

TOPS-20 BASIC-PLUS-2 Language Manual

AA-H654A-TM

October 1979

This document describes the BASIC-PLUS-2 programming language as implemented on the TOPS-20 Operating System.

This new document replaces the document of the same name, Order Number AA-0153A-TK, and the *TOPS-20 BASIC-PLUS-2 User's Guide*, Order Number AA-0152A-TM.

OPERATING SYSTEM: TOPS-20 Version 3A
SOFTWARE: BASIC-PLUS-2 Version 2

Software and manuals should be ordered by title and order number. In the United States, send orders to the nearest distribution center. Outside the United States, orders should be directed to the nearest DIGITAL Field Sales Office or representative.

NORTHEAST/MID-ATLANTIC REGION

Technical Documentation Center
Cotton Road
Nashua, NH 03060
Telephone: (800) 258-1710
New Hampshire residents: (603) 884-6660

CENTRAL REGION

Technical Documentation Center
1050 East Remington Road
Schaumburg, Illinois 60195
Telephone: (312) 640-5612

WESTERN REGION

Technical Documentation Center
2525 Augustine Drive
Santa Clara, California 95051
Telephone: (408) 984-0200

First Printing, October 1979

The information in this document is subject to change without notice and should not be construed as a commitment by Digital Equipment Corporation. Digital Equipment Corporation assumes no responsibility for any errors that may appear in this document.

The software described in this document is furnished under a license and may only be used or copied in accordance with the terms of such license.

No responsibility is assumed for the use or reliability of software on equipment that is not supplied by DIGITAL or its affiliated companies.

Copyright © 1979 by Digital Equipment Corporation

The postage-prepaid READER'S COMMENTS form on the last page of this document requests the user's critical evaluation to assist us in preparing future documentation.

The following are trademarks of Digital Equipment Corporation:

DIGITAL	DECsystem-10	MASSBUS
DEC	DECtape	OMNIBUS
PDP	DIBOL	OS/8
DECUS	EDUSYSTEM	PHA
UNIBUS	FLIP CHIP	RSTS
COMPUTER LABS	FOCAL	RSX
COMTEX	INDAC	TYPESET-8
DDT	LAB-8	TYPESET-11
DECCOMM	DECSYSTEM-20	TMS-11
ASSIST-11	RTS-8	ITPS-10

CONTENTS

		Page
PREFACE		xi
CHAPTER 1	ELEMENTS OF BASIC	1-1
1.1	STRUCTURE OF A BASIC PROGRAM	1-1
1.1.1	Character Set	1-2
1.1.2	Line Format	1-2
1.2	STATEMENTS	1-3
1.2.1	Single-Statement, Multi-Statement, and Continuation Lines	1-4
1.3	PROGRAM DOCUMENTATION	1-5
1.3.1	The REM Statement	1-5
1.3.2	The Comment Field of a BASIC Statement	1-6
1.4	CONSTANTS	1-7
1.4.1	Real Constants	1-7
1.4.2	Integer Constants	1-8
1.4.3	String Constants	1-9
1.5	VARIABLES	1-10
1.5.1	Real Variables	1-10
1.5.2	Integer Variables	1-11
1.5.3	String Variables	1-11
1.5.4	Subscripted Variables	1-12
1.6	EXPRESSIONS	1-13
1.6.1	Arithmetic Expressions	1-13
1.6.2	String Expressions	1-14
1.6.3	Relational Expressions	1-15
1.6.4	Logical Expressions	1-17
1.6.5	Functions	1-20
1.6.6	Evaluating Expressions	1-20
1.7	ASSIGNING VALUES TO VARIABLES	1-21
1.8	ARRAYS	1-23
1.8.1	The DIM Statement	1-24
CHAPTER 2	USING BASIC-PLUS-2	2-1
2.1	BASIC COMMAND LEVEL	2-1
2.1.1	BASIC Command Format	2-2
2.1.2	Entering BASIC Command Level (BASIC, HELP)	2-2
2.1.3	Returning to System-Command Level	2-3
2.1.4	Logging off from BASIC Command Level	2-4
2.1.5	Creating a Program (NEW)	2-4
2.1.5.1	Syntax Checking	2-5
2.1.6	Checking Input (LIST)	2-5
2.1.7	Correcting Mistakes	2-6
2.1.7.1	Using the DELETE Key	2-7
2.1.7.2	Deleting a Partially Typed Line	2-7
2.1.7.3	Deleting a Field Within a Line	2-8
2.1.7.4	Deleting Lines by Line Number	2-8

CONTENTS (CONT.)

		Page
2.1.7.5	Using the DELETE Command	2-8
2.1.8	Executing a Program (RUN)	2-9
2.1.9	Interrupting Execution (CTRL/O) (CTRL/C)	2-9
2.1.10	Clearing Memory (SCRATCH)	2-9
2.1.11	Changing Line Numbers (RESEQUENCE)	2-10
2.1.12	Saving a Program in a File (SAVE)	2-11
2.1.13	Saving an Executable Program in a File (BUILD)	2-12
2.2	WORKING WITH EXISTING FILES	2-12
2.2.1	Using Line-Sequenced ASCII Files (LSA)	2-12
2.2.2	Recalling Saved Source Programs in LSA Files (OLDSA)	2-13
2.2.3	Recalling Saved Source Programs in Regular ASCII Files (OLD)	2-14
2.2.4	Renaming a Program (RENAME)	2-15
2.2.5	DELETING a SAVED PROGRAM (UNSAVE)	2-15
2.2.6	Checking Your Directory (CATALOG)	2-15
2.2.7	Combining Programs (WEAVE)	2-16
2.2.8	Issuing BASIC Commands from a File (DO)	2-17
2.2.9	BASIC Compiler Switches (STATUS)	2-19
2.2.9.1	Disabling Compiler Switches	2-20
2.2.9.2	Enabling Compiler Switches	2-21
2.2.9.3	Controlling Compiler Mode	2-21
2.2.10	Listing a Compiled Program (MLIST)	2-21
2.2.11	A Sample Program	2-23
2.3	BASIC IMMEDIATE MODE	2-25
2.3.1	Immediate-Mode Statements	2-25
2.3.2	Immediate-Mode Contexts	2-26
2.3.3	Immediate-Mode Variables	2-27
2.3.4	Halting an Immediate-Mode Program (STOP)	2-28
2.3.5	Clearing the Work Area	2-30
2.3.6	Using the START Command	2-33
2.3.7	Using the DEBUG Command	2-34
2.4	NESTING IMMEDIATE-MODE STATEMENTS	2-35
2.5	IMMEDIATE MODE AND BASIC-PLUS-2 COMMANDS	2-37
CHAPTER 3	INPUT AND OUTPUT TO THE TERMINAL	3-1
3.1	INPUTTING DATA	3-1
3.1.1	INPUT Statement	3-1
3.1.2	INPUT LINE Statement and LINPUT Statement	3-4
3.1.3	READ, DATA, RESTORE, and NODATA Statements	3-5
3.1.3.1	The READ Statement	3-5
3.1.3.2	The DATA Statement	3-6
3.1.3.3	THE RESTORE Statement	3-7
3.1.3.4	The NODATA Statement	3-8
3.2	PRINTING OUTPUT - THE PRINT STATEMENT	3-8
3.2.1	Formatting with the Comma and Semicolon	3-9
3.2.2	Output Format for Numbers and Strings	3-12
3.2.3	The TAB Function	3-12
CHAPTER 4	PROGRAM CONTROL	4-1
4.1	UNCONDITIONAL TRANSFER - THE GOTO STATEMENT	4-1
4.2	MULTIPLE BRANCHING - THE ON-GOTO STATEMENT	4-3
4.3	CONDITIONAL TRANSFER - THE IF-THEN-ELSE STATEMENT	4-4

CONTENTS (CONT.)

		Page
4.4	LOOP EXECUTION	4-7
4.4.1	The FOR and NEXT Statements	4-7
4.4.2	Nested Loops	4-10
4.4.3	The Conditional FOR Statement	4-11
4.4.4	The FOR Statement with an Additional Termination Test	4-12
4.4.5	The WHILE and UNTIL Statements	4-13
4.5	TIME-LIMIT STATEMENTS	4-14
4.5.1	The SLEEP Statement	4-14
4.5.2	The WAIT Statement	4-14
4.6	STOPPING PROGRAM EXECUTION - THE STOP AND END STATEMENTS	4-15
4.7	USING SUBROUTINES	4-17
4.7.1	The GOSUB and RETURN Statements	4-18
4.7.2	The ON-GOSUB Statement	4-19
4.8	ERROR HANDLING	4-20
4.8.1	The ONERROR GOTO Statement	4-20
4.8.2	The RESUME Statement	4-22
4.8.3	The BASIC Error Variables	4-23
4.8.4	The LINO Function	4-24
4.9	STATEMENT MODIFIERS	4-25
4.9.1	The IF Modifier	4-26
4.9.2	The UNLESS Modifier	4-27
4.9.3	The WHILE Modifier	4-28
4.9.4	The UNTIL Modifier	4-29
4.9.5	The FOR Modifier	4-29
CHAPTER 5	PROGRAM SEGMENTATION	5-1
5.1	USING SUBPROGRAMS	5-1
5.1.1	Executing a Subprogram - The CALL Statement	5-2
5.1.2	Using Dummy and Actual Arguments	5-3
5.2	TRANSFERRING CONTROL TO ANOTHER PROGRAM - THE CHAIN STATEMENT	5-5
5.3	DECLARING COMMON VARIABLE STORAGE - THE COMMON STATEMENT	5-6
5.3.1	Sharing COMMON Variables across a CALL Statement	5-7
5.3.2	Sharing COMMON Variables across a CHAIN Statement	5-9
CHAPTER 6	USING FUNCTIONS	6-1
6.1	NUMERIC FUNCTIONS	6-1
6.1.1	Trigonometric Functions (PI, SIN, COS, TAN, COT, ATN, ATN2)	6-2
6.1.2	Algebraic Functions	6-4
6.1.2.1	Square Root Function (SQR)	6-5
6.1.2.2	Exponential and Log Functions (EXP, LOG, and LOG10)	6-5
6.1.2.3	Integer Function (INT)	6-7
6.1.2.4	Absolute Value Function (ABS)	6-8
6.1.2.5	SIGN(SGN) and FIX(FIX) Functions	6-9
6.1.3	Random Numbers (Function RND and RANDOMIZE Statement)	6-10
6.1.4	MOD Function	6-12
6.2	STRING FUNCTIONS	6-13

CONTENTS (CONT.)

	Page
6.2.1	Finding the Length of a String (LEN) 6-13
6.2.2	Trimming Trailing Blanks (TRM\$) 6-14
6.2.3	Finding the Position of a Segment (POS, INSTR) 6-14
6.2.4	Extracting a Segment from a String (SEG\$) 6-15
6.2.5	The MID\$ Function 6-17
6.2.6	The LEFT\$ and RIGHT\$ Functions 6-18
6.2.7	The STRING\$ and SPACE\$ Functions 6-19
6.2.8	The EDIT\$ Function 6-20
6.3	CONVERSION FUNCTIONS 6-22
6.3.1	Converting a Character to ASCII Code (ASCII) 6-22
6.3.2	Converting ASCII Code to a Character (CHR\$) 6-22
6.3.3	Converting an Integer to RADIX-50 (RAD) 6-23
6.3.4	Translating from one Storage Code to Another (XLATE) 6-24
6.3.5	The CHANGE Statement 6-24
6.3.6	Numbers and their String Representation (VAL%,VAL, NUM\$ and STR\$) 6-26
6.4	DATE, TIME, AND DIRECTORY FUNCTIONS 6-28
6.4.1	Returns Current Clock Time (CLK\$) 6-28
6.4.2	Returns Current Date in the Format: dd-mmm-yy (DAT\$) 6-29
6.4.3	Returns Date in the Format: mm/dd/yy (DATE\$) 6-29
6.4.4	Returns Time in the Format: hh:mm (TIME\$) 6-30
6.4.5	Returns Clock, CPU, or Job Connect Time (TIME) 6-30
6.4.6	Returns Connected Structure and Directory (USR\$) 6-31
6.5	TERMINAL-FORMAT FILE FUNCTIONS 6-31
6.5.1	Returns Margin Width (MAR%) 6-31
6.5.2	Returns Horizontal Print Position (POS%) 6-32
6.5.3	Returns Vertical Print Position (VPS%) 6-33
6.5.4	Returns Current Page Count (PPS%) 6-34
6.6	SYSTEM FUNCTIONS 6-34
6.6.1	Resume Program Output (RCTRL0) 6-34
6.6.2	Disable and Enable Echoing (NOECHO and ECHO) 6-35
6.6.3	Enable and Disable Trapping of CTRL/C Interrupts (CTRLC And RCTRLC) 6-36
6.6.4	Exit from a Program (ABORT) 6-37
6.7	USER-DEFINED FUNCTIONS - THE DEF STATEMENT 6-37
6.7.1	Single-Line DEF Statement 6-38
6.7.2	Multi-Line DEF Statement 6-41
6.7.3	Multi-Line DEF* Statement 6-43
CHAPTER 7	USING ARRAYS 7-1
7.1	DIMENSIONING AN ARRAY 7-1
7.2	INITIALIZING AN ARRAY 7-1
7.3	MATRIX OPERATIONS 7-4
7.3.1	Matrix Assignment 7-4
7.3.2	Matrix Addition and Subtraction 7-4
7.3.3	Matrix Multiplication 7-4
7.3.4	Matrix Transposition 7-5

CONTENTS (CONT.)

