switch.

Considerable extra code is required to support both RESUME and RESUME NEXT. If you can rewrite your programs so they contain RESUME *linenumber* | *linelabel* | statements instead, you can compile with the /E switch, thus making user code significantly smaller.

See Also

Section 3.2.4, "Operational Differences"

Appendix A.2, "Compiler Switches"

Example 1

This example has an error-handling routine that starts at line 500. If "fntest" tries to evaluate the square root of a negative number, line 500 prints its error message. In the interpreter, control then resumes at line 150, and the FOR...NEXT loop continues. In the compiler, however, control resumes at line 110, which contains the function definition. This causes execution to go into an endless loop. In programs such as this, you should change RESUME NEXT statements to RESUME *linenumber* or RESUME *linelabel*.

```
100 on error goto 500
110 def fntest(a) = 1 - sqr(a)
120 for i = 4 to -2 step -1
130     print i, fntest(i)
140 next
150 end
500 print "No negative arguments"
510 resume next
```

Interpreter output:

```

4      -1
3      -.7320509
2      -.4142136
1      0
0      1
-1     No negative arguments
-2     No negative arguments

```

Compiler output:

```

4      -1
3      -.7320509
2      -.4142136
1      0
0      1
-1     No negative arguments
4      -1
3      -.7320509
2      -.4142136
.
.
.

```

Example 2

This example has an error-handling routine that starts at line "errorhandle". If the error is the one that the user has defined as 200 (file is empty), then the routine asks for a new filename for input, and resumes execution at line "continue". If the error is something else, BASIC prints the message "Unprintable error".

```

let condition% = 0
on error goto errorhandle
open "phone" for input as #1
if lof(1) = 0 then error 200
continue:
.
.
.
end
errorhandler: rem ** Error handling fragment **
rem ** Next line prints "Unprintable error" if true **
if err <> 200 then error err _
else
rem ** Routine for error 200 **
print "Phone number file is empty"
input "Name of file that has information": fil$
close #1
open fil$ for input as #1
resume continue

```

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.

See Also

GOSUB

RUN Statement

Syntax

RUN [*linenumber*]

RUN "*filename*"

Action

Either restarts the program currently in memory, or executes the program specified by *filename*

Remarks

The first syntax restarts the program currently in memory. If *linenumber* is specified, execution begins on that line. Otherwise, execution begins at the lowest line number, or the first executable line, if you use line labels.

With the second form of the syntax, the named file is loaded from a device into memory and run. If there is a program in memory when the command executes, the original program is no longer in memory.

In the second syntax, the *filename* must be that used when the file was saved. This string expression can contain a path.

The default filename extension in the compiler is ".EXE", just as the default extension in the interpreter is ".BAS". Therefore, if "catchall" is the name of both an executable file and a BASIC source file, the command

```
run "catchall"
```

will work correctly in both the interpreter and the compiler.

RUN closes all open files and clears the current contents of memory before loading the designated program. The QuickBASIC 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 QuickBASIC compiler or by other languages; unlike interpreted programs, compiled programs cannot directly execute ".BAS" source files.

See Also

CHAIN

Example 1

The following program line transfers program control to the file "price", after closing all open files and deleting the current memory contents:

```
run "price"
```

Example 2

The following command runs the program currently in memory, starting with line 950:

```
run 950
```

SHARED Statement

Syntax

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

Action

Within a single module, gives a subprogram access to variables declared in the main program without having to pass them as parameters

Remarks

This statement is supported only by the compiler.

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.

The SHARED statement must appear only within a named subroutine; if SHARED occurs outside a subroutine, a subroutine (SB) error occurs. (See "SUB...END SUB Statement".)

See Also

COMMON, DIM, SUB...END SUB

Chapter 7, "Creating Structured Programs"

Example

The following subprogram example ("convert") changes any positive decimal integer to its string representation in another base. The variables d, b, and n\$ are shared with the main program.

```

loop = 1
while loop
  n$ = ""
  print
  input "Decimal number":d
  input "New base":b
  print : print d:" base 10 equals ":
  while d
    call convert
  wend
  print n$:" base ":b : print
  input "Convert another":r$
  c$ = left$(r$,1)
  if (c$ = "Y" or c$ = "y") _
    then loop = 1 _
    else loop = 0
wend
end

sub convert static
  shared d,b,n$
  r = d mod b
  d = d\b
  if r > 9 then goto letter _
  else _
    dgt$ = str$(r)
    ln = len(dgt$) - 1
    n$ = right$(dgt$,ln) + n$
  exit sub
letter:
  dgt$ = chr$(65 + r - 10)
  n$ = dgt$ + n$
end sub

```

Sample output:

```

Decimal number? 88
New base? 3

```

```

88 base 10 equals 10021 base 3

```

```

Convert another? n

```

STATIC Statement

Syntax

STATIC *variable* [, *variable*...]

Action

Designates simple variables or arrays as local to a subprogram or function definition, and preserves their values when the subprogram is exited and then reentered.

Remarks

This statement is supported only by the compiler.

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 "SUB...END SUB" and "DEF FN").

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

Don't confuse the **STATIC** statement with the **STATIC** that is part of the **SUB...END SUB** syntax or with the **\$STATIC** metacommmand. The **STATIC** attribute in **SUB...END SUB** shows that the subroutine is non-recursive, while **\$STATIC** statically allocates memory for arrays.

See Also

Chapter 6, "Using Metacommands"

DEF FN, SHARED, SUB...END SUB

Example 1

This example contrasts the **STATIC** and **SHARED** statements within a subroutine, "augment". The variables "r" and "R" are both local to this subroutine, while the variables "rep" and "num" are shared between the main program and the subroutine. The "static r,n" statement ensures that r and n retain the last values assigned to them each time the main program calls the subroutine.

```
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
```

STATIC Statement

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
n = 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$
    else f2$ = left$(f1$,mark - 1)
    f2$ = f2$ + ".new"
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 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 every occurrence of the string "Chapter 1" replaced by "Introduction".

SUB...END SUB/EXIT SUB Statements

Syntax

```
SUB global-name [(parameter-list)] STATIC
```

```
.
```

```
.
```

```
.
```

```
[EXIT SUB]
```

```
.
```

```
.
```

```
END SUB
```

Action

Marks the beginning and end of a subprogram

Remarks

This statement is supported only by the compiler.

The *global-name* is a variable name up to 31 characters long. This name cannot appear in any other SUB statement within the same program module.

The optional *parameter-list* can contain the names of simple variables and arrays passed to the subprogram by a CALL statement in the main 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.

STATIC 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, which in turn calls the first subprogram. This version of QuickBASIC supports only nonrecursive subprograms, and generates a warning error (ST) if you omit STATIC.

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.

SUB and END SUB mark, respectively, the beginning and end of a subprogram. You can also use the EXIT SUB statement to exit a particular subprogram. Both END SUB and EXIT SUB return program control to the calling program.

Subprograms are called by a CALL statement. When a subprogram is exited and later reentered, 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 reentry 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.

See Also

Chapter 7, "Creating Structured Programs"

CALL, SHARED, STATIC

Example

In this example, the main program calls a subprogram, "filesearch", 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 filesearch(test$,p$)
wend

sub filesearch(test$,p$) static
    static num
    num = num + 1
    x = instr(test$,p$)
    if x = 0 then exit sub _
    else _
        print "Line #":num;" ":test$
    end sub
```

SUB...END SUB/EXIT SUB Statements

Possible input:

File to be searched? search.bas
Pattern to search for? sub

Output:

```
Line # 9: sub filesearch(test$,p$) static
Line # 13:   if x = 0 then exit sub _
Line # 16: end sub
```


Syntax

UBOUND(*array* [, *dimension*])

Action

Returns the upper bound (largest available subscript) for the indicated dimension of an array.

Remarks

This function is supported only by the compiler.

The argument *dimension* is 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, UBOUND(ACCOUNT,1) finds the largest subscript in dimension A, 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
```

Syntax

UNLOCK [#] *filenum* [, *record*] [*start*] TO *end*]

Action

Releases access restrictions applied to specified portions of a file

Remarks

This function is supported only by the compiler.

The UNLOCK statement should be used only after a LOCK statement. See "LOCK" for examples and a complete discussion.

See Also

LOCK

WIDTH Statement

Syntax

WIDTH [*filenumber* | *device*] , *size*

Action

Assigns an output line width to a file or device

Remarks

The QuickBASIC Compiler implements an enhanced version of the interpreter's WIDTH statement in which files as well as devices can be assigned an output line width. The possible syntaxes of this statement are explained in the following list.