		Page
7.3.5	Inverting and Finding the Determinant of a Matrix	7-6
7.4	ARRAY INPUT AND OUTPUT	7-6
7.4.1	MAT INPUT Statement	7-7
7.4.2	MAT PRINT Statement	7-8
7.4.3	MAT READ Statement	7-9
CHAPTER 8	USING TERMINAL-FORMAT AND VIRTUAL-ARRAY FILES	8-1
8.1	TERMINAL-FORMAT FILES	8-1
8.1.1	Opening Terminal-Format Files	8-2
8.1.2	Closing Terminal-Format Files	8-4
8.1.3	Reading Data from a Terminal-Format File	8-4
8.1.3.1	The INPUT LINE # and LINPUT # Statements	8-6
8.1.4	Writing to a Terminal-Format File	8-7
8.1.5	Restoring a Terminal-Format File	8-8
8.1.6	Truncating a Terminal-Format File	8-9
8.1.7	Checking for the End of a Terminal-Format File	8-9
8.1.7.1	IFEND # Statement	8-9
8.1.7.2	IFMORE # Statement	8-10
8.1.7.3	NODATA # Statement	8-10
8.1.8	Changing Margins in a Terminal-Format File	8-10
8.1.9	Setting Page Size in a Terminal-Format File	8-11
8.2	VIRTUAL-ARRAY FILES	8-12
8.2.1	Dimensioning a Virtual-Array File	8-12
8.2.2	Opening and Closing Virtual-Array Files	8-13
8.3	FILE RENAMING AND DELETING	8-15
8.3.1	The NAME-AS Statement	8-15
8.3.2	The KILL Statement	8-16
CHAPTER 9	USING RECORD FILES	9-1
9.1	FILE ORGANIZATION	9-1
9.1.1	Sequential Organization	9-2
9.1.2	Relative Organization	9-2
9.1.3	Indexed Organization	9-2
9.2	ACCESS METHODS	9-3
9.2.1	Sequential Access	9-3
9.2.1.1	Sequential Access to Sequential Files	9-3
9.2.1.2	Sequential Access to Relative Files	9-4
9.2.1.3	Sequential Access to Indexed Files	9-4
9.2.2	Random Access	9-4
9.2.2.1	Random Access to Relative Files	9-5
9.2.2.2	Random Access to Indexed Files	9-5
9.3	RECORD FORMATS	9-6
9.3.1	Fixed-Length Records	9-7
9.3.2	Variable-Length Records	9-7
9.3.3	Stream-Format Records	9-7
9.4	RECORD MAPPING	9-8
9.5	FILE OPERATIONS	9-12
9.5.1	Creating and Accessing a File	9-12
9.5.1.1	Opening a Sequential File	9-16
9.5.1.2	Opening a Relative File	9-17
9.5.1.3	Opening an Indexed File	9-17

CONTENTS (CONT.)

		Page
9.5.2	Closing a File	9-18
9.5.3	Restoring a File	9-18
9.5.4	Truncating a File	9-18
9.6	RECORD OPERATIONS	9-19
9.6.1	Sequential Record Operations	9-19
9.6.2	Relative Record Operations	9-20
9.6.3	Indexed Record Operations	9-21
9.6.4	Record Locking	9-23
9.7	DYNAMIC MAPPING OF AN I/O BUFFER	9-24
9.8	EXAMPLES	9-26
CHAPTER 10	FORMATTED OUTPUT - THE PRINT USING STATEMENT	10-1
10.1	THE PRINT USING STATEMENT	10-2
10.2	FORMATTING NUMBERS WITH PRINT USING	10-2
10.2.1	Specifying the Number of Digits	10-3
10.2.2	Specifying the Decimal Point Location	10-4
10.2.3	Printing a Number that is Larger than the Field	10-4
10.2.4	Printing Numbers With Special Symbols	10-5
10.2.4.1	Printing Numbers with a Trailing Minus Sign	10-5
10.2.4.2	Printing Numbers in Asterisk-Fill Fields	10-6
10.2.4.3	Printing Numbers with Floating Dollar Signs	10-7
10.2.4.4	Printing Numbers with Commas	10-7
10.2.5	Printing Numbers in E (Exponential) Format	10-8
10.2.6	Fields that Exceed BASIC's Accuracy	10-8
10.3	FORMATTING STRINGS WITH PRINT USING	10-8
10.3.1	One-Character String Fields	10-9
10.3.2	Printing Strings in Left-Justified Format	10-9
10.3.3	Printing Strings in Right-Justified Format	10-9
10.3.4	Printing Strings in Centered Fields	10-10
10.3.5	Printing Strings in Extended Fields	10-10
10.4	SUMMARY OF FORMAT CHARACTERS	10-11
10.5	THE IMAGE STATEMENT	10-13
10.6	PRINT USING STATEMENT ERROR CONDITIONS	10-14
10.6.1	Fatal Error Conditions	10-14
10.6.2	Warning Conditions	10-14
APPENDIX A	DIAGNOSTIC MESSAGES	A-1
A.1	COMMAND ERROR MESSAGES	A-1
A.2	COMPILATION ERROR MESSAGES	A-2
A.2.1	Compilation Error Messages (Fatal)	A-3
A.2.2	Compilation Error Messages (Warning)	A-11
A.3	EXECUTION MESSAGES	A-12
A.3.1	Trappable Error and Warning Messages	A-12
A.3.2	Nontrappable Error and Warning Messages	A-17
APPENDIX B	USING THE TOPS-20 OPERATING SYSTEM	B-1
B.1	CONTACTING THE TOPS-20 OPERATING SYSTEM	B-1
B.1.1	What is a Job?	B-2
B.1.2	Logging In (LOGIN)	B-2
B.2	IF THE SYSTEM STOPS	B-3

CONTENTS (CONT.)

		Page
	B.3 LOGGING OFF THE SYSTEM	B-4
	B.4 FILE SPECIFICATIONS	B-4
	B.4.1 Protection Codes	B-7
APPENDIX C	RUNNING BASIC-PLUS-2 THROUGH BATCH	C-1
	C.1 CREATING A CONTROL FILE	C-1
	C.2 SUBMITTING A JOB TO BATCH	C-2
	C.3 A BATCH EXAMPLE	C-3
APPENDIX D	BASIC RESERVED WORDS	D-1
APPENDIX E	ASCII CODE	E-1
APPENDIX F	SUMMARY OF BASIC-PLUS-2 STATEMENTS, FUNCTIONS, AND OPERATORS	F-1
	F.1 STATEMENTS	F-1
	F.2 FUNCTIONS	F-13
	F.3 OPERATORS	F-16
INDEX		Index-1

TABLES

TABLE	1-1	Keywords and Spaces	1-4
	1-2	Number Notations	1-8
	1-3	Arithmetic Operators	1-14
	1-4	Relational Operators	1-15
	1-5	String Relational Operators	1-18
	1-6	Logical Operators	1-19
	1-7	Truth Tables	1-21
	1-8	Operator Precedence	1-21
	4-1	BASIC Statements	4-26
	5-1	Argument Data Types	5-5
	6-1	EDIT\$ Conversions	6-20
	8-1	ACCESS-Clause Keywords of the OPEN Statement	8-3
	9-1	Record-File Access Methods	9-3
	9-2	Record Formats	9-6
	10-1	Format Characters for Numeric Fields	10-11
	10-2	Sample Formats for Numeric Fields	10-12
	10-3	Format Characters for String Fields	10-13
	B-1	System Device Names	B-5
	E-1	ASCII Table	E-1
	F-1	Arithmetic Operators	F-16
	F-2	Logical Operators	F-16
	F-3	Relational Operators	F-17

PREFACE

This manual is a complete reference to the BASIC-PLUS-2 programming language as it is implemented on the TOPS-20 operating system. This manual is neither a primer nor a user's guide. If you are unfamiliar with the BASIC language, refer to DIGITAL's Introduction to BASIC.

If you are unfamiliar with the TOPS-20 operating system or the TOPS-20 batch system, refer to the following:


Getting Started With TOPS-20
TOPS-20 User's Guide
TOPS-20 Commands Reference Manual
Getting Started With Batch (TOPS-20)
TOPS-20 Batch Reference Manual

This manual describes all elements of BASIC-PLUS-2 in the following chapters:

- Chapter 1 contains descriptions of the elements of BASIC programs and the BASIC language.
- Chapter 2 provides descriptions of how language elements are combined to create programs. This chapter also describes how you manipulate programs from BASIC command level.
- Chapter 3 provides descriptions of input and output to and from the terminal.
- Chapter 4 describes the BASIC program-control statements. These statements enable you to direct the order in which BASIC executes statements within a program.
- Chapter 5 describes the techniques for segmenting large BASIC programs into smaller, more manageable units.
- Chapter 6 describes all the BASIC functions. In addition, this chapter describes the methods for defining your own functions.
- Chapter 7 describes the statements and techniques for manipulating numeric and character data in arrays.
- Chapters 8 and 9 describe a variety of techniques for performing program input and output to and from files. Chapter 8 describes two types of BASIC files called terminal-format and virtual-array files; Chapter 9 describes a third type of file, the record file.
- Chapter 10 describes how to format output with the PRINT USING statement.

For easy reference, the first page of each chapter contains a listing of that chapter's major subsections. In addition, Appendix F contains a brief summary of the BASIC statements, functions, and operators.

Conventions Used in this Manual

Symbol	Represents
CTRL/x	Pressing the CTRL key and another key simultaneously (for example, CTRL/C).
	Pressing the RETURN key (carriage return/line feed).
[]	Brackets indicating optional information that can be omitted from a statement or command.
{ }	Braces indicating a choice. Choose one from the enclosed options.
lowercase letters	Lowercase characters in a command string or statement indicate variable information you must supply.
UPPERCASE LETTERS	Uppercase characters in a command string or statement indicate literal information that you must enter as shown.
Examples	When you list a BASIC-PLUS-2 program from storage, leading zeroes are added to the number (for example, 10 becomes 00010).
Red print	Where examples contain user input and computer output, all user input is printed in red. Unless otherwise noted, all lines printed in red are ended by the RETURN key.

CHAPTER 1

ELEMENTS OF BASIC

BASIC (Beginner's All-purpose Symbolic Instruction Code) is a computer language developed at Dartmouth College under the direction of Professors John G. Kemeny and Thomas E. Kurtz. It is one of several programming languages used to translate symbolic language programs into machine language. Because the BASIC language is composed of easily understood statements and commands, it is one of the simplest programming languages to learn.

BASIC allows you to communicate directly with the language processor. It is a conversational programming language that uses simple English-like statements and familiar mathematical notations to perform operations.

The BASIC-PLUS-2 language is an outgrowth of Dartmouth BASIC. It encompasses both the elementary statements used to write simple programs and many new and advanced features. These new features, not found in standard Dartmouth BASIC, allow you to produce more complex and efficient programs.

This chapter is divided into the following sections:

- 1.1 Structure of a BASIC Program
- 1.2 Statements
- 1.3 Program Documentation
- 1.4 Constants
- 1.5 Variables
- 1.6 Expressions
- 1.7 Assigning Values to Variables
- 1.8 Arrays

1.1 STRUCTURE OF A BASIC PROGRAM

A BASIC program consists of a set of statements constructed with the language elements and syntax described in the following chapters. The next sections describe line numbers, statements and expressions used in BASIC.

ELEMENTS OF BASIC

1.1.1 Character Set

BASIC-PLUS-2 uses the full ASCII (American Standard Code for Information Interchange) character set for its alphabet. This set includes:

1. Letters A through Z and a through z
2. Numbers 0 through 9
3. Special characters (see the ASCII Table in Appendix E)

This character set enables you to include any ASCII character as part of a program. BASIC translates the characters that you type into machine language; some characters are processed and some are ignored. When BASIC does ignore an ASCII character, it prints a warning message to that effect.

BASIC translates characters in the following manner:

1. Letters A through Z and a through z - BASIC treats the same alphabetic in uppercase and lowercase as the same character, for example, I is the same as i.
2. Non-printing characters - BASIC interprets the code during input, prints a warning message, then ignores them during execution.
3. NUL characters - BASIC interprets the code during input, prints a warning message, and ignores them during execution.

BASIC translates string constants differently (See Section 1.4.3.). Everything you type into a string constant is interpreted literally by BASIC. Consequently, in a string constant:

1. All lowercase alphabetic (a,b,c) remain in lowercase.
2. All non-printing characters are processed.
3. All null characters are processed.

BASIC ignores all characters in a REMARK during execution.

System-editing characters affect terminal-input format only. Therefore, you need not be concerned with the way BASIC handles them. System-editing characters, such as CTRL/U, are described fully in the TOPS-20 User's Guide.

1.1.2 Line Format

The format of a BASIC program line is as follows:

```
line number keyword statement line terminator
10 PRINT R=SQR(X^2+Y^2) (RET)
```

Most lines in a BASIC program must begin with a number. (Only continuation lines do not; see Section 1.2.1.) This line number must be a positive integer within the range of 1 to 99999. A BASIC line number is a label that distinguishes one line from another within a program. Consequently, each line number in the program must be unique. If you type a line number that is outside the valid range, BASIC prints an error message and the line number and its contents are ignored.

ELEMENTS OF BASIC

Leading zeros (as well as leading and trailing spaces) have no effect on the number. However, you cannot have embedded spaces within a line number. For example, these numbers are the same to BASIC:

```
00010
  10
```

But this number is invalid:

```
0 1 0
```

BASIC ignores leading and trailing blanks, spaces, and tabs within a line (unless in a string enclosed by quotation marks). Therefore, you need not worry about leading and trailing blanks when typing in a program. For example:

```
10 LET A=B+C
```

can also be typed:

```
10      LET      A = B + C
```

Both lines are the same to BASIC.

However, embedded spaces in line numbers, keywords or variable names are illegal. For example, BASIC rejects the previous statement if you type it as follows:

```
1 0   L E T   A   =B + C
```

The longest line you can type into a BASIC program is 2,560 characters including spaces, tabs, line terminators, continuation characters, and line numbers.

1.2 STATEMENTS

BASIC statements consist of keywords (words recognized by BASIC) that you use in conjunction with elements of the language set: constants, variables, and operators. BASIC statements divide into two major groups: executable statements and non-executable statements.

At least one space or tab must follow all statement keywords in order for BASIC to recognize the keyword as such. For example:

```
This is acceptable      10 PRINT A
```

```
This is not              10 PRINTA
```

Certain keywords consist of two or more English words. Some keywords allow an optional space between words, some keywords require a space between words, and some keywords do not permit a space at all. Table 1-1 lists these keywords.

ELEMENTS OF BASIC

Table 1-1
Keywords and Spaces

Optional Space	Mandatory Space	No Space
GO TO GO SUB ON ERROR	MAT INPUT MAT PRINT MAT READ INPUT LINE	FNEND SUBEND

Keywords are reserved and, therefore, cannot be used as variable names (See Section 1.5.). Appendix D contains a complete list of reserved keywords.

1.2.1 Single-Statement, Multi-Statement, and Continuation Lines

You have the option, with BASIC-PLUS-2, of typing either one statement on one line, several statements on one line, or one or more statements on several lines.

A single-statement line consists of:

1. A line number (from 1 to 99999)
2. A statement keyword
3. The body of the statement
4. A line terminator

This is a single-statement line:

```
10 LET A=B*C
```

To enter more than one statement on a single line (multi-statement line), separate each complete statement with a backslash (\). The backslash symbol is the statement separator (or terminator). You must type it after every statement except the last in a multi-statement line. For example, the following line contains three complete PRINT statements:

```
10 PRINT A;\PRINT V;\PRINT G
```

The line number labels the whole line. Consequently, if you plan to transfer control to a particular line within a program and that line contains more than one statement, all statements will be executed. For instance, in the previous example, you cannot execute just the statement

```
PRINT V,
```

without executing PRINT A; and PRINT G.

Most statements can appear in a multi-statement line. The exceptions are noted in the descriptions of individual statements in this manual.

ELEMENTS OF BASIC

The rules for structuring a multi-statement line are:

1. Only the first statement in a series can have a line number.
2. Successive statements must be separated with a backslash (\).

BASIC also provides a continuation character (an ampersand (&)) in case the length of the statement or multi-statement line exceeds the line.

If you are at the end of a line and you want to continue it, type an ampersand (&) and then a line terminator. The next character you type prints in column one of the following line. The continuation line cannot have a line number. You reference the entire line by the original line number.

If the character immediately preceding the line terminator is an ampersand, BASIC continues executing the line as if all information were on the same line.

Consider the following example:

```
60 PRINT 'ABC', 2 &  
    ,4, 3*22.9
```

You can continue any statement. You cannot, however, continue a comment field (See Section 1.3.2.).

1.3 PROGRAM DOCUMENTATION

BASIC allows you to document your methods by inserting descriptive text in the source program. This type of documentation is known as a remark or comment. There are two ways of inserting information within a BASIC source program:

1. With the REM statement
2. With the comment field

1.3.1 The REM Statement

The REM statement has the following format:

```
REM comment
```

where:

```
comment          is anything you want to write.
```

You may place a REM statement anywhere in your program because it does not affect program execution.

A REM statement may be the only statement on the line:

```
10 REM THIS IS AN EXAMPLE
```

ELEMENTS OF BASIC

It can also be one of several statements in a multi-statement line. BASIC ignores anything in a line following the keyword REM. The only character that ends a REM statement is a line terminator. Therefore, a REM statement should be the only statement on the line or the last statement in a multi-statement line.

```
20 LET A=5\REM THE VALUE OF A IS 5
```

You can use the line number of a REM statement in a reference from another statement, for example, from a GOTO. In this, however, BASIC ignores the REM statement and proceeds to execute the next non-REM statement following the line referenced. For example:

```
00010 REM SQUARE ROOT FUNCTION
00020 INPUT A
00030 IF A=0 GOTO 60
00040 PRINT "THE SQUARE ROOT OF";A;"IS";SQR(A)
00050 GOTO 10
00060 END
```

Line 50 sends BASIC back to line 10. BASIC ignores the comment on line 10 and continues execution at line 20.

Remember that BASIC prints the remarks on the terminal only when you list the source program.

1.3.2 The Comment Field of a BASIC Statement

The second method for adding comments to a program is to use the comment field. You mark the beginning of the comment with an exclamation point (!). For example:

```
10 A = B+C ! THIS IS A TEST.
```

The comment has no effect on the execution of the statement. You can end the comment with either an exclamation point (!) or a line terminator.

You can place a comment between statements on a multi-statement line if you terminate the comment with an exclamation point. A backslash does not terminate a comment field because BASIC interprets it as part of the comment.

You cannot continue a comment field to another line. You can, however, continue the statement preceding the comment. For example:

```
10 IF A=B ! THIS IS A COMMENT &
    THEN PRINT B
```

BASIC ignores the comment and continues the IF statement on the next line.

The DATA statement and the IMAGE statement are the only BASIC statements that cannot have comment fields. Each must be the only statement on its respective line.

ELEMENTS OF BASIC

1.4 CONSTANTS

There are three types of constants in BASIC:

1. Real (also called floating-point numbers)
2. Integer (whole numbers)
3. String (alphanumeric and/or special characters)

1.4.1 Real Constants

A real constant is one or more decimal digits, either positive or negative, in which the decimal point is optional.

The following are all valid real constants (real numbers):

5	42861
74	-125
6.	.95

BASIC accepts real constants in the range of 1.70142×10^{-38} to $1.70141 \times 10^{+38}$.

If you type a real constant into the source text that is outside the range of real values, BASIC prints a warning message and internally substitutes zero (if the number is too small), or $1.70141 \times 10^{+38}$ (if the number is too large).

You can, however, enter very large numbers and very small numbers within this range by using a method similar to scientific notation. To do so, use the following format:

+ or - x.xxxxxE + or - n
5.24016E-3

where:

- + or - is the sign of the number. The plus sign (+) is optional with positive numbers; the minus sign (-) is mandatory with negative numbers.
- x is the number which may be carried to eight decimal places.
- E represents the words "times 10 to the power of".
- + or -n is the positive or negative exponential value (the power of 10).

This method of mathematical shorthand is called E notation or floating-point notation. It is BASIC's way of representing scientific notation. To use this format, append the letter E to the number, then follow the E with an optionally signed integer constant. This constant is the exponent. It can be 0 but never blank. Thus you can type:

6000000 as 6E6 and .000005 as 5E-6

ELEMENTS OF BASIC

With E notation you are actually positioning the decimal point internally. A positive exponent moves the decimal point to the right; a negative exponent moves the decimal point to the left. For instance, if you type the number

5.2041E-3

BASIC interprets it as .0052041.

Table 1-2 shows the different methods of writing real constants.

Table 1-2
Number Notations

Standard Notation	Scientific Notation	E Notation
1000000	1×10^6	1.00000E+06
10000000	1×10^7	1.00000E+07
100000000	1×10^8	1.00000E+08
1000000000000	1×10^{12}	1.00000E+12

The following are examples of real constants:

.84103E-06	-377	-12345
6.64	5E+03	8.0E-03
-9.4177	6562	25

BASIC uses single-precision floating-point format when storing and calculating most numbers. Integers, however, are handled in a slightly different manner.

1.4.2 Integer Constants

An integer constant is a whole number (no fractional part) written without a decimal point. To type an integer constant, type an optional sign followed by one or more decimal digits terminated by a percent sign (%). For example, the following numbers are all integer constants (whole numbers):

29%	-8%
3432%	1%
12345%	205%

The following are not integer constants:

1.6	.08%
754.2%	5.2041E+06

In BASIC, you can type integer constants within the range of $(-2^{35})-1$ to $(2^{35})-1$.

If you type an integer constant into the source text that is outside the range of integer values, BASIC prints a warning and internally substitutes $(2^{35})-1$, if the number is too big, and $-(2^{35})$, if the number is too small.

ELEMENTS OF BASIC

1.4.3 String Constants

A string constant (also called a literal) is one or more alphanumeric and/or special characters enclosed by double quotation marks ("text") or single quotation marks ('text'). You can include double quotation marks within a string constant delimited by single quotation marks and vice versa. Include both the starting and ending delimiters when typing a string constant in a source program. These delimiters must be of the same type (both double quotation marks or both single quotation marks).

A character in a string constant can be a letter, a number, a space, or any ASCII character except a line terminator. The value of the string constant is determined by all its characters, including leading, embedded, or trailing spaces. For example, because of the number of spaces between the quotation marks and the characters in the following example

```
"    DIGITAL    " is not the same as "DIGITAL"
```

BASIC prints every character between quotation marks exactly as you type it into the source program, including:

- Lowercase letters (a-z)
- Leading, trailing, and embedded spaces
- Tabs

Note, however, that BASIC does not print delimiting quotation marks when the program is executed.

```
00010 PRINT "DIGITAL"  
00020 END
```

```
READY  
RUNNH  
DIGITAL
```

In order to make BASIC print quotation marks, you must enclose them within a pair of quotation marks, either double or single, as follows:

```
00010 PRINT 'HE SAID, "GOOD MORNING!"'  
00020 END
```

```
READY  
RUNNH  
HE SAID, "GOOD MORNING!"
```

Here are some examples of string constants:

```
"This is a String Constant"  
'SO IS THIS'  
"GARY'S TENNIS RACKET"
```

The following are examples of invalid string constants:

```
"WRONG TERMINATOR"  
'SAME HERE"  
"NO TERMINATOR"
```

ELEMENTS OF BASIC

1.5 VARIABLES

Depending on the operations you specify in a program, the value of a variable may change from statement to statement. BASIC uses the most recently assigned value of a variable when performing calculations. This value remains the same until a statement is executed that assigns a new value to that variable.

BASIC has three types of variables:

1. Real
2. Integer
3. String

Variable names cannot be keywords because all keywords are reserved (See Appendix D). The following sections describe the formation of legal variable names.

1.5.1 Real Variables

A real variable is a named location in which a single real value is stored. You name a real variable with a single letter followed by 29 optional characters consisting of letters, digits, or periods. Therefore, the maximum length of a real variable name is 30 characters:

1 letter
29 optional characters

Do not embed spaces between characters. The following are legal real variable names:

C	L...5
M1	BIG47
F67T.J	Z2.

The following are not legal real variable names:

6	225
.A	G*T
4D	8/3

Before program execution, BASIC sets all real variables to 0 except for those in a virtual array. If you require an initial value other than 0, you can assign it with the LET statement (See Section 1.7.). Otherwise, you can declare the value implicitly by just typing the variable in a program.

NOTE

Because other BASIC implementations may not set all variables to zero before program execution, you should not rely on this feature. Good programming practice dictates that you initialize all variables at the beginning of the program.

ELEMENTS OF BASIC

1.5.2 Integer Variables

An integer variable (like a real variable) is a named location in which a single value can be stored. Using an integer variable in your program indicates that space is reserved for the storage of a whole number (no fractional part).

You name an integer variable with a single letter followed by 29 optional characters consisting of letters, digits, or periods, and you terminate the name with a percent sign (%). No embedded spaces are allowed. Therefore, the maximum length of an integer variable name is 31 characters:

1 letter
29 optional characters
1 percent sign (%)

The following are legal integer variable names:

ABCDEFG% C.8%
B% D6E7%

The following are not legal integer variable names:

A B2\$
1B% 123%

If you include an integer variable in a program, then the value you supply for it must be an integer value. If a real value (real number) is assigned to an integer variable, BASIC drops the fractional portion of the value. The number is not rounded to the nearest integer; it is truncated. Consider the following example:

B% = -5.7

BASIC assigns the value -5 to the integer variable, not -6. This method of truncating can lead to serious inaccuracies.

If you assign an integer value to a real variable, BASIC stores the integer as a real number internally.

1.5.3 String Variables

A string variable is a named location used to store strings. You name a string variable with a letter, followed by 29 optional characters consisting of letters, digits, or periods, and you terminate the name with a dollar sign (\$). No embedded spaces are allowed between characters. The dollar sign (\$) must be the last character in the name. Therefore, the maximum length of a string variable name is 31 characters:

1 letter
29 optional characters
1 dollar sign (\$)

The following are examples of legal string variable names:

C1\$ M\$
L.6\$ F34G\$
ABC1\$ T..\$

ELEMENTS OF BASIC

These are not legal string variable names:

C1	12345\$
6.L\$.\$8
\$56A	AB

Strings have a value and a length. During execution, the length of a character string associated with a string variable can vary from 0 (signifying a null or empty string) to 262,143 characters.

Note that when all three variables -- a simple real variable, an integer variable, and a string variable -- have the same name, each one represents a different variable. The following names are all valid within the same BASIC program:

A5	a simple real variable
A5%	an integer variable
A5\$	a string variable

NOTE

One of the most frequent errors made in BASIC programs is to forget the percent sign on an integer variable when the program also contains a real variable with the same spelling. Be sure to include a percent sign at the end of an integer variable name when your program contains a real variable name with the same spelling.

1.5.4 Subscripted Variables

A subscripted variable is a real, integer, or string variable with one or two subscripts appended to it. The subscripts can be any positive expression. BASIC converts non-integer expressions to integer by truncating the fraction.

The subscript in a subscripted variable is a pointer to a specific location in a list or table in which a value is stored. (See Section 1.8 for more information on lists and tables.) You designate the pointer with either one or two subscripts enclosed by parentheses. If there are two subscripts, separate them with a comma. The value stored can be real, integer, or string data.

To name a subscripted variable, start with a real, integer, or string variable name:

A	A%	A\$
---	----	-----

To refer to an element in a list (one dimension), follow the variable name with one subscript within parentheses. For example:

A(6)	A%(6)	A\$(6)
------	-------	--------

A(6) refers to the seventh item in this list:

A(0)	A(1)	A(2)	A(3)	A(4)	A(5)	A(6)
10	20	30	40	50	60	70

ELEMENTS OF BASIC

To refer to an element in a table (two dimensions), follow the variable name with two subscripts. The first subscript designates the row number, and the second subscript designates the column number. Separate the two subscripts with a comma. For example:

A(7,2) A%(4,6) A\$(17,23)

BASIC accepts the same alphanumeric characters for a simple real variable and a subscripted variable within the same program. However, do not use the same alphanumeric characters for two arrays (See Section 1.8.) with a different number of subscripts.