Syntax	Action
WIDTH <i>size</i>	Sets the width of the terminal screen to <i>size</i> . The maximum width is 255.
WIDTH <i>filenumber</i> , <i>size</i>	Sets the width of a file opened to an output device (e.g., LST or CON) to <i>size</i> . The <i>filenumber</i> 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 <i>device</i> , <i>size</i>	Sets the width of <i>device</i> (a device filename) to <i>size</i> . The <i>device</i> should be a string value enclosed in double quotes; for example, "CONS". Note that this width assignment is <i>deferred</i> until the next OPEN statement affecting the device; the assignment does not affect output for an already open file.

Example 1

In the following example, the width of the lineprinter is set to 120 columns:

```
width "lpt1:",120
```

Example 2

In the following example, the record width in file #1 is set to 40 columns:

```
width #1,40
```

Example 3

In the following example, file #1 is the console display:

```
open "scrn:" for output as #1
test$ = "1234567890"
width #1,2
    print #1, test$
width #1,4
    locate 7,1
    print #1, test$
```

Output:

```
12
34
56
78
90

1234
5678
90
```


Appendixes

A	Summary of Commands	173
B	Questions Most Frequently Asked	177
C	Listing File Formats	181
D	Using the Input Line Editor	187
E	Microsoft QuickBASIC Reserved Words	189
F	Error Messages	191

Appendix A

Summary of Commands

- A.1 Microsoft QuickBASIC Metacommands 175
- A.2 Compiler Switches 176

A.1 Microsoft QuickBASIC Metacommands

Name	Metacommand
\$DYNAMIC	Causes dynamic allocation of arrays.
\$INCLUDE:'file'	Switches compilation from the current source file to <i>file</i> .
\$LINESIZE: <i>size</i>	Sets the width of the source code listing, in columns.
\$LIST [+ -]	Turns on or off source listing. Errors are always listed.
\$MODULE:'name'	Changes an internal module name passed to Microsoft LINK.
\$OCODE [+ -]	Turns on or off listing of disassembled object code.
\$PAGE	Skips to next page. Page line number is reset.
\$PAGEIF: <i>number</i>	Skips to next page if there are <i>number</i> lines or less left on the listing page.
\$PAGESIZE: <i>size</i>	Sets length of source listing in lines.
\$SKIP [: <i>number</i>]	Skips <i>number</i> lines or to end of page.
\$STATIC	Causes static allocation of arrays that are later dimensioned.
\$SUBTITLE:'title'	Sets source listing page subtitle.
\$TITLE:'title'	Sets the source listing page title.

A.2 Compiler Switches

Using the prompt method, you can place compiler switches after the filename response to any prompt, before you press Return. Using the command-line method, you can place compiler switches anywhere on the command line after the name of the source file. Switches are always preceded by a forward slash (/).

Switch	Function
/A	Includes a listing of the disassembled object code in the source listing
/D	Generates debugging code for run-time error checking and enables Control-Break
/E	Indicates the presence of ON ERROR GOTO with RESUME <i>linenumber</i> statement
/O	Substitutes the BCOM10.LIB run-time library for BRUN10.LIB
/R	Stores arrays in row order
/S	Writes quoted strings to .OBJ file instead of symbol table
/V	Enables event trapping for communications (COM), lightpen (PEN), joystick (STRIG), the timer (TIMER), and function keys (KEY). Checks between statements for occurrence of an event
/W	Enables event trapping for same statements as /V, but checks between lines for occurrence of an event
/X	Indicates presence of ON ERROR GOTO with RESUME, RESUME NEXT, or RESUME 0

Appendix B

Questions Most Frequently Asked

There are questions about Microsoft BASIC that are asked more frequently than others. This appendix includes answers to these questions.

How do I do random file I/O?

Random files are not stored in ASCII format, so the methods for getting data from them and putting data in them are not the same as ASCII-format sequential files. To create a random file, first give it a name and file size by using the OPEN statement. The next statement should be a FIELD statement that describes the order and size of the buffer variables. Each of these buffer variables is a string variable, regardless of whether the data that will go in it is string data or numerical data.

You must never alter these buffer variables in a program. They are used to load and unload data from the files. To load the values of working program variables into these buffer variables, use LSET or RSET, not LET or "=".

To convert numeric program variables into strings that can be put into the buffer variables, change integer values to string values with the \$MKI statement, change single-precision numbers into string values with the MKS\$ statement, and change double-precision numbers to string values with the MKD\$ statement. The following is an example of this process:

```
lset a$ = mks$(assets).
```

At this point, the value of the numeric variable is in string form and stored in the data file buffer. Use the PUT statement to place the information from the buffer in the file. The file will then contain the information.

Remember, in random files, if you only write to records 1 and 3, record 2 will have undefined data in it since you have not written to it yet. Nonsense exists there from previous disk use. You must keep track of what records have and have not been written to in order to avoid reading nonsense from a record to which nothing has yet been written.

You don't have to close and then reopen a random file to get information back out of the file. If, however, you want to open a random file to get information out of it, use the OPEN statement, define the FIELDS for the buffer variables, and use the GET statement to load the right record into the data buffer. Again, you cannot use these buffer variables in your program. To reference them, you must transfer them to working program variables. In addition, if the actual information is not string information, you'll need to convert the variables from the string format of the buffer variable to what you need. To do this, use CVI, CVS, and CVD statements. If the data in the buffer is going to be a string in your program, you don't need to convert the data. For example:

```
let company$ = a$
let debt = cvd(b$)
```

To close a random file, use the CLOSE statement.

Because of the way operating systems interact with software, EOF, LOC, and LOF will not necessarily work with random files. These functions are most useful, however, with sequential files.

How do you FIELD a random file record when the list of buffer variables exceeds the length of a legal program line?

When the list of buffer variables is long enough to exceed a legal BASIC line, use multiple FIELD statements next to each other. In the first FIELD statement, deal with the first part of the record. In the second FIELD statement, refer to the entire range of records in the first field statement as one buffer variable. Then continue your naming of variables. For example:

```
120 open "r", 4, "account.dat", 143
140 field 4, 21 as company$, 8 as accountno$, 4 as
    a$, 4 as b$, 4 as c$, 4 as d$, 4 as e$, 2 as f$,
    4 as g$, 21 as street$, 10 as street2$
160 field 4, 86 as ignore$, 14 as h$, 14 as i$, 9
    as j$, 2 as k$, 2 as l$, 2 as m$, 2 as n$, 2 as
    o$, 2 as p$, 2 as q$, 2 as r$, 2 as s$, 2 as t$
```

In the above example, a random file, ACCOUNT.DAT, is opened. The first field statement describes the first 76 characters in the record. The second field statement refers to all the information referred to in the first as IGNORE\$. The individual buffer variables in the first statement can still be accessed by the names given in the first field statement. The second field statement goes on to describe the rest of the buffer variables in the file.

How do I read what I've written in my sequential file?

If you already have the file opened for either Output or Append, you must first close the file, and then reopen it for Input. In other words, when you use sequential access, you can have a sequential file opened for Input only or for Output only, but never for both at the same time.

What might cause a "String Space Corrupt" error?

The three most frequent causes of that error in the order of their frequency are as follows:

1. Stack overflow, for reasons such as the following:
 - GOSUBs without RETURNS
 - Illegal exits out of FOR...NEXT loops
 - Redefinition, by means of DEF *type* statements, of variables previously defined in a COMMON statement
2. Using POKE to write into the wrong place in an improperly written assembly-language routine
3. A string array subscript that is not within the stated dimension of the array

Appendix C

Listing File Formats

- C.1 Source Listing 183
- C.2 Disassembly Listing 184
- C.3 Linker Listing 185

This appendix contains sample listing files for the following program:

```
for a=1 to 10
  print "Hello, world!"
next a
```

C.1 Source Listing

The source listing gives you the hexadecimal address of each line in the program relative to the start of the .EXE file, the hexadecimal offset from the start of the data segment for any data values generated by the line, the program line itself, and any warning and error messages generated during compilation.

You can control the format of the listing file using the metacommands described in Chapter 6, "Using Metacommands."