For example, these names are acceptable in the same program:

D simple real variable
D(8) subscripted variable

These names are not acceptable in the same program:

D(8) one subscript
D(8,6) two subscripts

1.6 EXPRESSIONS

An expression can be a constant, variable, function (See Section 1.6.5.), array reference (See Section 1.8.), or any combination of these, separated by any of the following operators:

- Arithmetic operators
- String operators
- Relational operators
- Logical operators

1.6.1 Arithmetic Expressions

BASIC allows you to perform addition, subtraction, multiplication, division, and exponentiation of integer variables, real variables, and constants with the following operators:

** or ^ Exponentiation
* Multiplication
/ Division
+ Addition, Unary +
- Subtraction, Unary -

Performing an arithmetic operation on two arithmetic expressions of the same data type yields a result of that same type. For example:

A%+B% = an integer expression
G3*M5 = a real expression

ELEMENTS OF BASIC

If you combine an integer quantity with a real quantity, the result will be real. For example:

$A * B\%$ = a real expression

$6.83 * 5\% = 34.15$

Note that, in general, you cannot place two arithmetic operators consecutively in the same expression. The exceptions are the unary plus and unary minus. For example, both of the following are valid expressions:

$A * -B$ $A * (-B)$

Table 1-3 provides examples of arithmetic operators and their meaning.

Table 1-3
Arithmetic Operators

Operator	Example	Meaning
-	-A	Negate A.
+	+A	Has no effect.
+	A + B	Add B to A.
-	A - B	Subtract B from A.
*	A * B	Multiply A by B.
/	A/B	Divide A by B.
^	A^B	Calculate A to the power B.
**	A**B	Calculate A to the power B.

1.6.2 String Expressions

BASIC provides the plus sign (+) and the ampersand (&) as operators for string expressions. By using either of these operators you can append one string to another. This operation is called concatenation.

Consider the following example:

```
10 C$ = "GOOD" + "BYE"  
20 PRINT C$  
30 END
```

During execution, BASIC prints the following:

GOODBYE

$A\$ + B\$$ or $A\$ \& B\$$ mean concatenate or append string $B\$$ to the end of string $A\$$.

ELEMENTS OF BASIC

1.6.3 Relational Expressions

A relational operator is a symbol used to compare the value of one variable or expression to another variable or expression within a BASIC program, thus creating a relational expression. As explained in Section 4.3, one of the uses of relational expressions is with the IF-THEN-ELSE statement to create conditional transfers.

When a relational operator is used its value is -1 if the relation is true, and 0 if the relation is false. The value -1 is logically true and 0 is false. See Section 1.6.4 for a description of logical expressions.

NOTE

BASIC issues an error message if you try to compare a real or integer expression to a string expression using a relational operator.

Table 1-4 provides examples of relational operators and their meaning.

Table 1-4
Relational Operators

Operator	Example	Meaning
=	A = B	A is equal to B.
<	A < B	A is less than B.
>	A > B	A is greater than B.
<=, =<	A<= B	A is less than or equal to B.
>=, =>	A>= B	A is greater than or equal to B.
#, <>, ><	A<>B	A is not equal to B.
==	A==B	A is approximately equal to B if the difference between A and B is less than 10^{-6} .

When you use a relational operator to compare the value of two string expressions, you create a relational string expression. BASIC uses the ASCII character collating sequence to determine which character is greater or lesser in value than the other. (See Appendix E for the ASCII Table.) The comparison is made, character by character, left to right, by ASCII value until BASIC finds a difference in value.

When applied to strings, relational operators compare characters for alphabetic sequence. Consider the following program:

```
00010 A$ = "ABC"
00020 B$ = "DEF"
00030 IF A$<B$ GOTO 60
00040 PRINT A$
00050 PRINT B$
00060 END
```

ELEMENTS OF BASIC

When BASIC executes line 30, it compares string A\$ with B\$ to determine if A\$ occurs first in alphabetic sequence. In this case, it does, and the program transfers control to line 60. If string B\$ occurred before string A\$, program execution would have continued to the next statement following the comparison, that is, line 40.

BASIC compares strings just as you compare words to be placed in alphabetical order. BASIC compares the first character in each string, A and D. The letter A precedes the letter D in the ASCII table; therefore, string A\$ precedes string B\$ in alphabetic sequence. If the first two characters are equal, BASIC proceeds to the second two characters, until a difference is found. For example:

```
ABC
AEF
```

BASIC compares A to A and finds them equal in value. Then BASIC compares B and E and finds B less than E. The comparison ends here, and BASIC concludes that ABC occurs before AEF in alphabetic sequence.

Table 1-5 provides examples of string operators and their meaning.

Table 1-5
String Relational Operators

Operator	Example	Meaning
=	A\$ = B\$	Strings A\$ and B\$ are equal after removing trailing blanks and nulls.
<	A\$ < B\$	String A\$ occurs before string B\$ in alphabetic sequence.
>	A\$ > B\$	String A\$ occurs after string B\$ in alphabetic sequence.
<=,=<	A\$<=B\$	String A\$ is equal to, or precedes, string B\$ in alphabetic sequence.
>=,=>	A\$>=B\$	String A\$ is equal to, or follows, string B\$ in alphabetic sequence.
#, <>,><	A\$#B\$	String A\$ is not equal to string B\$.
==	A\$==B\$	Strings A\$ and B\$ are identical (exactly the same length without padding and the same composition of all characters).

Note that the relational operator == has a different meaning when applied to strings than when applied to numbers. When comparing strings of different lengths, BASIC treats the shorter string as if it were padded with trailing blanks to the length of the longer string. In order to perform character-to-character comparison, BASIC needs two characters to compare. This is where the trailing blanks serve their purpose.

ELEMENTS OF BASIC

Consider the following example:

```
00010 A$ = "ANTONY"  
00020 B$ = "CLEOPATRA"  
00030 IF A$<B$ GOTO 50  
00040 A$ = A$ + B$  
00050 PRINT A$  
00060 END
```

BASIC compares the A in ANTONY and the C in CLEOPATRA. The ASCII value of A is 65; the ASCII value of C is 67. Therefore string A\$ precedes string B\$ in alphabetic sequence. Control shifts to line 50. If A\$ were greater than B\$, the program would continue to line 40. This is what happens when BASIC executes the program:

```
RUNNH  
ANTONY
```

1.6.4 Logical Expressions

A logical expression consists of either one operand preceded by a logical operator or two operands separated by a logical operator. Logical expressions are used in statements like the IF-THEN-ELSE statement (See Section 4.3.) where a condition is tested to determine subsequent operations within the program. The operands in this case are usually relational expressions. Logical expressions can also be used with integer data. Logical operations on strings, however, are invalid. Logical operations on real data although allowed will probably not produce the expected results due to the internal representation of real numbers.

BASIC determines whether the condition is true or false by testing the result of the logical expression for non-zero and zero, respectively. (That is, a non-zero result is true, and a zero result is false.) BASIC supplies the value -1 for true when it evaluates a relational expression, but BASIC accepts any non-zero value when performing a test. Therefore in the following example

```
A% = 10%  
B% = NOT A%
```

both A% and B% can be true, if, for example, A% = 10 and B% = -11. Logical operators perform bit-wise operations. In the above example, B% is -11 (true) and not 0 (false). Logical operators are reserved words (See Appendix D.).

Table 1-6 provides a list of logical operators and their meaning.

ELEMENTS OF BASIC

Table 1-6
Logical Operators

Operator	Example	Meaning
NOT	NOT A%	The logical opposite of A%. If A% is false, NOT A% is true.
AND	A% AND B%	The logical product of A% and B%. A% AND B% is true only if both A% and B% are true.
OR	A% OR B%	The logical sum of A% and B%. A% OR B% is false only if both A% and B% are false; otherwise A% OR B% is true.
XOR	A% XOR B%	The logical exclusive OR of A% and B%. A% XOR B% is true if either A% or B% is true but not both; and false otherwise.
EQV	A% EQV B%	A% is logically equivalent to B%. A% EQV B% has the value true if A% and B% are both true or both false; and has the value false otherwise.
IMP	A% IMP B%	The logical implication of A% and B%. A% IMP B% is false if, and only if, A% is true and B% is false; otherwise the value is true.

Logical expressions are valid wherever real expressions are valid in BASIC. Both operands, however, must be integers.

ELEMENTS OF BASIC

Table 1-7 contains tables called truth tables. They describe graphically the results of the above logical operations on a bit-by-bit basis. Every possible combination of bits for A% and B% is given. In these tables, 1 equals true and 0 equals false.

Table 1-7
Truth Tables

A% 0 1	NOT A% 1 0	A% B% 0 0 0 1 1 0 1 1	A% OR B% 0 1 1 1
A% B% 0 0 0 1 1 0 1 1	A% AND B% 0 0 0 1	A% B% 0 0 0 1 1 0 1 1	A% EQV B% 1 0 0 1
A% B% 0 0 0 1 1 0 1 1	A% XOR B% 0 1 1 0	A% B% 0 0 0 1 1 0 1 1	A% IMP B% 1 1 0 1

Notice that the two operators XOR and EQV are exact opposites.

ELEMENTS OF BASIC

1.6.5 Functions

Functions are predefined sequences of instructions. A function can be accessed in a program by including its name in an expression. When BASIC evaluates an expression containing a function name, the sequence of instructions associated with that function name is automatically executed. BASIC has two types of functions: library functions and user-defined functions. Library functions are those supplied with the BASIC software; user-defined functions are those defined by you. (For more information on both types of functions, see Chapter 6.)

A function name is used just like a variable name. When you type a function name into a program, you are actually calling a pre-written routine. Consider the following example that uses a BASIC library function:

```
00010 PRINT SQR(49)
00020 END

READY
RUNNH
7
```

SQR is the function that defines the square root of the number within parentheses, in this case 49.

Using and creating BASIC functions are described in Chapter 6.

1.6.6 Evaluating Expressions

BASIC determines a value for expressions according to operator precedence. Each arithmetic, relational, and string operator in an expression has a predetermined position in the hierarchy of operators. The operator's position tells BASIC when to evaluate the operator in relation to the other operators in the same expression. Parentheses may be used to alter the order of precedence.

In the case of nested parentheses (one set of parentheses within another), BASIC evaluates the innermost expression first, then the one immediately outside it, and so on. The evaluation proceeds from the inside out until all parenthetical expressions have been evaluated. For example:


```
B = (25+(16*(9^2)))
```

Because (9^2) is the innermost parenthetical expression, BASIC evaluates it first, then $(16*81)$, and then $(25+1296)$.

ELEMENTS OF BASIC

Table 1-8 lists all operators in the order BASIC evaluates them:

Table 1-8
Operator Precedence

**, ^ - (unary minus) *, / +, - + (concatenation) all relational operators NOT AND OR, XOR IMP EQV	FIRST  LAST
--	--

Operators shown on the same line have equal precedence. Except for the operators + and *, BASIC evaluates operators of the same precedence level from left to right. All relational operators are on the same precedence level. The operators + and * are evaluated in any order since that is algebraically correct. For example, BASIC evaluates A^B^C as $(A^B)^C$.

BASIC evaluates expressions enclosed in parentheses first, even when the operator enclosed in parentheses is on a lower precedence level than the operator outside the parentheses. Consider the following example:

$$A = 15^2 + 12^2 * (35 - 8)$$

BASIC evaluates this expression in five ordered steps:

1. $15^2 = 225$ Exponentiation (leftmost expression)
2. $12^2 = 144$ Exponentiation
3. $35 - 8 = 27$ Subtraction
4. $144 * 27 = 3888$ Multiplication
5. $225 + 3888 = 4113$ Addition

1.7 ASSIGNING VALUES TO VARIABLES

The LET statement enables you to assign a value to a variable or list of variables. The LET statement has the following format:

[LET] variable(s) = expression

ELEMENTS OF BASIC

where:

LET	is an optional keyword.
variable(s)	is a variable or list of variables separated by commas.
expression	can be a string, numeric, relational, or logical expression.

The LET statement replaces the variable on the left of the equal sign (=) with the value on the right. Hence, the equal sign signifies the assignment of a value and not algebraic equality. BASIC evaluates the expression from left to right and assigns the values from right to left. Here is an example:

```
10 LET A=482.5
```

This statement assigns the value 482.5 to the variable A. You can also write the statement this way:

```
10 A=482.5
```

BASIC also evaluates any expression you assign:

```
10 A=(X+Y)-84
```

BASIC calculates the expression $(X+Y)-84$ and then assigns the resulting value to the variable A.

In addition, you can assign a value to more than one variable at a time, as in the following example:

```
10 A,B,C=64*82
```

This statement has the same effect as:

```
10 C=64*82
20 B=64*82
30 A=64*82
```

BASIC also converts the data type of the value (integer or real) to the data type of the associated variable. In the following example

```
10 A%, B, CZ, D = 9.5
```

is the same as

```
10 D = 9.5
20 CZ = 9
30 B = 9.5
40 A% = 9
```

Moreover, if you reference an array element as a variable, BASIC calculates the value of the subscript before assigning values to the other variables in the list. For example, the following lines assign the value 5 to the list element A(2), then change the value of I to 5:

```
10 I=2
20 A(I), I=5
```

ELEMENTS OF BASIC

You can also assign a string expression to a variable. You cannot, however, mix strings and numeric expressions in the same LET statement. If you do, BASIC prints an error message on the terminal. Here is an example of a string assignment:

```
10 A$="HELLO"  
20 PRINT A$  
30 END
```

RUNNH

HELLO

You can place a LET statement anywhere in a multi-statement line:

```
00010 PRINT "THE VALUE OF I IS:";\LET I=42\PRINT I
```

READY

RUNNH

THE VALUE OF I IS: 42

READY

1.8 ARRAYS

BASIC automatically reserves storage space for arrays with maximum subscripts of 10, for example, A(10) or A(10,10). If you require a larger amount of reserved space, define the dimensions with the DIM statement (See Section 1.8.1.). Conversely, if you do not require the space that BASIC supplies by default, save space for your program by using the DIM statement to reserve a smaller area.