Offset Data Source Line Microsoft QuickBASIC Compiler V1.00

```
0030 0006 for a=1 to 10
0043 0006      print "Hello, world!"
0043 0006 next a
0068 000A
006B 000A
```

```
50448 Bytes Available
50170 Bytes Free
```

```
0 Warning Error(s)
0 Severe Error(s)
```

C.2 Disassembly Listing

In addition to the information contained in a source listing file, a disassembly listing gives you the assembly-language code that corresponds to your program:

Offset Data Source Line Microsoft QuickBASIC Compiler V1.00

```

0030 0006 rem %code+
0030 0006 for a=1 to 10
0030 ** CALL $INI
0035 ** CALL $560
003A ** 100001: MOV SI,OFFSET <const>
003D ** INT 3FH
003F ** DB 6FH
0040 ** JMP 100002
0043 0006 print "Hello, world!"
0043 0006 next a
0043 ** 100003: INT 3FH
0045 ** DB 0BCH
0046 ** MOV BX,OFFSET <const>
0049 ** INT 3FH
004B ** DB 6EH
004C ** INT 3EH
004E ** DB 79H
004F ** MOV DI,OFFSET <const>
0052 ** MOV SI,OFFSET A!
0055 ** INT 3FH
0057 ** DB 7FH
0058 ** 100002: MOV DI,OFFSET A!
005B ** INT 3FH
005D ** DB 7DH
005E ** MOV SI,DI
0060 ** MOV DI,OFFSET <const>
0063 ** INT 3FH
0065 ** DB 9EH
0066 ** JNA $-25H
0068 000A
0068 ** INT 3EH
0068 ** DB 02H
0068 000A

```

50448 Bytes Available

50170 Bytes Free

0 Warning Error(s)

0 Severe Error(s)

C.3 Linker Listing

The linker listing file gives the starting and ending addresses, length, name, and class of each segment. The information in the "Start" and "Stop" columns shows the address (in hexadecimal) of each segment relative to the program's entry point in memory. The "Name" column gives the name of the segment, and the "Class" column gives information about the segment type. For example, a CODE segment contains program code, and a DATA segment contains data.

The starting address and the name of each group are listed near the end of the file. The program entry point is listed at the bottom of the file.

Start	Stop	Length	Name	Class
00000H	0006AH	0006BH	SAMPLE_CODE	BC_CODE
00070H	00105H	00096H	CSEC	CODESG
00110H	00110H	00000H	SHELL	CODESG
00110H	00110H	00000H	CODE	CODE
00110H	0012FH	00020H	BC_INC_CODE	INIT_CODE
00130H	00132H	00003H	BC_IDS_CODE	INIT_CODE
00140H	00496H	00357H	INIT_CODE	INIT_CODE
004A0H	0184FH	013B0H	DSEG	DATASG
01850H	018B7H	00068H	RIMLOAD	DATASG
018B8H	018DEH	00028H	PSEUDOCOMMON	DATASG
018EOH	018EOH	00000H	COMMON	BLANK
018EGH	018EOH	00000H	CONST	CONST
018EOH	018EOH	00000H	DATA	DATA
018EOH	018E9H	0000AH	BC_DATA	BC_VARS
018EAH	018EAH	00000H	BC_FT	BC_SEGS
018FOH	019CFH	00020H	BC_CN	BC_SEGS
01910H	01912H	00003H	BC_DS	BC_SEGS
01920H	01B1FH	00200H	STACK	STACK

Origin	Group
004A:0	DGROUP

Program entry point at 0000:0030

Appendix D

Using the Input Line Editor

When a program asks for input, you can respond by entering a line up to 255 characters long. Using the input line editor, you can edit this line at any point before you press Return to return the string to the program. The input line editor allows you to move the cursor to any position between the prompt and the rightmost character on the screen; you can insert or overwrite characters at any of these cursor positions. Table D.1 summarizes the input line editor commands:

Table D.1

Input Line Editor Commands

Function	Key
Toggles between overwrite and insert mode. Cursor is an underline for overwrite mode and half-block for insert mode.	Control-R or Insert
Inserts/overwrites (mode determined by Control-R) to next tab field.	Control-I
Toggles function key label display.	Control-T
Returns string to program.	Control-M or Return
Exits from program to MS-DOS.	Control-C
Moves cursor left one word.	Control-B
Moves cursor right one word.	Control-F
Moves cursor left one character.	Control-[or left arrow
Moves cursor right one character.	Control-] or right arrow
Moves cursor to beginning of line.	Control-K
Moves cursor to end of line.	Control-N
Deletes character over cursor.	DEL
Deletes entire line.	Control-U
Deletes character left of cursor.	Control-H or Backspace
Deletes characters right of cursor.	Control-E

Appendix E

Microsoft QuickBASIC Reserved Words

The following is a list of reserved words used in Microsoft QuickBASIC:

ABS	CONT	ERDEV	KEY
ACCESS	COS	ERDEV\$	KILL
AND	CSNG	ERL	LBOUND†
APPEND†	CSRLIN	ERR	LCOPY
ASC	CVD	ERROR	LEFT\$
AS†	CVI	EXIT†	LEN
ATN	CVS	EXP	LET
AUTO	DATA	FIELD	LINE
BEEP	DATE\$	FILES	LIST
BLOAD	DEF FN	FIX	LLIST
BSAVE	DEF USR	FOR	LOAD
CALL	DEFDBL	FRE	LOC
CALLS	DEFINT	GET	LOCAL†
CDBL	DEFSNG	GO	LOCATE
CHAIN	DEFSTR	GOSUB	LOCK†
CHDIR	DELETE	GOTO	LOF
CHR\$	DIM	HEX\$	LOG
CINT	DRAW	IF	LPOS
CIRCLE	EDIT	IMP	LPRINT
CLEAR	ELSE	INKEY\$	LSET
CLOSE	END	INP	MERGE
CLS	ENVIRON	INPUT	MID\$
COLOR	ENVIRON\$	INPUT#	MKD\$
COM	EOF	INPUT\$	MKDIR
COMMAND\$†	EQV	INSTR	MKI\$
COMMON	ERASE	INT	MKS\$

† These words are reserved in the compiler only, and may be used in the interpreter.

MOD	PRESET	SOUND	USR
MOTOR	PRINT	SPACE	USR0
NAME	PRINT# USING	SPACE\$	USR1
NEW	PSET	SPC	USR2
NEXT	PUT	SQR	USR3
NOISE	RANDOMIZE	STATIC†	USR4
NOT	READ	STEP	USR5
NULL†	REDIM†	STICK	USR6
OCT‡	REM	STOP	USR7
ON	RENUM	STR\$	USR8
OPEN	RESET	STRIG	USR9
OPEN COM	RESTORE	STRING\$	VAL
OPTION	RESUME	SUB†	VARPTR
OR	RETURN	SWAP	VARPTR\$
OUT	RIGHT\$	SYSTEM	VIEW
OUTPUT†	RMDIR	TAB	WAIT
PAINT	RND	TAN	WEND
PALETTE	RSET	THEN	WHILE
PALETTE USING	RUN	TIME\$	WIDTH
PEEK	SAVE	TIMER	WINDOW
PEN	SCREEN	TO	WRITE
PLAY	SEG†	TROFF	WRITE#
PMAP	SGN	TRON	XOR
POINT	SHARED†	UBOUND†	
POKE	SHELL	UNLOCK†	
POS	SIN	USING	

† These words are reserved in the compiler only, and may be used in the interpreter.

Appendix F

Error Messages

F.1	Invocation Errors	193
F.2	Compile-Time Errors	194
F.3	Run-Time Errors	200
F.4	Microsoft LINK Errors	206

During development of a Microsoft BASIC program with the Microsoft QuickBASIC Compiler, four different kinds of errors may occur:

1. Invocation errors
2. Compile-time errors and warnings
3. Microsoft LINK errors
4. Run-time errors

Each type of error is associated with a particular step in the program development process. Invocation errors occur when you invoke the compiler; compile-time errors and warnings occur during compilation; Microsoft LINK errors occur while you are linking your program; and run-time errors occur when the compiled Microsoft QuickBASIC program is running.

This appendix lists error codes and error messages for each type of error, along with any error numbers that are assigned.

F.1 Invocation Errors

Invocation errors occur when you enter illegal input on the command line or in response to prompts during invocation. The messages that may occur when the compiler is invoked are listed below:

Bad filename

Improper file specification entered.

Bad switch: /s

Illegal compiler switch *s*.

Can't create file

Disk is write protected or disk is full.

Command error: 'c'

An error has occurred at the character specified by the character *c*.

Disk *drivename* full

The disk in the drive *drivename* is full. If no *drivename* appears, the disk in the default drive is full.

File not found

The file does not exist on the specified disk.

F.2 Compile-Time Errors

When errors occur while your program is compiling, the compiler displays the line containing the error, and a two-character code for the error. An arrow beneath that line points to the place in the line where the error occurred. In some cases, the compiler reads ahead on a line to determine whether an error has really occurred, and the arrow will point a few characters beyond where the error actually took place.

The messages listed below describe both severe errors and warning errors. When a severe error occurs, the compiler will attempt to continue. However, the resulting object file will not be correct. You must correct the error and recompile the source file before linking.

On the other hand, when a warning error occurs, compilation continues, but warning errors are displayed to point out poorly constructed program statements. The resulting object file can be linked to form an executable program, but the program may not perform as intended. Errors and warnings are indicated either by a long message or by a two-letter code. Long error messages describe general conditions that are not associated with a particular line number. Two-letter codes indicate errors on specific lines.

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.

BS

Bad subscript. The following situations cause this error:

- Illegal array dimension value
- Wrong number of subscripts

CD

Duplicate COMMON variable.

CN

COMMON array not dimensioned.

CO

COMMON out of order. COMMON must appear before any executable statements.

DD

Array already dimensioned. This error can be caused by the following:

- More than one DIM statement for same array
- DIM statement after initial use of array
- OPTION BASE after array dimensioned

FD

Function definition error. This error occurs when a previously defined function is redefined or when DEF...EXIT DEF/END DEF statements are incorrectly nested.

FN

FOR...NEXT error. This can be caused by the following conditions:

- FOR loop index variable already in use
- FOR without NEXT
- NEXT without FOR

IN

\$INCLUDE file not found.

Internal error

An internal error has occurred in the Microsoft QuickBASIC Compiler. Please note precisely what actions preceded this message and report the problem to Microsoft immediately.

Line *label* is undefined

A statement refers to a nonexistent line label.

Line *n* is undefined

A statement refers to a nonexistent line number.

LL

Line too long. Lines are limited to 254 characters.

LS

String constant too long. Strings are limited to 32767 characters.

MC

Warning error. A metacommand is incorrect.

Memory overflow

Available memory has been exhausted. Try compiling with the /S switch or without any of the debug switches. If memory is still exhausted, break your program into parts and use the CHAIN command or separate compilation facilities.

Missing NEXT for variable

No NEXT was found to match a FOR statement.

ND

Warning error. An array is declared but not dimensioned.

OM

Out of memory. This error can be caused by the following conditions:

- Array too big
- Data-memory overflow
- Too many statement numbers
- Program-memory overflow

OV

Math overflow. The result of a calculation is too large to be represented in BASIC number format.

SB

This is a subprogram definition error and is usually caused by one of the following:

- The subprogram is already defined, or a subprogram of that name is already defined.
- The program contains incorrectly nested SUB...EXIT SUB/END SUB statements.

SI

Warning error. Statement ignored. This warning often results when an unimplemented command is used in a program.

SN

Syntax error. This error can be caused by the following conditions:

- Illegal argument name
- Illegal assignment target
- Illegal constant format
- Illegal debug request
- Illegal DEF xxx character specification
- Illegal expression syntax
- Illegal function name
- Illegal function formal parameter
- Illegal separator
- Illegal format for statement number
- Invalid character
- Missing AS
- Missing equal sign
- Missing GOTO or GOSUB
- Missing comma
- Missing INPUT
- Missing line number

- Missing left parenthesis
- Missing minus sign
- Missing operand in expression
- Missing right parenthesis
- Missing semicolon
- Name too long
- Expected GOTO or GOSUB
- String assignment required
- String expression required
- String variable required
- Illegal syntax
- Variable required
- Wrong number of arguments
- Formal parameters not unique
- Single variable only allowed
- Missing TO
- Illegal FOR loop index variable
- Illegal COMMON name
- Missing THEN
- Missing BASE
- Illegal subroutine name

SQ

This is a sequence error and is usually caused by one of the following conditions:

- Duplicate statement number
- Statement out of sequence

ST

Warning error. There is a missing STATIC or SUB statement.

TC

This message is caused by one of the following conditions:

- Expression too complex
- Too many arguments in function call (limit of 60)
- Too many dimensions (limit of 255)
- Too many variables for LINE INPUT (limit of 1)
- Too many variables for INPUT (limit of 60)

TM

Type mismatches are caused by the following conditions:

- Data type conflict
- Variables of different types

UC

Unrecognizable command.

UF

Function not defined. Functions must be defined before they are used.

WE

This is a WHILE...WEND error, caused by either a WHILE statement without a corresponding WEND, or a WEND statement without a corresponding WHILE.

/O

This message is caused by division by zero, division of the integer -32768 by 1 or -1, or moding of the integer -32768 by 1 or -1.

/E

Missing "/E" switch. Programs that contain ON ERROR GOTO or ON event GOTO statements must be compiled with the /E switch.

/X

Missing "/X" switch. Programs that contain RESUME, RESUME NEXT, and RESUME 0 statements must be compiled with the /X switch.

F.3 Run-Time Errors

The coded errors listed below may occur while the program is running. The compiler run-time system prints long error messages followed by an address, unless a /D, /E, or /X switch is specified in the compiler 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 the error messages are as follows:

Error *X* in module *zzzzzzzz* at address *nnnn:nnnn*.

and

Error *X* in line *yyyy* of module *zzzzzzzz* at address *nnnn:nnnn*.

A description of uncoded run-time error messages follows this list.

Code	Message
2	Syntax Error A line is encountered that contains an incorrect sequence of characters in a DATA statement.
3	RETURN without GOSUB A RETURN statement is encountered for which there is no previous, unmatched GOSUB statement.
4	Out of Data A READ statement is executed when there are no DATA statements with unread data remaining in the program.
5	Illegal Function Call A parameter that is out of range is passed to a math or string function. A function call error may also occur for the following reasons: <ul style="list-style-type: none"> • A negative or unreasonably large subscript is used. • A negative number is raised to a power that is not an integer.

Code	Message
	<ul style="list-style-type: none"> • A USR function is called that has an undefined starting address. • A negative record number is given when using GET <i>file</i> or PUT <i>file</i>. • An improper or out-of-range argument is given to a function. • Strings are concatenated to create a string greater than 32767 characters in length.
6	<p>Floating Overflow or Integer Overflow</p> <p>The result of a calculation is too large to be represented within the range allowed for floating-point numbers.</p>
7	<p>Out of memory</p> <p>Not enough memory is available to allocate a file buffer.</p>
9	<p>Subscript Out of Range</p> <p>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 /D switch was specified at compile time.</p>
10	<p>Redimensioned Array</p> <p>A second DIM statement was executed for an already dimensioned dynamic array.</p>
11	<p>Division by Zero</p> <p>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 -32768 is divided by 1 or -1, or if -32768 is moded by 1 or -1.</p>
13	<p>Type mismatch</p> <p>The argument types are not compatible.</p>

Code	Message
14	Out of String Space String variables exceed the allocated amount of string space.
16	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.
19	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.
20	RESUME without Error A RESUME statement is encountered before an error-trapping routine is entered.
50	Field Overflow A FIELD statement is attempting to allocate more bytes than were specified for the record length of a random file.
51	Internal Error An internal malfunction occurred in the Microsoft Quick-BASIC Compiler. Report to Microsoft the conditions under which the message appeared.
52	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.
53	File Not Found A KILL, NAMES, FILES, or OPEN statement references a file that does not exist on the current disk.

Code	Message
54	<p>Bad File Mode</p> <p>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.</p>
55	<p>File Already Open</p> <p>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.</p>
57	<p>Disk I/O Error</p> <p>An I/O error occurred on a disk I/O operation. The operating system cannot recover from the error.</p>
58	<p>File Already Exists</p> <p>The filename specified in a NAME statement is identical to a filename already in use on the disk.</p>
61	<p>Disk Full</p> <p>All disk space has been allocated.</p>
62	<p>Input Past End</p> <p>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.</p>
63	<p>Bad Record Number</p> <p>In a PUT or GET statement, the record number is equal to zero.</p>
64	<p>Bad File Name</p> <p>An illegal form is used for the filename with LOAD, SAVE, KILL, or OPEN (e.g., a filename with too many characters).</p>
67	<p>Too Many Files</p> <p>The 255-file directory maximum is exceeded by an attempt to create a new file with SAVE or OPEN statements.</p>

Code	Message
68	Device unavailable The device you are attempting to access is not on-line or does not exist.
70	Permission Denied An attempt was made to write to a write-protected disk.
71	Disk not ready The disk-drive door is open, or no disk is in the drive.
72	Disk media error Disk-drive hardware has detected a physical flaw on the disk.
74	Rename across disks An attempt was made to rename a file with a new drive designation. This is not allowed.
75	Path/file access error During an OPEN, MKDIR, CHDIR, or RMDIR operation, MS-DOS was unable to make a correct path-to-filename connection. The operation is not completed.
76	Path not Found During an OPEN, MKDIR, CHDIR, or RMDIR operation, MS-DOS was unable to find the path specified. The operation is not completed.

The following is a list of uncoded run-time error messages:

Cannot find BRUN10.EXE Enter new path:

This message appears if the run-time module is not found. "Enter new path" prompts for input of the correct drive letter. This error is severe and cannot be trapped.

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 MS-DOS.

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.

No Line Number in *modnam* at address *nnnn:nnnn*

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 65527. 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.