When you establish the size of the array, BASIC stores the dimensions in a particular area for future reference. BASIC starts counting elements from 0, not 1; therefore, you have an additional element for a list and an additional row and column for a table. For example, dimensioning the array A(6) gives you seven storage areas in the list, not six:

```
ROW 1 A(0)  
    2 A(1)  
    3 A(2)  
    4 A(3)  
    5 A(4)  
    6 A(5)  
    7 A(6)
```

Array B(3,3) contains storage space for 16 elements. This is the layout of array B(3,3):

	COLUMN	1	2	3	4
ROW	1	B(0,0)	B(0,1)	B(0,2)	B(0,3)
	2	B(1,0)	B(1,1)	B(1,2)	B(1,3)
	3	B(2,0)	B(2,1)	B(2,2)	B(2,3)
	4	B(3,0)	B(3,1)	B(3,2)	B(3,3)

ELEMENTS OF BASIC

If you reference an array with the wrong number of subscripts, BASIC prints an error message.

Remember that it is possible to use the same alphanumerics to name both a simple variable and an array within the same program. But using the same name for two arrays with a different number of subscripts is invalid within the same program, that is, you could not name A(5) and A(3,4) in the same program.

1.8.1 The DIM Statement

The DIM statement allows you to define the dimensions of an array in your program. By using the DIM statement, you reserve storage space to be filled with values of either numeric or string data.

Because the DIM statement is not executed, you may place it anywhere in the program. It can also be one of several statements in a multi-statement line.

NOTE

It is good programming practice to dimension all arrays used by a program prior to referencing them. This practice not only makes the program easier to understand, but it also causes references to array elements to be executed faster at execution time.

The DIM statement has the following format:

DIM subscripted variable(s)

or

DIMENSION subscripted variable(s)

where:

subscripted variable(s) is one or more subscripted real, integer, or string variable(s) separated by commas (See Section 1.5.4.). The subscripts themselves must be unsigned real or integer constants.

Each subscripted variable name represents a distinct list or table. Because BASIC automatically reserves storage for arrays with maximum subscripts of 10, that is, A(10) and B(10,10), the DIM statement only needs to be used to reserve storage for lists with 11 or more elements and matrices with more than 121 elements, or, to save space, for lists with subscripts less than 10.

ELEMENTS OF BASIC

In the DIM list, you are specifying:

1. The name of the array
2. The number of subscripts (one or two)
3. The maximum value of each subscript
4. The data type of the array - real, integer, or string

Here is an example of a DIM statement:

```
10 DIM A(25),B(3,5),C$(7,16),D$(15)
```

No array can have more than two subscripts. If you do not specify a subscript in the second position, only one subscript is permitted for that variable name in future references. When using the DIM statement to set the maximum values for the subscripts, you are not obligated to fill every storage space you allocate.

Arrays are stored as if the rightmost subscript varied the fastest. For example, the following DIM statement sets up 20 contiguous storage areas:

```
10 DIM A(3,4)
```

The storage addresses look like this:

```
0,0  0,1  0,2  0,3  0,4
1,0  1,1  1,2  1,3  1,4
2,0  2,1  2,2  2,3  2,4
3,0  3,1  3,2  3,3  3,4
```

Notice that reading across left to right, the second subscript increases first.

As stated previously, the first element of every array begins with a subscript of 0. If you dimension a matrix C(6,10), you set up storage for 7 rows and 11 columns. The 0 element is illustrated in the following program:

```
LISNH
00010 REM MATRIX CHECK PROGRAM
00020 DIM C(6,10)
00030 FOR I=0 TO 6
00040 LET C(I,0)= I
00050 FOR J=0 TO 10
00060 LET C(0,J)= J
00070 PRINT C(I,J);
00080 NEXT J\PRINT\NEXT I
00090 END
```

ELEMENTS OF BASIC

READY
RUNNH

```
0 1 2 3 4 5 6 7 8 9 10
1 0 0 0 0 0 0 0 0 0 0
2 0 0 0 0 0 0 0 0 0 0
3 0 0 0 0 0 0 0 0 0 0
4 0 0 0 0 0 0 0 0 0 0
5 0 0 0 0 0 0 0 0 0 0
6 0 0 0 0 0 0 0 0 0 0
```

READY

A real or integer variable has a value of 0 until you assign it another value. A string variable has the value of a null string, that is, it has a non-printing value of 0.

You can also use the DIM statement to dimension string arrays, for example:

```
00010 DIM A$(5)
00020 INPUT A$(1),A$(2),A$(3),A$(4),A$(5)
00030 MAT PRINT A$(5)
00040 END
```

READY

RUNNH

? H,E,L,L,0

H
E
L
L
0

NOTE

There is a set of statements which operate on arrays and matrices. These are the MAT statements (See Chapter 7.) and the MOVE statement (See Section 9.7.). These statements do not recognize elements in the zero row and column of an array and, in some cases, destroy the contents of these elements. If you use these statements, it is important not to use array elements in the zero row and column for data.

CHAPTER 2

USING BASIC-PLUS-2

This chapter describes all of the BASIC commands. These commands enable you to perform a variety of tasks from BASIC command level, including: creating, executing, and saving BASIC programs. Also, this chapter describes the BASIC immediate mode. In immediate mode, you can test and debug BASIC programs. These topics are described in the following sections:

- 2.1 BASIC Command Level
- 2.2 Working with Existing Files
- 2.3 BASIC Immediate Mode
- 2.4 Nesting Immediate-Mode Statements
- 2.5 Immediate Mode and BASIC-PLUS-2 Commands

2.1 BASIC COMMAND LEVEL

BASIC has several commands that allow you to:

1. Create, edit, and manipulate files
2. Run BASIC programs
3. Enter system command level
4. Obtain information

The commands affect an entire BASIC program, and, therefore, are functionally different from the BASIC statements that constitute the program. For example, typing the RUN command (See Section 2.1.8.) causes BASIC to compile and execute the current program in memory. If, however, you include the RUN command as a statement in a BASIC program, an error is generated and the program cannot run. BASIC commands are executed after the RETURN key (or other line terminator) is pressed. BASIC statements, on the other hand, are executed after a program has been created and you issue the RUN command.

When BASIC is run (by typing BASIC RET), it allocates a portion of memory for your use. This portion of memory is known as the work area. Any work done on a program, such as creating, editing, and running BASIC programs, is done within the work area. You cannot, however, store your BASIC programs in the work area; devices such as disks provide a place for permanent storage.

To create a new file or program, to edit an existing file, or to run a file containing a program, the file must be established in the BASIC work area. The NEW command, OLD command, and the default to NONAME provide the means by which a file is named and established in the BASIC work area.

USING BASIC-PLUS-2

2.1.1 BASIC Command Format

Each BASIC command must be contained on a single line and terminated by a line feed, form feed, vertical tab, or carriage return/line feed. The command name may not contain spaces or tabs. The general command syntax is:

```
[ command      {line number(s)} ] (RET)
  {file spec}
```

where:

line number(s) represents one line number, or a series of up to 20 line numbers separated by commas (See description below.)

file spec is a TOPS-20 file specification (See Section B.4). Many commands assume the default type of .B20 unless otherwise specified.

(RET) indicates pressing the RETURN key (carriage return/line feed). In all BASIC-command formats and examples pressing the RETURN key at the end of the line is assumed.

A command can be either a fully specified command name, for example, RENAME, or a unique abbreviation with a minimum of three letters. The minimum accepted abbreviation is given with the command.

A line number list may consist of:

1. A single line number, for example, 80.
2. A pair of line numbers, separated by a hyphen, denoting an inclusive range, for example, 90-210.
3. A list of line numbers and ranges separated by commas, for example, 10, 85, 70-90, 60, 40-55. Line numbers can also be reversed, for example, 98-46.

If the command you type is not recognized, BASIC prints

? What

on the terminal. You must then retype the entire command string.

2.1.2 Entering BASIC Command Level (BASIC, HELP)

To enter BASIC command level and access the BASIC compiler, type:

```
@BASIC
```

This action clears the memory area and establishes contact with the BASIC compiler. When BASIC is ready to accept commands, it prints the prompt:

```
READY
```

After this message appears, you can type any acceptable BASIC-PLUS-2 command or statement and start inputting your program. If you need a summary of all BASIC commands, type:

```
HELP
```

USING BASIC-PLUS-2

The HELP command prints the following text:

```
READY
HELP
```

The following commands are available:

```
BUILD - Compile program and save code on disk
BYE - Exits from BASIC and loss the user off the system
CATalog - List files in directory
DEBUg - Compile and initialize (but do not run) program
DELeTe - Delete lines of the program
DO - Accept and execute lines from specified command file
GOOdbye - (Same as BYE.)
HELP - Type out this text
LIST - List lines of the program
LISTNH - Same as LIST but without header
LISNH - Same as LIST but without header
MODe DEF * - The program has GOTO's out of DEF *'S
MODe NODeF * - The opposite of MODE DEF * (default)
MONitor - Return to the EXEC (continue or reenter allowed)
NEw - SCRATCH program and set new program name
OLd - SCRATCH program and read in new one from disk
OLdLSA - Same as OLD for Line Sequenced ASCII files
QUIet - Turn off everything
QUIet HEAder - Turn off headers and trailers for LIST and RUN
QUIet CheCk - Turn off SYNTAX CHECKING
QUIet WARN - Silence compile time warnings
QUIet COMmand - Turn off echoing of command file input (default)
REName - Change name of current program
RESequence - Resequence line numbers in the program
RUN - Compile and run program
RUNNH - Same as RUN but without header
SAVe - Save copy of program on disk
SCRatch - Delete all lines of program
SCRatch ALL - Delete generated code (do not touch source program)
SCRatch IMMEdiate - Reset immediate mode (zero immediate mode variables)
SCRatch RESEt - Reset immediate mode (do not touch variables)
STATus - Type out status of QUIET and MODE flags
STArt - Initialize and run program (don't compile)
SYStem - Return to the EXEC (continue or reenter allowed)
UNSAve - Delete a disk file
VERbose - Turn on everything (default except for COMmand)
VERbose HEAder - Turn on headers and trailers for LIST and RUN
VERbose CHECk - Turn on SYNTAX CHECKING
VERbose WARN - Enable warning messages during compile
VERbose COMmand - Echo input from command file on user's terminal
```

```
READY
```

2.1.3 Returning to System-Command Level

To return to system command level, type:

```
SYS[TEM]
```

or

```
MON[ITOR]
```

USING BASIC-PLUS-2

The system responds with the prompt character (the "at" sign (@)), returns you to system-command level, and awaits a system command. When you are at system-command level, you can give only TOPS-20 system commands. BASIC commands will not be recognized.

If you plan to return to BASIC level after using the system facilities, be sure you do not issue a system command that will change or destroy the contents of memory, for example, the PLEASE command which destroys memory. (Refer to the TOPS-20 User's Guide for more information on system commands.)

When you decide to return to BASIC level, use one of the following:

```
ST [ART]
CON [TINUE]
REE [NTER ]
```

BASIC responds with the READY prompt, and you can continue working in BASIC command level. If memory was destroyed, type BASIC to reenter BASIC command level. Note that CONTINUE and REENTER preserve the current BASIC program but START does not.

2.1.4 Logging off from BASIC Command Level

To log off the system while in BASIC command mode, use:

```
[GOOD]BYE
```

The system responds by logging you off and printing information concerning your job. For example:

```
BYE
Killed Job 12, User FORREST, Account 1, TTY 221,
  at 9-APR-79 15:22:56, Used 0:0:1 in 0:0:20
```

All the files currently stored are saved on disk but the current contents of the work area will be lost.

2.1.5 Creating a Program (NEW)

To create a new source program in memory, use the NEW command, as follows:

```
NEW [file spec]
```

where:

file spec is a TOPS-20 file specification (See Section B.4). The default is NONAME.B20.1.

If you do not type a file spec after the NEW command, BASIC asks for one by printing:

```
New Program name--
```

If you do not want to specify a name, press the RETURN key again. BASIC will name your program NONAME.B20.

USING BASIC-PLUS-2

When BASIC executes the NEW command, the old contents in memory are lost. BASIC waits for your input by signaling READY. For example:

```
READY
NEW
New Program name--TEST.B20
READY
```

2.1.5.1 Syntax Checking - BASIC does a check of every program line you type, after you press the RETURN key. If BASIC finds an error in the syntax, that is, a misspelled statement keyword, a missing quotation mark, or any other syntax error, it prints the incorrect line and an error message.

For example:

```
10 PR "ABC"
10 PR "ABC"

? Statement not recognized

10 PRINT "ABC"
10 PRINT "ABC"
% Unterminated literal - treated as a comment
```

You can disable syntax checking with the QUIET command (See Section 2.2.9.1).

2.1.6 Checking Input (LIST)

To print a copy of the current program in memory, use the LIST command, as follows:

```
LIS[TNH] line number(s)
```

where:

LIS	is the accepted abbreviation; it prints the entire program and header.
NH	suppresses printing of the program name, the current time, and the current date. NH designates NO HEADER.
line number(s)	is 1 line number or a series of up to 20 line numbers.

You can disable the header with the QUIET command.

Arguments after the LIST command are optional. Lines and blocks of lines may be specified in any order. The lines print in the order you specify. You can reverse the order and even duplicate line numbers. Up to 20 arguments (19 commas) may be specified.

USING BASIC-PLUS-2

The following example prints a program in sequence:

```
READY
LISNH
00010 LET A=62
00020 LET B=75
00030 LET C=89
00040 LET D=A*B/C
00050 PRINT "D= ";D
00060 PRINT "THIS IS A TEST."
00070 PRINT "NEXT LINE OF TEST."
00080 IF D=245 GOTO 50
00090 END
```

READY

This next example demonstrates using the LISNH command to list a range of program lines in reverse order:

```
LISNH 90-10
00090 END
00080 IF D=245 GOTO 50
00070 PRINT "NEXT LINE OF TEST."
00060 PRINT "THIS IS A TEST."
00050 PRINT "D= ";D
00040 LET D=A*B/C
00030 LET C=89
00020 LET B=75
00010 LET A=62
```

READY

The following LIST command string prints certain lines in the program including duplicates:

```
LIST 50,70-90,30,10-40,5

TEST.B20
Thursday, March 2, 1979 15:26:24

00050 PRINT "D= ";D
00070 PRINT "NEXT LINE OF TEST."
00080 IF D=245 GOTO 50
00090 END
00030 LET C=89
00010 LET A=62
00020 LET B=75
00030 LET C=89
00040 LET D=A*B/C
```

READY

2.1.7 Correcting Mistakes

A program must be in memory before you can edit it. That is, you edit a program as it is built, or a saved program after it is brought into memory with the OLD command. You cannot edit a compiled program.