Program too large

Not enough memory is available to load BRUN10.EXE. This error is severe and cannot be trapped.

String Space Corrupt in [*line number of*] *modnam* at address *nnnn:nnnn*

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.

String Space Corrupt during G.C. in [*line number of*] *modnam* at address *nnnn:nnnn*

This error occurs when an invalid string in string space is being deleted during garbage collection. The probable causes for either of the "String Space Corrupt" errors are as follows:

- A string descriptor or string backpointer 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 /D switch may be used to ensure that array subscripts do not 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.

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.

Wrong version of runtime module

This error occurs when the version of the BRUN10.EXE run-time module being used does not match the version of the Microsoft QuickBASIC Compiler used to generate the code. This error is severe and cannot be trapped.

You cannot run BASIC as a Child of BASIC

During initialization, BASIC discovers that it is being run as a child program. This is caused by a SHELL *prog* statement, where *prog* is also a QuickBASIC program. It is not permitted to SHELL to another QuickBASIC program.

F.4 Microsoft LINK Errors

A listing of Microsoft LINK error messages may be found in the manuals supplied with your MS-DOS software. For your convenience, we have also listed them in this manual.

All errors cause the link session to terminate. No executable file is created. After you have corrected the error, you must relink your object files.

The following error messages are displayed by Microsoft LINK:

Attempt to access data outside of segment bounds,
possibly bad object module

There is probably a bad object file.

Bad numeric parameter

A numeric value is not in digits.

Invalid object module

An object module(s) is incorrectly formed or incomplete (as when assembly is stopped in the middle of the module).

Symbol defined more than once

Microsoft LINK found two or more modules that define a single symbol name.

Program size or number of segments exceeds capacity of linker

The total size may not exceed 384K, and the number of segments may not exceed 255.

Requested stack size exceeds 64K

You specified a size greater than or equal to 64K with the STACK switch.

Segment size exceeds 64K

The addressing system limit is 64K.

Symbol table capacity exceeded

Very many and/or very long names were entered, exceeding the limit of approximately 25K.

Too many external symbols in one module

The limit is 256 external symbols per module.

Too many groups

The limit is 10 groups.

Too many libraries specified

The limit is eight libraries.

Too many PUBLIC symbols

The limit is 1024 PUBLIC symbols.

Too many segments or classes

The limit is 256 (segments and classes combined).

Unresolved externals:*list*

The external symbols listed have no defining module among the modules of library files specified.

VM read error

This is a disk error; it is not caused by Microsoft LINK.

Warning: No stack segment

None of the object modules specified contains a statement allocating stack space, but the user typed the STACK switch.

Warning: Segment of absolute or unknown type

There is a bad object module, or an attempt has been made to link modules that Microsoft LINK cannot handle (e.g., an absolute object module).

Write error in TMP file

No more disk space remains to expand the VM.TMP file.

Write error on run file

This usually indicates there is not enough disk space for the run file.

Glossary

The definitions in this glossary are intended for use with this manual. Neither the individual definitions nor the list of terms is comprehensive.

Baseline

The portion of the filename that precedes the filename extension. For example, SAMPLE is the baseline of the file SAMPLE.BAS.

Call by reference

See "Pass by reference."

Call by value

See "Pass by value."

Compile time

The time during which the compiler is executing, compiling a Microsoft BASIC source file, and creating a relocatable object file.

Compiler

A set of programs that translates BASIC programs into a language understood by the computer.

Disassembly listing

A listing that shows assembled code, instructions, and addresses relative to the start of the program or module.

Double precision

A real value that is allocated 8 bytes of memory.

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.

Heap

An area of random access memory that is used by BASIC as the data management area. Variables and arrays are stored here.

Library

A directory that stores related modules of compiled code.

Link map file

A file that shows the address of every code and data segment in a program, relative to the start of the program.

Link time

The time during which Microsoft LINK is executing, and during which it collects and links relocatable object files and library files. *See also* Compile time, 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.

Machine code

Instructions that a microprocessor can execute.

Main program

In a program that calls subroutines, the calling program is the main program.

Memory map

A representation of where in memory the computer expects to find certain types of information.

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. There are several types of modules, including relocatable and executable modules. The compiler creates relocatable modules that are later processed by Microsoft LINK. Your final executable program is an executable module.

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.

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 Microsoft LINK 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 Microsoft BASIC language. A library routine usually corresponds to a feature or subfeature of the Microsoft BASIC language.

Run time

The time during which a compiled and linked 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 Microsoft BASIC language. It is a peculiarity of the run-time module that it is an executable .EXE file. The run-time module is named BRUN10.EXE and is, for the most part, a library of routines. It is made executable so that you can see the version number.

Subprogram

A separately compilable module of code delimited by the SUB and END SUB or EXIT SUB statements. *See also* Main program.

Unbound

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 from the link command line the name of the module containing the desired symbol.

Index

+, linker line extension character, 57

/A SAVE option, 25

/A switch, 48-49

Alphanumeric line labels. *See* Line labels, alphanumeric

Arithmetic overflow check, 47

Arrays

 bounds check, 48

 chained programs, use with
 COMMON, 32

 COMMON, use with, 32

 dynamic

 common, use with, 32

 defined, 30

 DIM, use with, 69

 dimensioning, 30

 ERASE, 130

 ERASE, effect, 31

 ERASE statement, 30

 memory space allocation, 30

 REDIM statement, 30, 148

 REDIM, use with, 69

 redimensioning, 30

 elements, passing with CALL, 77

 LBOUND function, 139

 lower-bound function, 77

 memory allocation, 69

 passing to assembly-language
 subroutines, 88

 passing with CALL, 77

 static

 bounds checking, 32

 COMMON, use with, 32

 defined, 30

 dimensioning, 30

 ERASE, 130

 ERASE, effect, 31

 ERASE, use with, 69

 REDIM, effect, 31

 REDIM, use with, 69

 storing, 50

Arrays (*continued*)

 subprograms

 parameter list form, 75

 preserving values, 82

 STATIC statement, 82

 unallocated, default to static, 32

 upper-bound function, 77

 Assembly code switch, /A, 49

 Assembly-language subroutines

 arguments

 location, calculating, 89

 referencing, 89

 arrays, passing, 88

 CALL statement, 87

 converting interpreted programs, 87

 linking, 55

 naming requirements, 88

 using, 87

AUTO, 25

AUX, 42

Auxiliary communications port device
 name, 42

Backing up disks, 11

Bascom command, 41, 45

Basename, defined, 209

BASIC run-time errors, 200

BASCOMEXE, purpose, 12

BCOM10.LIB

 advantages, 60

 invocation, 60

 linking with, 49

 purpose, 12

 use with CHAIN and COMMON, 53

BIN directory, 15

Binary file, executing within a
 program, 61

BLOAD, 25

Bounds checking, arrays, 32

BRUN10.EXE

 linking with, 53

 purpose, 12

- BRUN10.LIB
 - advantages, 59
 - invocation, 60
 - linking with, 49, 53
 - purpose, 12
- BSAVE, 25
- CALL statement
 - assembly-language subroutines, use with, 87
 - BASIC subprograms, 75
 - interpreter statement, modification required, 25
 - passing array elements, 77
 - passing parameters, 74
 - subprograms, use with, 84
- CALLS statement, 25, 110
- CHAIN
 - arrays, use with, 32
 - BASIC programs only, 61
 - COMMON, use with, 115
 - subprograms, use with, 84
 - syntax, 111
 - use with BCOM10.LIB, 53, 59
 - use with BRUN10.LIB, 59
- Character. *See* specific character
- COM, 46
- COMMAND\$ Function, 113
- Command link, 58
- Command syntax conventions, 6
- Commands
 - Bascom, 41, 45
 - link, 54
 - MS-DOS
 - diskcopy, 11
 - set, 13
 - prohibited
 - AUTO, 25
 - BLOAD, 25
 - BSAVE, 25
 - CONT, 25
 - DEF USR, 25
 - DELETE, 25
 - EDIT, 25
 - LIST, 25
 - LLIST, 25
 - LOAD, 25
 - MERGE, 25
 - NEW, 25
 - prohibited (*continued*)
 - RENUM, 25
 - SAVE, 25
 - USR, 25
 - SAVE, 25
 - COMMON
 - arrays, use with, 32
 - described, 27
 - DIM, restriction with, 30
 - dynamic arrays, use with, 32
 - order of variables, 116
 - SHARED attribute, 79, 80
 - static arrays, use with, 32
 - subprogram parameter passing, 80
 - syntax, 115
 - use with BCOM10.LIB, 53, 59
 - use with BRUN10.LIB, 59
- Communications port device name, 42
- Compile time
 - defined, 209
 - error messages, 195
- Compiler
 - interpreter compared, 23
 - invoking, 45
 - prompts, 41
 - setup
 - hard disk, 14
 - two floppy drives, 13
 - severe errors, 194
 - starting, 41
 - switches, 46
- Compiling, sample session, 15
- CON, 42, 50
- CONST, 87
- CONT, 25
- Control-Break, 48
- Converting programs, 24
- Creating structured programs, 74
- CSEG, 87
- /D switch
 - error handling, 37, 47-48
 - static array bounds checking, 32
- DATA, 87
- Debugging, 38
- Debugging messages, 47
- DEF FN statement, 27, 34, 118
- DEF FN...END DEF block, multiline
 - function prohibited in, 35