USING BASIC-PLUS-2

There are a number of ways to make corrections to a BASIC source program. These editing methods include typing:

1. The DELETE key and retyping
2. The CTRL/U (^U)
3. The CTRL/W (^W)
4. Line number and RETURN key
5. The DELETE command

2.1.7.1 Using the DELETE Key - As you create new programs, you can erase misspelled or incorrect lines with the DELETE key and then type the corrections. This must be done before pressing the RETURN key. For example, to correct a misspelled RESTORE statement, press the DELETE key once for each character you want to delete. Then type the correct characters on the same line. Note that the DELETE key erases characters one at a time in a reverse order from the last character typed. DELETE also prints a backslash after each deleted character. For example:

```
50 RETOREE\R\O\T\STORE
LINE# 50
00050 RESTORE

READY
```

NOTE

On a video display terminal, no backslashes are printed; the deleted characters are merely removed from the screen.

2.1.7.2 Deleting a Partially Typed Line - To delete an entire line that has not been entered into memory, use CTRL/U. A CTRL/U deletes unwanted lines and performs a carriage return/line feed. When you type a CTRL/U, the system prints 3X's on the terminal. If you have a display terminal, the system removes the line from the screen. For example:

```
10 PRINT "A = B * C";A = B * C XXX
```

USING BASIC-PLUS-2

2.1.7.3 Deleting a Field Within a Line - If you want to erase a field of a statement or command but not the entire line, type CTRL/W. CTRL/W erases everything between it and the last preceding space. The system prints an underline () but does not perform a carriage return/line feed as CTRL/U does. Instead, you can continue on the same line and type the correct information. (The underline may appear as a backarrow on some terminals.) If you have a display terminal, the system removes the field from the screen. For example:

```
10 IF A = 62 GOTO 20_30
LISNH 10
00010 IF A = 62 GOTO 30

READY
```

2.1.7.4 Deleting Lines by Line Number - To remove a line entered in memory, simply type the line number and the RETURN key. For example:

```
00110
```

You can replace a line in a program by typing it again.

```
10 LET A = 878
10 LET A = 90
LISNH 10
00010 LET A = 90

READY
```

2.1.7.5 Using the DELETE Command - To remove a specific line or lines from a program, use the DELETE command as follows:

```
DEL[ETE] line number(s)
```

where:

DEL is the accepted abbreviation.

line number(s) is one line number or a series of line numbers separated by commas (See Section 2.1.1).

For example:

```
DELETE 50
```

removes line 50 from the program.

```
DELETE 50,90
```

removes lines 50 and 90 from the program.

```
DELETE 50-100
```

removes lines 50 through and including line 100.

If you do not specify a line number with the DELETE command, no lines are removed and an error message is returned. If you specify a range of line numbers (20-75) and one of the specified lines does not exist, all of the lines within that range are removed anyway.

USING BASIC-PLUS-2

2.1.8 Executing a Program (RUN)

To execute a complete program in memory, use the RUN command as follows:

```
RUN[NH]
```

where:

RUN cannot be abbreviated.

NH indicates that no header will be printed.

The RUN command compiles (if necessary) and executes the program currently in the work area. For example:

```
READY
10 A = 25\B = 75
20 C = A + B
30 PRINT "C = ";C
40 END
RUNNH
C = 100
```

If the program can be run, BASIC executes it and prints the results requested by PRINT statements. The printing of results does not guarantee that the program is semantically correct (the results could be wrong), but it does indicate that no syntactical errors exist (missing line numbers, misspelled words). If errors of this type do exist, BASIC prints a message (or several messages) naming the errors. (Appendix A contains a list of the BASIC diagnostic messages.)

2.1.9 Interrupting Execution (CTRL/O) (CTRL/C)

If the results BASIC is printing seem to be incorrect, you may want to suppress printout or stop execution.

To suppress printout, type CTRL/O (^O). The program continues to run but the results are not printed on the terminal.

To stop program execution, type CTRL/C twice (^C^C). This action halts program execution and returns you to BASIC command level. BASIC responds with:

```
READY
```

2.1.10 Clearing Memory (SCRATCH)

To clear memory and start again, use the SCRATCH command as follows:

```
SCR[ATCH]
```

The SCRATCH command is a subset of the NEW command. It deletes the contents of the current work area and reinitializes the compiler. However, it retains the file specification associated with the work area. This file specification is the default for certain BASIC commands (for example, SAVE) that use a default file specification if one is not explicitly supplied.

USING BASIC-PLUS-2

2.1.11 Changing Line Numbers (RESEQUENCE)

Use the RESEQUENCE command to renumber a program so that you can insert new lines between existing lines of a program. The format of this command is as follows:

```
RES[EQUENCE][new line #, old line #, increment]
```

where:

RES is the accepted abbreviation.
new line # is the first line number of the new sequence.
old line # is the location of the starting place in the old file where resequencing begins.
increment is the increment between line numbers in the file.

The new line #, old line #, and increment need not be specified. The default values are:

```
new line # = 100  
old line # = beginning of current file  
increment = 10
```

If you specify only two out of the three arguments, for example

```
RES 50,25
```

the first argument is used as the new line #, and the second argument is used as the increment (not the old line #).

You can also resequence blocks of lines within a file. Use the following format to edit the current file:

```
RES new line #, line # - line #, increment
```

The two line numbers separated by a hyphen represent the beginning and ending line numbers of the block of lines to be resequenced.

Because the RESEQUENCE command destructively updates the program in the work area, the following procedure is recommended:

1. Save the current version of the source program (using the SAVE command) before issuing the RESEQUENCE command.
2. Never interrupt a RESEQUENCE command. If you do, you may leave the source program in an inconsistent state.

Note that RESEQUENCE does not allow you to delete lines or change the order of lines within a program. You can change the line number of a statement while changing the rest of the program but you cannot move the text to another part of the program. For example, given the following program

```
00010 A = 55  
00020 B = 72  
00030 PRINT A*B  
00040 END
```

USING BASIC-PLUS-2

RESEQUENCE cannot be used to produce:

```
00010 A = 55
00025 PRINT A*B
00030 B =72
00040 END
```

2.1.12 Saving a Program in a File (SAVE)

To store a source program on disk, use the SAVE command as follows:

```
SAV[E][file spec]
```

where:

SAV is the accepted abbreviation.

file spec is a TOPS-20 file specification (See Section B.4.). This is optional and, if it is not specified, the current program name and type will be used as set by the last NEW, OLD, or RENAME command.

The SAVE command copies the contents of the source file in the work area to the storage area associated with your directory or the directory specified in the file specification.

If a file already exists in storage with the name you specify in the SAVE command, BASIC replaces the old file with the one you are currently storing, and increments the generation number by 1.

This may, or may not, be what you intended to do. If you do not want to delete the old file, return to system command level by typing the SYS command. Then type UNDELETE and the name (including the generation number) of the file you want to retrieve. Return to BASIC command level by typing CONTINUE. For example:

```
CAT
MAIL.TXT.1
NONAME.B20.1

READY

OLD NONAME.B20.1

READY
5 PRINT "THIS IS THE NEW VERSION."
SAVE

READY
CAT

MAIL.TXT.1
NONAME.B20.2

READY
SYS
@UNDELETE (FILES) NONAME.B20.1
NONAME.B20.1 OK
@CONTINUE
```

USING BASIC-PLUS-2

```
READY
CAT

MAIL.TXT.1
NONAME.B20.1
NONAME.B20.2

READY
```

2.1.13 Saving an Executable Program in a File (BUILD)

The BUILD command always compiles the source program currently in the work area and saves an executable version of the program as a file.

The BUILD command has the following format:

```
BUI[LD] file spec
```

where:

BUI is the accepted abbreviation.
file spec is a TOPS-20 file specification (See Section B.4.).

The saved file that the BUILD command creates can be executed outside the BASIC environment. You can also execute a saved file from within the BASIC environment by using the CHAIN statement (See Section 5.2.).

The memory-image file that the BUILD command saves has a file type .EXE. To execute this file, return to system level and type the RUN command. For example:

```
BUILD PROGRAM

READY
MON
@RUN PROGRAM.EXE
```

2.2 WORKING WITH EXISTING FILES

The following sections describe the commands for bringing source programs (both line-sequenced ASCII files and regular ASCII files) back from disk storage to the work area, checking the files in your directory, renaming files, and deleting files from your directory.

2.2.1 Using Line-Sequenced ASCII Files (LSA)

To create a BASIC-PLUS-2 source program (or data for a program), you can use either the BASIC-PLUS-2 editor or the system editor, EDIT. The BASIC-PLUS-2 editor builds a regular ASCII file; EDIT builds a line-sequenced ASCII (LSA) file. Because LSA files were supported by a previous BASIC compiler as source input (.BAS), you may have programs or data files in LSA format. If you plan to use EDIT to create BASIC source programs or to read LSA data files with BASIC-PLUS-2, you should be aware of the constraints and problems you may encounter.

USING BASIC-PLUS-2

EDIT provides a line-sequence number for every line you type into the file.

For example:

```
@CREATE PROG.B20
Input: PROG.B20.1
00100   A=5
00200   B=6
00300   C=A*B
00400   PRINT C
00500   END$
*E
```

```
PROG.B20.1
@
```

The BASIC-PLUS-2 compiler and object-time system assume that all the ASCII files are regular ASCII files, not LSA files. When the BASIC compiler or a BASIC program reads an LSA file believing it to be a regular ASCII file, the first word (containing the line number) is not passed to the program. In particular, if you provide the name of an LSA file to the OLD command (See Section 2.2.3), the compiler will not receive the LSA line numbers. You will receive a syntax error unless the remainder of each line contains a line number.

This example shows an attempt to bring PROG.B20 (created with EDIT) into memory:

```
OLD PROG.B20
? Input line to OLD does not have line number

READY
```

To avoid this, you could create a file with extra line numbers and then remove the LSA line numbers when exiting from EDIT. For example:

```
@CREATE PROG.B20
Input: PROG.B20.2
00100   10 A=5
00200   20 B=6
00300   30 C=A*B
00400   40 PRINT C
00500   50 END$
*EU
```

```
PROG.B20.2
@
```

To avoid retyping LSA files you have already stored, use the OLDLSA command. See below.

2.2.2 Recalling Saved Source Programs in LSA Files (OLDLSA)

The OLDLSA command processes LSA files without losing the line-sequence numbers. The OLDLSA command has the following format:

```
OLDLSA [file spec]
```

USING BASIC-PLUS-2

where:

OLDLSA cannot be abbreviated.
file spec is a TOPS-20 file specification of a previously
 created LSA file. If no file is specified, BASIC
 prompts you to supply a file by printing: Old
 file name--.

Once you have the LSA file in memory, you can compile and execute it. Use the QUIET WARN command (See Section 2.2.9.1.) before bringing the file into the work area. Otherwise, you will receive the warning message:

```
%Illegal character in input stream - ignored
```

This warning is generated whenever an LSA file is brought into the work area; it can be safely ignored.

To convert LSA files to ASCII files, bring the LSA file into memory with OLDLSA, edit the file with the BASIC-PLUS-2 editor (if needed), then SAVE the file. Since any file generated by BASIC-PLUS-2 is a regular ASCII file, the file you SAVE is ASCII, not LSA.

2.2.3 Recalling Saved Source Programs in Regular ASCII Files (OLD)

To retrieve a source program stored on disk, use the OLD command as follows:

```
OLD [file spec]
```

where:

OLD may not be abbreviated.
file spec specifies the TOPS-20 file specification of the
 file you want to retrieve from storage.

When you type the OLD command, the old contents of the work area are lost, BASIC searches for the file name in your directory, and, if it is there, brings it into memory. If the file name is not in your directory, BASIC prints:

```
?File not found
```

If you do not specify a file name, BASIC prompts you with:

```
Old file name--
```

If you press the RETURN key again, BASIC searches for a file called NONAME.B20. If such a file does not exist in your directory, BASIC prints:

```
?File not found
```


USING BASIC-PLUS-2

2.2.4 Renaming a Program (RENAME)

To change the name of a program currently in memory, use the RENAME command as follows:

```
REN[AME][file spec]
```

where:

REN is the accepted abbreviation.

file spec specifies the new name and type for the file. The default is NONAME.B20.

2.2.5 DELETING a SAVED PROGRAM (UNSAVE)

To delete a file from storage, use the UNSAVE command as follows:

```
UNS[AVE][file spec]
```

where:

UNS is the accepted abbreviation.

file spec specifies the file you want to delete.

The UNSAVE command deletes the specified file. There is no need to bring it into the work area. If no file specification is given, the current file name and type is assumed.

If there is no program or file currently in memory and you do not specify a file name, BASIC prints:

```
?File not found
```

If you change your mind about deleting a file, you can retrieve it before you log off (if the operator has not expunged files). Refer to the TOPS-20 User's Guide for information on expunging files.

To retrieve a file you have UNSAVED, type:

```
SYS
@UNDELETE file name.typ.gen#
filename.typ.gen# OK
@CONT
READY
```

2.2.6 Checking Your Directory (CATALOG)

To check the files you have stored in your directory, use the CATALOG command as follows:

```
CAT[ALOG]
```

USING BASIC-PLUS-2

where:

CAT is the accepted abbreviation.

The CATALOG command prints the names of files that reside in your directory. The file names are listed in alphabetic order on your terminal.

For example:

```
READY
CAT
FILE.B20.1
FILE1..5
LOGIN.CMD.2
LOGIN.QMD.2
LOGIN.REL.1
MOD.B20.1
```

2.2.7 Combining Programs (WEAVE)

To combine the lines of a program in the current work area with a program in storage, use the WEAVE command as follows:

```
WEA[VE]file spec
```

where:

WEA is the accepted abbreviation.

file spec specifies the TOPS-20 file specification of the file in storage.

If you do not specify a file spec, BASIC prompts with:

```
WEA
Old file name--
```

Using the WEAVE command has the same effect as typing the lines of the OLD program into the current work area. Consider this example:

```
OLD A
READY
LISNH
00010 B=2
00015 D=4
00020 F=6
00025 H=8
00030 END
READY
NEW B
READY
5 A=1
11 C=3
16 E=5
21 G=7
WEAVE A
```

USING BASIC-PLUS-2

```
READY
LIST
B.B20
Thursday, June 2, 1979 15:41:47
```

```
00005  A=1
00010  B=2
00011  C=3
00015  D=4
00016  E=5
00020  F=6
00021  G=7
00025  H=8
00030  END
```

```
READY
```

If two lines in the programs being combined have the same line number, the program in storage has precedence and overwrites the line in the work area.

For example, Line 10 in PROGRAM1 (program in storage) has precedence over line 10 in PROGRAM2 (program in the work area).

```
OLD PROGRAM1
READY
LIST
PROGRAM1.B20
Thursday, June 2, 1979 15:42:16
00010 PRINT "THIS PROGRAM HAS PRECEDENCE."
00020 END
READY
NEW PROGRAM2
READY
10 PRINT "THIS LINE WILL BE REPLACED."
WEAVE PROGRAM1
READY
LIST
PROGRAM.B20
Thursday, June 2, 1979 15:42:46
00010 PRINT "THIS PROGRAM HAS PRECEDENCE."
00020 END
READY
```

2.2.8 Issuing BASIC Commands from a File (DO)

To instruct BASIC to take its input from a specified command file rather than from your terminal, use the DO command. The command file may contain any BASIC command or immediate-mode statement that BASIC would accept from the terminal. BASIC will not accept another DO command from within the originally referenced command file. Nested command files are therefore invalid.

USING BASIC-PLUS-2

Use a text editor to create a command file. Include in the command file any BASIC commands (except a DO command) or immediate-mode statements in the order that you want BASIC to process them. Use the DO command as follows:

```
DO [file spec]
```

where:

DO cannot be abbreviated.

file spec specifies the TOPS-20 file specification of the file in storage. This file contains the commands you want BASIC to process. If you do not supply a file type, BASIC uses the default type .CMD.

If you do not specify a file spec, BASIC prompts with:

```
DO  
Old filename--
```

If any errors occur while BASIC is processing a command file, BASIC stops processing the command file, reports the error, echoes the line that contains the error, and returns the READY prompt. This prompt signals that BASIC is again ready to accept commands from your terminal.

In this example, the command file is built using the text editor EDIT.

```
@  
@CREATE (FILE) BUILD-PROGRAM.CMD  
Input: BUILD-PROGRAM.CMD.1  
00100 10 !THIS PROGRAM DEMONSTRATES THE USE  
00200 20 !OF COMMAND FILE TO CREATE AND RUN  
00300 30 !A PROGRAM IN BASIC.  
00400 40 PRINT 'FIND THE SQUARE ROOT OF';  
00500 50 INPUT A  
00600 60 B=SQR(A)  
00700 70 PRINT 'THE SQUARE ROOT OF',A,'IS:',B  
00800 80 END  
00900 LIST  
01000 BUILD SQUARE-ROOT.EXE  
01100 RUN  
*E
```

```
BUILD-PROGRAM.CMD.1
```

USING BASIC-PLUS-2

Next, BASIC is run and the command file BUILD-PROGRAM.COMD is specified using the DO command.

```
@BASIC

READY
DO
Command file name--BUILD-PROGRAM.COMD

NONAME.B20
Thursday, March 1, 1979 11:12:40

00010 !THIS PROGRAM DEMONSTRATES THE USE
00020 !OF A COMMAND FILE TO CREATE AND RUN
00030 !A PROGRAM IN BASIC.
00040 PRINT 'FIND THE SQUARE ROOT OF',;
00050 INPUT A
00060 B=SQR(A)
00070 PRINT 'THE SQUARE ROOT OF';A;' IS';B
00080 END

Compile time: 0.077 secs      Elapsed time: 0:00:01

NONAME.B20
Tuesday, June 26, 1979 09:03:56

FIND THE SQUARE ROOT OF ? 16
THE SQUARE ROOT OF 16 IS 4

Run time: 0.149 secs        Elapsed time: 00:00:09
```

From the above results, you can see that BASIC first compiled the immediate-mode statements (lines 400-800), then listed the program (line 900), then saved the executable version of the program in a file called SQUARE-ROOT.EXE (line 1000), and finally ran the program (line 1100). A TOPS-20 DIRECTORY command finds the new file in the connected directory:

```
READY
MON
@DIR SQUARE-ROOT.EXE

PS:<FORREST>
SQUARE-ROOT.EXE.1
@
```

2.2.9 BASIC Compiler Switches (STATUS)

BASIC provides certain commands that disable and enable various compiler switches.

The STATUS command lists the current status of the compiler switches. The STATUS command has the following format:

```
STAT[US]
```

where:

STAT is the accepted abbreviation.

USING BASIC-PLUS-2

For example, when logging in to the system and accessing BASIC, the default status of the compiler switches is as follows:

```
STATUS
VERBOSE CHECK
VERBOSE HEADER
VERBOSE WARN
QUIET COMMAND
MODE NODEF *
```

READY

In the above list, VERBOSE means that the switch is enabled and QUIET means that the switch is disabled. The switches are as follows:

CHECK	enables line-by-line syntax checking of BASIC source programs. The default status of CHECK is VERBOSE.
HEADER	enables printing of program name, current date, and time statistics. The default status of HEADER is VERBOSE.
WARN	enables printing of compiler warning messages on the terminal. The default status of WARN is VERBOSE.
COMMAND	enables echoing of BASIC command file input. The default status of COMMAND is QUIET.

MODE NODEF * is the default setting specifying the compiler mode. For information on controlling compiler mode, see Section 2.2.9.3.

2.2.9.1 Disabling Compiler Switches - The QUIET command disables the printing of program headers, compiler warning messages, syntax checking, and command file input.

The QUIET command has the following format:

$$\text{QUI}[\text{ET}] \left[\left(\begin{array}{l} \text{CHE}[\text{CK}] \\ \text{HEA}[\text{DER}] \\ \text{WAR}[\text{N}] \\ \text{COM}[\text{MAND}] \end{array} \right) \right]$$

where:

QUI	is the accepted abbreviation.
CHE[CK]	disables line-by-line syntax checking.
HEA[DER]	disables program headers.
WAR[N]	disables compiler warning messages.
COM[MAND]	disables echoing of command file input.

When you log into the system, the CHECK, WARN, and HEADER features are enabled (VERBOSE); the COMMAND feature is disabled (QUIET). By typing QUIET without arguments, you disable all switches.

USING BASIC-PLUS-2

2.2.9.2 Enabling Compiler Switches - To enable syntax checking, program headers, compiler warning messages, and echoing of command file input, use the VERBOSE command.

The VERBOSE command has the following format:

$$\text{VER}[\text{BOSE}] \left[\left(\begin{array}{l} \text{CHE}[\text{CK}] \\ \text{WAR}[\text{N}] \\ \text{HEA}[\text{DER}] \\ \text{COM}[\text{MAND}] \end{array} \right) \right]$$

where:

VER	is the accepted abbreviation.
CHE[CK]	enables line-by-line syntax checking.
HEA[DER]	enables program headers.
WAR[N]	enables compiler warning messages.
COM[MAND]	enables echoing of command file input.

By typing VERBOSE without arguments, you enable all four features.

2.2.9.3 Controlling Compiler Mode - The MODE command enables and disables compiler mode.

The MODE command has the following format:

$$\text{MODE} \left\{ \begin{array}{l} \text{DEF *} \\ \text{NODEF *} \end{array} \right\}$$

where:

DEF *	turns on the DEF * compile mode.
NODEF *	turns off the DEF * compile mode. This is the default.

DEF* is a special type of multi-line user-defined function (See Section 6.7.3).

To determine the current compiler mode (MODE DEF * or MODE NODEF *) use the STATUS command described in Section 2.2.9.

2.2.10 Listing a Compiled Program (MLIST)

The MLIST command outputs a listing of the compiled code for the current program to the specified TOPS-20 file. The format for the MLIST command is as follows:

MLIST

USING BASIC-PLUS-2

If you type MLIST RET, BASIC prompts you with: Output File--.
Respond to this prompt by typing the file specification where you want the program listed. If you do not specify a file specification and file, BASIC places the compiled code in a file that has the filename currently associated with the work area, and the file type .LST.

The following example shows a compiled program being listed on the terminal:

```

READY
00010 A=3
00020 END
RUNNH

READY
MLIST
Output file: TTY:
3000/ 201040003420      MOVEI      1,3420
3001/ 265240440031      JSP        5,MAININIT

00010  A=3

3002/ 200300003400      MOVE      6,3400
3003/ 202314000001      MOVEM     6,1(14)

00020  END

3004/ 201700003402      MOVEI     AF,3402
3005/ 260740440101      PUSHJ    P,0END

3400/ 202600000000      MOVEM    14,0
3401/ 0                  (@       0,0
3402/ 0                  (@       0,0
3403/ 0                  (@       0,0
3404/ 466031147100      DRMM     0,@147100(11)
3405/ 502451743644      HLLM    11,(11)
3406/ 406320000000      ANDM    6,@0
3407/ 2003404          (@      0,3404(VREG)
3410/ 12003002          (@      0,3002(12)
3411/ 24003004          (@      0,@3004(4)
3412/ 5000000000       0        0,0
3413/ 1                  (@      0,1
3414/ 3403              (@      0,3403
3415/ 3404              (@      0,3404
3416/ 3407              (@      0,3407
3417/ 777773000000     P,@0(13)
3420/ 422400003412      ANDCM    10,3412
3421/ 0                  (@      0,0
3422/ 422440000000      ANDCM    11,0
3423/ 422540000000      ANDCM    13,0
3424/ 414100000000      SETM    VREG,0
3425/ 0                  (@      0,0

READY

```


USING BASIC-PLUS-2

2.2.11 A Sample Program

The following example illustrates the use of BASIC-PLUS-2 under a timesharing system:

```
SYSTEM 2116 THE BIG ORANGE WELCOMES YOU, TOPS-20 Monitor 3A(2013)
@LOGIN (USER) FORREST (PASSWORD) (ACCOUNT) 10400
  Job 37 on TTY220 7-May-79 15:21:49
@BASIC
```

```
READY
OLD SORT
```

```
READY
LIST
```

```
SORT.B20
Monday, May 7, 1979 15:22:22
```

```
00010 DIM SORT(10)      !ELEMENTS TO SORT
00020 INPUT "NUMBER OF ENTRIES ";CNT
00025 INPUT SORT(I) FOR I = 1 TO CNT
00030 SORT.FLG = -1
00035  WHILE SORT.FLG<>0
00040    SORT.FLG=0
00045    FOR I = 1 TO CNT -1 !FOR EACH ELEMENT EXCEPT LAST
00050      GOTO 80 IF SORT(I)<=SORT(I+1)
00055      !EXCHANGE ELEMENTS I AND I+1
00060      T=SORT(I)
00065      SORT(I)=SORT(I+1)
00070      SORT(I+1)=T
00075      SORT.FLG=-1 !NOT DONE YET
00080    NEXT I
00085  NEXT
00090 PRINT SORT(I),FOR I = 1 TO CNT
99999 END
```

```
READY
RUN
```

```
SORT.B20
Monday, May 7, 1979 15:22:29
```

```
NUMBER OF ENTRIES ? 5
? 72
? 31
? 16
? 65
? 23
16          23          31          65          72
```

```
Compile time: 0.206 secs
run time: 0.181 secs          Elapsed time: 0:0:21
```

```
READY
```

USING BASIC-PLUS-2

This program accepts up to ten numbers as input, sorts them by size, and prints them in ascending order on the terminal. The procedure by which the program is typed into the system and executed is detailed below. The explanations are keyed to the commands and program lines.

OLD SORT	The program named SORT already resides in storage. The OLD command brings the program into memory.
READY	This is printed by BASIC and indicates that the compiler is prepared to accept input. It also indicates that the previous command (OLD) has been successfully executed.
LIST	This command prints the program in the proper sequence. Notice the preceding 0's in the line numbers.
Line 10	This program line dimensions an array named SORT. Although 10 is given as the dimension, the array is dimensioned for 11 elements because BASIC begins array storage at 0. Notice that the array name is 4 characters. BASIC-PLUS-2 allows variable and array names of up to 30 characters. Also, notice that the comment, ELEMENTS TO SORT, is delimited by an exclamation point.
Line 20	This is an INPUT statement that prints requests for the number of elements that will be sorted.
Line 25	Another INPUT statement prints a request for the array elements. The FOR modifier ensures that the correct number of elements is asked for by the prompt.
Lines 30-85	These lines contain a sorting procedure composed of a bubble-sort algorithm.
Line 90	This line prints the sorted array elements.
Line 99999	This is the highest line number possible in a BASIC-PLUS-2 program. It contains the END statement.
RUN	The RUN command causes execution of the program. The RUN command also causes the program name and date (header) and the compiler and run-time statistics (trailer) to be printed. To suppress headers and trailers, append NH (No Header) to the RUN command.
READY	BASIC prints this message to inform you that the RUN command has been executed and that BASIC is again ready to accept commands.

Programs can also be submitted through batch. See Appendix C for information on submitting batch jobs.

USING BASIC-PLUS-2

2.3 BASIC IMMEDIATE MODE

Immediate mode is an alternative to writing and executing full programs in BASIC. In immediate mode, you can type statements ordinarily found in a program and have BASIC execute them as soon as you terminate the line. The advantages of immediate mode are:

1. The ability to debug programs by stopping and requesting information during execution
2. The use of a powerful calculator (the CPU) at your disposal

With BASIC immediate mode, you can examine the state of a program at various points by stopping execution and printing the values of variables. You can also build a program consisting of library routines, perform calculations with immediate-mode variables using that library, and print the results on the terminal.