- Device names, 42
- DIM
 - CHAIN, use with, 32
 - COMMON, restriction with, 30
 - dynamic arrays, 30
 - dynamic arrays, use with, 69
 - SHARED attribute, 79
 - static arrays, 30
 - usage, 30
- Disabling metacommands, 65
- Disassembled object code listing, 48, 66
- Disk drive device name, 42
- DISKCOPY command, 11
- Disks, backing up, 11
- DRAW statement, 124
- /DSALLOCATE, use prohibited, 58
- DSEG, 87
- \$DYNAMIC
 - See also* Metacommands
 - array memory allocation, 30, 69
- Dynamic arrays. *See* Arrays
- /E switch, 37, 47
- EDIT, 25
- END, 27
- END DEF, 34
- End-of-line character, 28
- END statement, 129, 135
- END SUB, 74
- END, use with \$INCLUDE, 69
- Environment variables
 - creating, 13
 - defined, 13
 - hard disk, 15
 - LIB, 13
 - PATH, 13
 - pathnames, use in, 13
 - two floppy drives, 14
- ERASE
 - deallocating with, 30
 - dynamic array, effect on, 31
 - statement, 130
 - static array
 - effect on, 31
 - use with, 69
- ERL, 37
- ERR, 37
- ERROR, 37
- Error handling
 - statements, functions, 37
- Error handling (*continued*)
 - switches, 37, 47
- Error messages, 131
- Error trapping, 29, 150
- Errors
 - run-time, 200
 - severe, 194
- Event trapping
 - between lines, /W, 47
 - between statements, /V, 46
- EXE file, creating, 53
- Executable files
 - creating, 24, 53
 - naming convention, 54
- Executing a program, 61
- Execution time. *See* Program execution
- EXIT DEF, 35
- EXIT SUB, 74
- Expressions, passing to subprograms, 78
- External symbol, 209
- FAC, 87
- File access
 - LOCK statement, 141
 - UNLOCK statement, 141
- Filename conventions, 42, 54
- Filenames
 - device names, 42
 - null file, 42
- Files
 - backing up, 11
 - BASCOM.EXE purpose, 12
 - BCOM10.LIB purpose, 12
 - BRUN10.EXE purpose, 12
 - BRUN10.LIB purpose, 12
 - compiler, linker, setting up, 14
 - device names, 42
 - .EXE, creating, 53
 - executable, creating, 53
 - executable. *See* Executable file
 - extensions, 42
 - library, 12
 - locked, releasing, 34
 - naming conventions, 42
 - NUL.LST, 45
 - NUL.MAP, 56
 - object. *See* Object file
 - README.DOC purpose, 12
 - restricting access, 33

- Files (continued)
 - object. *See* Object file
 - README.DOC purpose, 12
 - restricting access, 33
 - source, format, 25
- FOR...NEXT statements, 131
- FRCINT, 87
- FRE function, 27, 134
- Functions
 - ERL, 37
 - ERR, 37
 - multiline, 34-35
 - user-defined, 119
- GOSUB statement, 28, 135
- GOTO statement
 - described, 28
 - \$INCLUDE, use with, 69
 - subroutines, use with, 135
 - syntax, 137
- Graphics macro language, 124
- Heap, 210
- /HIGH, use prohibited, 58
- IF...THEN...ELSE block, multiline
 - function prohibited, 35
- IF...THEN, line numbers required, 29
- \$INCLUDE, 69
- Input/Output, using the console, 50
- Internal module names, changing, 70
- Interpreted programs, converting for the compiler, 87
- Interpreter, compiler compared, 23
- Interpreter, debugging with, 38
- Invocation command line, 113
- Invocation errors, 193
- Invoking the compiler, 41, 45
- Invoking the linker, 54
- KEY, 46
- LBOUND, 77
- LBOUND function, 139
- LIB directory, 15
- LIB environment variable, 13
- LIB variable
 - hard disk, 15
 - two floppy drives, 14
- Libraries, described, 53
- Library files
 - described, 12
 - naming convention, 54
- Library of subprograms, creating, 74
- Line 0, effect on error trapping, 29
- Line labels
 - alphanumeric, 28-29
 - RESUME statement, 30
- Line length restrictions, 28
- Line number check, 48
- Line numbers, 28-30
- Linefeed character, 28
- \$LINESIZE, 67
- Link command, 54, 58
- Link time, 210
- Linker
 - default file extensions, 54
 - invoking, 54
 - library search order, 57
 - /MAP switch, 56
 - prompts:
 - default responses, 57
 - defaults, 58
 - extending lines, 57
 - line extension character, 57
 - list file, 56
 - listed, 54
 - map file, 54
 - multiple filenames, 55
 - run file, 55
 - separating entries, 57
 - stopping, 58
- Linking
 - assembler routines, 55
 - choosing a library, 53, 56
 - described, 24
 - sample session, 15
 - with BCOM10.LIB, 60
 - with BCOM10.LIB, /O switch, 49
 - with BRUN10.LIB, 59
- LIST, 25
- \$LIST, 66
- Listings. *See* specific listing
- LLIST, 25
- LOAD, 25

- LOCK, 33
- LOCK statement, 141
- Loops, 131
- LST, 42
- Machine language translation, 23
- Main program, defined, 73
- MAKINT, 87
- MERGE, 25
- Metacommands
 - See also specific metacommand*
 - defined, 65
 - disabling, 65
 - \$DYNAMIC, 30, 69
 - \$INCLUDE, 69
 - \$LINESIZE, 67
 - \$LIST, 66
 - listing format, 67
 - \$MODULE, 70
 - \$OCODE, 66
 - \$PAGE, 67
 - \$PAGEIF, 67
 - \$PAGESIZE, 67
 - purpose, 65
 - \$STATIC, 30, 69
 - \$SUBTITLE, 67
 - syntax, 65
 - \$TITLE, 67
- Microsoft LINK. *See* Linking
- Modular programming, defined, 73
- \$MODULE metacommand, 70
- Module, relocatable, 211
- MS-DOS procedures, 11
- Multiline functions
 - definition location, 34-35
 - prohibited where, 35
 - recursive prohibited, 35
 - requirements, 35
 - run-time features, nesting, 37
- Networked systems, file access control, 33
- NEW, 25
- Notational conventions, 6
- NUL, 42
- NUL.LST file, 45
- NUL.MAP, 56
- /O switch, 49, 60
- Object file
 - created by compiler, 24
 - naming conventions, 42, 54
- Object filename
 - extension, 43
 - prompt, 41, 43, 45
- \$OCODE, 66
- ON ERROR GOTO
 - exception to interpreter, 37
 - line 0, effect, 29
 - /X switch, 47
- ON ERROR GOTO...RESUME, /E
 - switch, 47
- Optimization, 24
- \$PAGE, 67
- \$PAGEIF, 67
- \$PAGESIZE, 67
- Parameter passing
 - multiple source files, 80
 - single module
 - SHARED attribute, 79
 - SHARED statement, 79
- Pass by value, 211
- Passing parameters, SHARED
 - statement, 80
- Passing variables, COMMON, 115
- PATH variable
 - hard disk, 15
 - two floppy drives, 14
- Pathnames
 - use in environment variables, 13
 - use with linker prompts, 55
 - use with object filename prompt, 43
 - use with source filename prompt, 43
- PEN, 46
- PLAY statement, 144
- Practice session, 15
- Preliminary procedures
 - file backup, 11
 - outline, 11
- Printer device name, 42
- PRN, 42
- Program execution, speeding up, 24
- Program, running, 61
- Program termination, 129
- Programs
 - converting interpreted to compiled, 24

- Programs (*continued*)
 - modular, creating, 74
 - structured, creating, 74. *See also* structured programs
 - subprograms. *See* Subprograms
- Prompts
 - compiler. *See* Compiler
 - default responses, 45
 - linker. *See* Linker prompts, 54
 - object filename, 41, 43, 45
 - source filename, 41, 43, 54
 - source listing, 41, 44, 45
- /R switch, 50
- README.DOC, purpose, 12
- Records
 - locked, releasing, 34
 - restricting access, 33
- Recursive function definitions, 35
- REDIM
 - CHAIN, use with, 32
 - dynamic arrays, use with, 69
 - effect on static arrays, 31
 - SHARED attribute, 79
 - statement, 148
 - use with static arrays, 69
- Relocatable, 211
- RENUM, 25
- Reserved words, 189
- RESUME
 - line 0, effect, 29
 - line labels, numbers, use with, 30, 37
 - syntax, 150
- /X switch, 47
- RETURN, 28
- Return character, hiding, 28
- RETURN statement
 - check for GOSUB, 48
 - multiline function, use in, 36
 - syntax, 135
- Routine, 211
- RUN, 61
- Running a program, 19, 24, 61
- Run-time
 - defined, 212
 - environment, 61
 - error messages, 200
 - library, purpose, 24
- Run-time (*continued*)
 - module
 - See also* BRUN10.EXE
 - loading, 61
 - search order, 61
 - /S switch, 49
 - Sample compilation, 15
 - SAVE, 25
 - Saving a program for compilation, 25
 - Screen device name, 42
 - SET command, environment variable
 - definition, 13
 - SHARED attribute, COMMON, 80
 - SHARED statement, form, 80
 - \$SKIP, 67
 - Source files
 - format, 25
 - \$INCLUDE, use restrictions, 69
 - naming conventions, 42
 - Source filenames
 - extension, 43
 - input from keyboard, 50
 - prompt, 41, 43, 45
 - prompt, USER response, 50
 - Source listing
 - contents, 44
 - file-naming convention, 42
 - filename, extension, 44
 - format metacommands, 67
 - output to screen, 50
 - prompt, 41, 44, 45
 - prompt, USER response, 50
 - turning off, 66
 - Starting the compiler, 41, 45
 - Starting the linker, 54
 - Statement execution
 - compiler, 23
 - interpreter, 23
 - Statement, ERROR, 37
 - Statements
 - DEF FN, 34
 - END DEF, 34
 - enhanced
 - COMMON, 27
 - DEF FN, 27
 - END, 27
 - FRE, 27

enhanced (continued)

GOSUB, 28
 GOTO, 28
 RETURN, 28
 WIDTH, 28

EXIT DEF, 35

LOCK, 33

modification required

CALL, 25
 CALLS, 25
 RUN, 25

ON ERROR GOTO, 37

RESUME, 37

RETURN, 36

UNLOCK, 33-34

\$STATIC, 30, 69

STATIC, 75

Static arrays. *See* Arrays

STATIC statement, 81, 82

\$STATIC

See also Metacommands

array memory allocation, 69

metacommand, 69

STOP statement, 135

Stopping the linker, 58

STRIG, 46

String space compaction, 134

Strings, writing to disk, 49

Structured programs

See also Subprograms

COMMON, use in, 80

creating, 74

defined, 73

multiple source files

COMMON, 80

passing parameters, 80

single module

passing parameters, 78

SHARED attribute, 79

SHARED statement, 79

Subprograms

See also Structured programs

argument, parameter list errors, 82

array values, preserving, 82

CALL, 107

CALL, use instead of CHAIN, 84

CALLS, 110

CHAIN, 111

COMMON, 115

defined, 73

Subprograms (continued)

format, 74

invoking with CALL, 75

library of, creating, 74

parameter passing, 74

passing

arrays, 75

expressions, 78

variables, 76

prohibited expressions, 75

STATIC statement, 81

using, 73

variable aliasing errors, 82

Subroutines, 135

SUB...SUB END block, multiline

function prohibited, 35

SUB

formal parameter list, 75

subprogram delimiter, 74

syntax, 74

\$SUBTITLE, 67

Switches

/A switch, 48-49

array storage, /R, 50

command line, use with, 46

/D, 32, 37, 47-48

/E, 37, 47

error handling. *See* Error handling

switches

linker, 58

/O, 49

prompts, use with, 46

/R, 50

/S, 49

/V, 46

/W, 47

/X, 37, 47

TIMER, 46

\$TITLE, 67

TROFF, /D switch required, 47

TRON, /D switch required, 47

UBOUND, 77

Unbound, 212

UNLOCK, 33-34

UNLOCK statement, 141

USER, 42, 50

User-defined functions. *See* Multiline
functions
USER, use with CON, 50
USR, 25

/V switch, 46

Variables

data type, 121

passing to subprograms

by reference, 76

multiple source file, 81

single module, 79, 80

SHARED attribute, 79

SHARED statement, 80

STATIC, 81

subprogram aliasing errors, 82

subprogram, preserving, 81

/W switch, 47

WHILE...WEND block, multiline

function prohibited, 35

WIDTH, 28

/X switch, 37, 47

Documentation Report Form

Please tell Microsoft what you think about the documentation that accompanies the software. Your comments and suggestions help us improve our products. Mail this questionnaire to:

MICROSOFT Corporation
10700 Northup Way, Building #4
Box 97200
Bellevue, WA 98009
Attn: User Education, Systems Languages

Respond to any or all of the following questions. If you want to make additional comments, use the back of this page or a separate page.

Product Name:

Version Number:

1. Did you find errors in the documentation? Please give the document title, page number, and a description of the error.
2. Which parts of the documentation do you consider most important? Do you have any suggestions for improving these parts?
3. Is it easy to find the information you need? Is anything missing?
4. Is the documentation clear and easy to read?
5. What type of user are you?
 - ☐ First-time programmer
 - ☐ First-time programmer in this language, but experienced in at least one other language
 - ☐ Experienced programmer

Name _____
Street _____
City _____ State _____ Zip _____
Phone _____ Date _____

Instructions

Use this form to report software bugs, documentation errors, or suggested enhancements. Mail the form to Microsoft.

Category

_____ Software Problem _____ Documentation Problem
_____ Software Enhancement (Document # _____)
_____ Other

Software Description

Microsoft Product _____
Rev. _____ Registration # 007114-100
Operating System _____
Rev. _____ Supplier _____
Other Software Used _____
Rev. _____ Supplier _____

Hardware Description

Manufacturer _____ CPU _____ Memory _____ KB
Disk Size _____ " Density: _____ Sides: _____
Single Single

Problem Description

(Describe the problem, show describe how to reproduce it, and your diagnosis and suggested correction.) Attach a listing if available.

Microsoft Use Only

Self-Support _____

Date Received _____

Routing Code _____

Date Resolved _____

Report Number _____

Action Taken _____

CLAIMER OF THIS AGREEMENT, AND FURTHER INCLUDES LIMITED WARRANTY. MICROSOFT SOFTWARE LICENSE, SOFTWARE LICENSE, DISCLAIMER OF WARRANTY, AND HARDWARE LIMITED WARRANTY (collectively the "Agreement"). THIS AGREEMENT CONSTITUTES THE COMPLETE AGREEMENT BETWEEN YOU AND MICROSOFT CORPORATION. IF YOU DO NOT AGREE TO THE TERMS OF THIS AGREEMENT, DO NOT OPEN THE DISK PACKAGE. PROMPTLY RETURN THE UNOPENED DISK PACKAGE AND THE OTHER ITEMS (INCLUDING WRITTEN MATERIALS, BINDERS OR OTHER CONTAINERS, AND HARDWARE, IF ANY) THAT ARE PART OF THIS PRODUCT TO THE PLACE WHERE YOU OBTAINED THEM FOR A FULL REFUND.

Microsoft License Agreement

MICROSOFT SOFTWARE LICENSE

1. **GRANT OF LICENSE.** In consideration of payment of the LICENSE fee, which is a part of the price you paid for this product, Microsoft, as Licensor, grants to you, the LICENSEE, a nonexclusive right to use and display this copy of a Microsoft software program, (hereinafter the "SOFTWARE") on a single COMPUTER (i.e., with a single CPU) at a single location. If the single computer on which you use the SOFTWARE is a multiuser system, the License covers all users on that single system. Microsoft reserves all rights not expressly granted to LICENSEE.

2. **OWNERSHIP OF SOFTWARE.** As the LICENSEE, you own the magnetic or other physical media on which the SOFTWARE is originally or subsequently recorded or fixed, but Microsoft retains title and ownership of the SOFTWARE recorded on the original disk copy(ies) and all subsequent copies of the SOFTWARE, regardless of the form or media in or on which the original and other copies may exist. This License is not a sale of the original SOFTWARE or any copy.

3. **COPY RESTRICTIONS.** This SOFTWARE and the accompanying written materials are copyrighted. Unauthorized copying of the SOFTWARE, including SOFTWARE that has been modified, merged, or included with other software, or of the written materials is expressly forbidden. You may be held legally responsible for any copyright infringement that is caused or encouraged by your failure to abide by the terms of this License. Subject to these restrictions, and if the SOFTWARE is not copy-protected, you may make one (1) copy of the SOFTWARE solely for backup purposes. You must reproduce and include the copyright notice on the backup copy.

4. **USE RESTRICTIONS.** As the LICENSEE, you may physically transfer the SOFTWARE from one computer to another provided that the SOFTWARE is used on only one computer at a time. You may not electronically transfer the SOFTWARE from one computer to another over a network. You may not distribute copies of the SOFTWARE or accompanying written materials to others. You may not modify, adapt, translate, reverse engineer, decompile, disassemble, or create derivative works based on the SOFTWARE. You may not modify, adapt, translate, or create derivative works based on the written materials without the prior written consent of Microsoft.

5. **TRANSFER RESTRICTIONS.** This SOFTWARE is licensed only to you, the LICENSEE, and may not be transferred to anyone without the prior written consent of Microsoft. Any authorized transferee of the SOFTWARE shall be bound by the terms and conditions of this Agreement. In no event may you transfer, assign, rent, lease, sell, or otherwise dispose of the SOFTWARE on a temporary or permanent basis except as expressly provided herein.

6. **TERMINATION.** This License is effective until terminated. This License will terminate automatically without notice from Microsoft if you fail to comply with any provision of this License. Upon termination you shall destroy the written materials and all copies of the SOFTWARE, including modified copies, if any.

7. **UPDATE POLICY.** Microsoft may create, from time to time, updated versions of the SOFTWARE. At its option, Microsoft will make such updates available to the LICENSEE and transferees who have paid the update fee and returned the Registration Card to Microsoft.

8. **MISCELLANEOUS.** This Agreement is governed by the laws of the State of Washington.

DISCLAIMER OF WARRANTY AND LIMITED WARRANTY

THE SOFTWARE AND ACCOMPANYING WRITTEN MATERIALS (INCLUDING INSTRUCTIONS FOR USE) ARE PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND. FURTHER, MICROSOFT DOES NOT WARRANT, GUARANTEE, OR MAKE ANY REPRESENTATIONS REGARDING THE USE, OR THE RESULTS OF THE USE, OF THE SOFTWARE OR WRITTEN MATERIALS IN TERMS OF CORRECTNESS, ACCURACY, RELIABILITY, CURRENTNESS, OR OTHERWISE. THE ENTIRE RISK AS TO THE RESULTS AND PERFORMANCE OF THE SOFTWARE IS ASSUMED BY YOU. IF THE SOFTWARE OR WRITTEN MATERIALS ARE DEFECTIVE YOU, AND NOT MICROSOFT OR ITS DEALERS, DISTRIBUTORS, AGENTS, OR EMPLOYEES, ASSUME THE ENTIRE COST OF ALL NECESSARY SERVICING, REPAIR, OR CORRECTION.

Microsoft warrants to the original LICENSEE that (a) the disk(s) on which the SOFTWARE is recorded is free from defects in materials and workmanship under normal use and service for a period of ninety (90) days from the date of delivery as evidenced by a copy of the receipt and (b) the hardware accompanying the SOFTWARE is free from defects in materials and workmanship under normal use and service for a period of one (1) year from the date of delivery as evidenced by a copy of the receipt. Further, Microsoft hereby limits the duration of any implied warranty(ies) on the disk or such hardware to the respective periods stated above. Some states do not allow limitations on duration of an implied warranty, so the above limitation may not apply to you.

Microsoft's entire liability and your exclusive remedy as to the disk(s) or hardware shall be, at Microsoft's option, either (a) replacement of the disk or hardware or (b) replacement of the disk or hardware that does not meet Microsoft's Limited Warranty and which is returned to Microsoft with a copy of the receipt. If failure of the disk or hardware has resulted from accident, abuse, or misapplication, Microsoft shall have no responsibility to replace the disk or hardware or refund the purchase price. Any replacement disk or hardware will be warranted for the remainder of the original warranty period or thirty (30) days, whichever is longer.

THE ABOVE ARE THE ONLY WARRANTIES OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE, THAT ARE MADE BY MICROSOFT ON THIS MICROSOFT PRODUCT. NO ORAL OR WRITTEN INFORMATION OR ADVICE GIVEN BY MICROSOFT, ITS DEALERS, DISTRIBUTORS, AGENTS, OR EMPLOYEES SHALL CREATE A WARRANTY OR IN ANY WAY INCREASE THE SCOPE OF THIS WARRANTY, AND YOU MAY NOT RELY ON ANY SUCH INFORMATION OR ADVICE. THIS WARRANTY GIVES YOU SPECIFIC LEGAL RIGHTS. YOU MAY HAVE OTHER RIGHTS, WHICH VARY FROM STATE TO STATE.

NEITHER MICROSOFT NOR ANYONE ELSE WHO HAS BEEN INVOLVED IN THE CREATION, PRODUCTION, OR DELIVERY OF THIS PRODUCT SHALL BE LIABLE FOR ANY DIRECT, INDIRECT, CONSEQUENTIAL, OR INCIDENTAL DAMAGES INCLUDING DAMAGES FOR LOSS OF BUSINESS PROFITS, BUSINESS INTERRUPTION, LOSS OF BUSINESS INFORMATION, AND THE LIKE) ARISING OUT OF THE USE OF OR INABILITY TO USE SUCH PRODUCT EVEN IF MICROSOFT HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. BECAUSE SOME STATES DO NOT ALLOW THE EXCLUSION OR LIMITATION OF LIABILITY FOR CONSEQUENTIAL OR INCIDENTAL DAMAGES, THE ABOVE LIMITATION MAY NOT APPLY TO YOU. This Disclaimer of Warranty and Limited Warranty is governed by the laws of the State of Washington.

This Disclaimer of Warranty and Limited Warranty is governed by the laws of the State of Washington.

U.S. GOVERNMENT RESTRICTED RIGHTS

The SOFTWARE and documentation is provided with RESTRICTED RIGHTS. Use, duplication, or disclosure by the Government is subject to restrictions as set forth in subdivision (b)(3)(ii) of The Rights in Technical Data and Computer Software clause at FAR 25.227-7013. Contractor/manufacture is Microsoft Corporation 9700 Northup Way/Box 97200/Redmond, WA 98009.

For more information concerning this Agreement, or if you desire to contact Microsoft for any reason, please contact us at 1-800-485-2048 or write to Microsoft Corporation, 9700 Northup Way, Box 97200, Redmond, WA 98009.

CAREFULLY READ THE MICROSOFT LICENSE AGREEMENT
ON THE FRONT OF THIS PACKET BEFORE OPENING!

The program on this
disk is licensed to one user.

By opening this packet, you
indicate your acceptance of the
Microsoft License Agreement.

Microsoft Corporation

MICROSOFT

Microsoft QuickBASIC Compiler

Take advantage of Microsoft QuickBASIC Compiler's speed and compatibility.

- Design, write, and test programs using a BASIC Interpreter, then use the QuickBASIC Compiler so programs execute 3 to 10 times faster.
- Make the most of the QuickBASIC Compiler's compatibility with the IBM BASIC Interpreter Version 2.0.

Generate outstanding graphics and sound effects.

- Add music and sound effects to your programs using Microsoft QuickBASIC's PLAY and SOUND statements.
- Generate sophisticated graphics using the LINE, CIRCLE, PAINT, and DRAW statements.

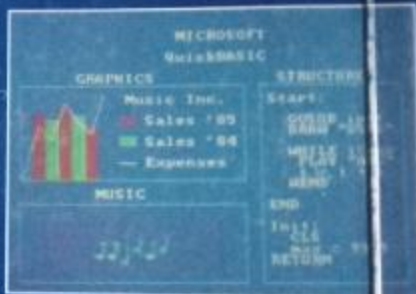
Use Microsoft QuickBASIC's advanced features to write structured code.

- Create subprograms that can use local and global variables.
- Use separate module compilation to create programs larger than 64K.
- Use alphanumeric labels and optional line numbers to enhance the readability of your programs.

System Requirements

- 128K memory, DOS 2.0 or higher, one double-sided disk drive

Microsoft is a registered trademark of Microsoft Corporation.
Part No. 007-114-001



Use the Microsoft QuickBASIC Compiler to create programs that have outstanding graphics, music, and sound effects.



Develop sophisticated applications with the IBM BASIC Interpreter, then compile them with Microsoft QuickBASIC for faster execution.

MICROSOFT

Microsoft QuickBASIC Compiler



For IBM, Personal Computers
and Compatibles

Microsoft Office Basic Access Computer

1:01

Microsoft Office Basic Access Computer