The BASIC commands used for immediate-mode operations are:

CONTINUE	SCRATCH ALL
DEBUG	SCRATCH IMMEDIATE
RUN	SCRATCH RESET
SCRATCH	START

These commands affect compilation and the state of immediate-mode variables and statements.

2.3.1 Immediate-Mode Statements

Immediate mode allows you to use not only the BASIC commands listed above but also the following BASIC statements:

CALL	NEXT
CHAIN	NODATA
CHANGE	ON GOSUB
CLOSE	ON GOTO
DELETE	ONERROR
FIND	OPEN
FNEXIT	PAGE
FOR	PRINT
GET	PUT
GOSUB	RANDOMIZE
GOTO	READ
IF	REMARK and !
IFEND	RESET
IFMORE	RESTORE
INPUT	RETURN
INPUT LINE	SCRATCH
KILL	SLEEP
LET	STOP
LINPUT	SUBEXIT
MARGIN	TIME
MAT Arithmetic	UNSAVE
MAT Initialization	UNTIL
MAT Input/Output	UPDATE
MOVE	WAIT
NAME-AS	WHILE

You can also use statement modifiers and function calls in immediate mode.

USING BASIC-PLUS-2

The following statements are invalid in immediate mode and, if used as an immediate-mode command, cause an error message to appear on the terminal:

COMMON	FNEND
DATA	IMAGE
DEF	MAP
DEF*	RESUME
DIMENSION	SUB
END	SUBEND

Also, any action creating an array is invalid in immediate mode. If you plan to use the CHANGE statement in immediate mode, you must define the array within a program and the program must be in a context that can access that array (See Section 2.3.2.).

BASIC distinguishes between lines entered for later execution and those entered for immediate execution solely by the presence (or absence) of a line number. Statements that begin with line numbers are stored; statements without line numbers are executed immediately. Therefore, the following line produces no action on the terminal:

```
10 PRINT "THIS IS A PROGRAM LINE."
```

However, this line is executed immediately:

```
PRINT "THIS IS IMMEDIATE MODE."  
THIS IS IMMEDIATE MODE.
```

2.3.2 Immediate-Mode Contexts

You may use immediate mode only when certain conditions are present. The conditions form an immediate-mode "context." The context determines which lines in a program you can access (if any) with immediate-mode commands and statements. It also determines whether or not the variables you use are immediate-mode variables or variables that have a value within the program.

The four immediate-mode contexts are:

1. No program - you have not compiled or executed a program.
 - a. All immediate-mode references to line numbers are invalid since there is no program.
 - b. All variables referenced in this context are defined as immediate-mode variables.
2. STOP - the program being executed is currently at a STOP statement.
 - a. The context is that of the line at which the STOP occurred, that is, main program, subprogram, or function.
 - b. References to variables and line numbers are treated as they would be at that line.
 - c. Variables not defined in the context of the line are considered immediate-mode variables. These immediate-mode variables are treated as globals by later immediate-mode statements.

USING BASIC-PLUS-2

- d. References to line numbers outside this context are invalid.
3. Compiled Program - the program has been compiled with the DEBUG command, but not executed. You are in the context of the main program of that compilation. You can:
 - a. Reference main-program variables (though their values will be zero).
 - b. Transfer to lines in the main program.
 - c. Define and reference immediate-mode variables.
 4. Executed Program - the program has been executed. You are in the context in which the END statement was executed. You can:
 - a. Reference main program variables. In this context, variables retain the values they were assigned in the program.
 - b. Transfer to other lines in the program.
 - c. Define and reference immediate-mode variables.

Keep in mind that the same rules concerning transfers in a BASIC program apply in immediate-mode operations. When your program is in an immediate-mode context, you cannot transfer into and out of a subprogram or user-defined function (except for DEF* functions, see Section 6.7.3).

2.3.3 Immediate-Mode Variables

During immediate-mode operations, you can use variables that exist within a program. The context determines which variables are accessible. However, you can also assign values to variables other than the ones defined in the program. These are immediate-mode variables. For example:

```
READY
10 A=90\B=72\C=A*B
20 PRINT "C = ";C
30 END
RUNNH
C = 6480
```

```
READY
GOTO 10
C = 6480
```

```
D=9
C=A*B/D
PRINT D
9
PRINT C
720
```

USING BASIC-PLUS-2

The first three variables (A, B, and C) are program variables; variable D is an immediate-mode variable. A variable not assigned in the current context of the program is an immediate-mode variable.

You can use immediate-mode variables in subsequent immediate-mode statements and commands until you type one of the following commands that erases the immediate-mode buffer:

BUILD	SCRATCH
DEBUG	SCRATCH ALL
RESEQUENCE	SCRATCH IMMEDIATE
RUN	START

The rules for determining local and global variables in a BASIC program also apply in immediate-mode situations. If you are at a STOP in a main program, all subprogram variables are inaccessible and vice versa. You can, however, assign a value to an immediate-mode variable while within the subprogram or function (even if a variable of the same name exists in the main program).

Once you return to the main program, the immediate-mode variable is erased. For example:

```
READY
LISNH
00010 A=5\B=10\C=15
00020 CALL ARG (B)
00030 CALL ARG (C)
00040 PRINT "A= ";A;"B= ";B;"C= ";C
00050 END
00060 SUB ARG (D)
00070 A=30
00080 STOP
00090 D=D*A
00100 SUBEND
```

```
READY
A=50
RUNNH
STOP at line 00080 of ARG
```

```
READY
PRINT A
  30
A=50
PRINT A
  50
GOTO 90
STOP at line 00080 of ARG
CONTINUE
A=  5 B=  500 C=  450
```

```
READY
```

2.3.4 Halting an Immediate-Mode Program (STOP)

The STOP statement has the following format:

```
STOP
```

This statement causes program execution to halt, at which point BASIC prints the message:

```
STOP at line n of program segment
```

USING BASIC-PLUS-2

where:

n is the line number of the STOP statement.
Files remain open.

program segment is either the words MAIN PROGRAM, a function
name, or a subprogram name.

When the program stops, you can type immediate-mode statements and commands and refer to lines in that context. For example:

```
STOP at line 00050 of MAIN PROGRAM
PRINT A
92
```

the value contained in the variable A is 92 at line 50.

If the STOP occurs within a subprogram or user-defined function, you cannot refer to any line outside that context.

To continue program execution from the point where the program left off, use the CONTINUE command as follows:

```
CON[TINUE]
```

where:

CON is the accepted abbreviation.

The CONTINUE command is valid only after a STOP statement has been executed or after an error occurs that is trappable but no ONERROR statement is in effect.

In the first case, the program continues from the last STOP executed by BASIC, including STOP statements issued from immediate mode. For example:

```
READY
LISNH
00010 PRINT "ABCDE" \ STOP \ PRINT "FGHIJ"
00020 END
```

```
READY
RUNNH
ABCDE
STOP at line 00010 of MAIN PROGRAM
```

```
READY
CONTINUE
FGHIJ
```

```
READY
```

USING BASIC-PLUS-2

In the second case, where a trappable error occurred and no ONERROR statement was in effect, you can correct the error and then type CONTINUE so that the program can resume execution. For example:

```
SCR

READY
X=-1
PRINT X
-1
A=LOG(X)

? 53 Attempt to take LOG of negative argument in immediate mode
PRINT X
-1
X=4
CONTINUE

READY
PRINT A
1.386294
PRINT EXP (A)
4
```

2.3.5 Clearing the Work Area

The SCRATCH command enables you to clear all or a portion of the work area. The SCRATCH command has four formats, as follows:

$$\text{SCR}[\text{ATCH}] \left[\begin{array}{l} \text{(no argument)} \\ \text{RESET} \\ \text{IMM}[\text{EDIATE}] \\ \text{ALL} \end{array} \right]$$

where:

SCR no argument is the accepted abbreviation for the SCRATCH command. The SCRATCH command, with no argument, deletes the current source and executable versions of the program, all immediate-mode statements, and all immediate-mode variables. The SCRATCH command does not close open files.

SCR RESET is the accepted abbreviation for the SCRATCH RESET command. The SCRATCH RESET command deletes all immediate-mode statements, but does not delete any immediate-mode variables. The SCRATCH RESET command does not delete the source or executable versions of the program. The SCRATCH RESET command does not close any open files.

The SCRATCH RESET command is used primarily to delete nested immediate-mode statements. (Nesting immediate-mode statements is described in Section 2.4.)

USING BASIC-PLUS-2

SCR IMM is the accepted abbreviation for the **SCRATCH IMMEDIATE** command. The **SCRATCH IMMEDIATE** command deletes all immediate-mode statements and variables. It does not delete the source or executable versions of the program. The **SCRATCH IMMEDIATE** command does not close any open files.

NOTE

The **RESEQUENCE** command (See Section 2.1.11.) automatically performs a **SCRATCH IMMEDIATE** command.

SCR ALL is the accepted abbreviation for the **SCRATCH ALL** command. The **SCRATCH ALL** does not delete the current source program, but does delete all immediate-mode statements and all immediate-mode variables. The **SCRATCH ALL** command also re-initializes the compiler, thereby erasing the executable version of the source program. The **SCRATCH ALL** command terminates all open files without closing them, thereby causing any data output to these files to be lost.

If you receive an error message notifying you of an internal inconsistency, use the **SCRATCH ALL** command to rectify the situation.

In the first example, the **SCRATCH** command clears the current source program from the work area:

```
LISNH  
00010 PRINT "THIS IS A SOURCE PROGRAM"  
00020 END
```

```
READY  
SCRATCH
```

```
READY  
LISNH
```

```
READY
```

USING BASIC-PLUS-2

In the second example, the SCRATCH RESET command deletes nested immediate-mode statements, eliminating the need for the user to issue successive CONTINUE commands to get back to the top level (first immediate-mode statement):

```
READY
A=15
B=37 \ STOP \ PRINT "HI"
STOP in immediate mode
PRINT B
  37
PRINT A
  15
SCRATCH RESET
```

```
READY
PRINT A,B
  15          37
CONTINUE
? Cannot CONTINUE, use RUN or DEBUG.
```

```
READY
```

In the third example, the SCRATCH IMMEDIATE command deletes immediate-mode variables:

```
LISNH
00010 PRINT "AT LINE 10"
00020 END
```

```
READY
RUNNH
AT LINE 10
```

```
READY
GOTO 10
AT LINE 10
A=15
PRINT A
  15
SCRATCH IMMEDIATE
```

```
READY
PRINT A
PRINT A
% Reference to uninitialized immediate mode variable A
  0
GOTO 10
AT LINE 10
```

USING BASIC-PLUS-2

In the fourth example, the SCRATCH ALL command deletes the compiled version of the program and all immediate-mode variables:

```
LISNH
00010 PRINT "AT LINE 10"
00020 END

READY
RUNNH
AT LINE 10

READY
A=5
PRINT A
5
SCRATCH ALL

READY
START

NONAME.B20
Wednesday, August 8, 1979 11:19:17

? Cannot START, use RUN or DEBUG.

READY
PRINT A
PRINT A
% Reference to uninitialized immediate mode variable A
0
```

2.3.6 Using the START Command

The START command executes a compiled program. If you have edited the program since the last RUN, or have not compiled or executed the program, BASIC prints a message telling you to use the RUN or DEBUG command as shown in the example below.

The START command has the following format:

```
STA[RT]
```

where:

STA is the accepted abbreviation.

The START command causes a SCRATCH of all immediate-mode variables and statements but does not cause a recompile of the program.

USING BASIC-PLUS-2

For example:

```
LISNH
00010 REM We cannot START this program until it has
00020 REM been compiled
00030 PRINT "AT LINE 10"
00040 END
```

```
READY
START
```

```
TEST.B20
Thursday, August 9, 1979 16:20:49
```

```
? Cannot START, use RUN or DEBUG.
```

```
READY
RUN
```

```
TEST.B20
Thursday, August 9, 1979 16:20:56
```

```
AT LINE 10
```

```
Compile time: 0.089 secs
Run time: 0.115 secs           Elapsed time: 0:00:05
```

```
READY
START
```

```
TEST.B20
Thursday, August 9, 1979 16:21:08
```

```
AT LINE 10
```

```
Run time: 0.129 secs           Elapsed time: 0:00:04
```

```
READY
```

2.3.7 Using the DEBUG Command

THE DEBUG command compiles a program, erases all immediate-mode variables, stops before the first line of the program, and then prints READY.

At this point, you can set up initial values for variables and then "GOTO" any line number in the program. However, any command that modifies the work area terminates the DEBUG command.

The DEBUG command has the following format:

```
DEB[UG]
```

where:

DEB is the accepted abbreviation.

USING BASIC-PLUS-2

For example:

```
00010 PRINT "AT LINE 10"  
00020 PRINT "AT LINE 20"  
00030 PRINT "AT LINE 30"  
00040 END
```

```
READY  
GOTO 20  
AT LINE 20  
AT LINE 30  
25 PRINT A  
DEBUG
```

```
READY  
A=137.5  
GOTO 20  
AT LINE 20  
137.5  
AT LINE 30
```

2.4 NESTING IMMEDIATE-MODE STATEMENTS

If you execute a STOP statement with an immediate-mode command, that is, if you call a function from immediate mode that contains a STOP statement, at this point you can enter other immediate-mode statements, causing a nest. For example, the following program is executed and the GOTO statement is used to create a nest of immediate-mode statements:

```
00020 B$= 'THIS IS A TEST'  
00030 GOTO 99  
00038 STOP  
00040 GOTO 99  
00055 STOP  
00099 END
```

```
READY  
GOTO 55  
STOP at line 00055 of MAIN PROGRAM  
PRINT B$  
THIS IS A TEST  
GOTO 38  
STOP at line 00038 of MAIN PROGRAM
```

The fixed buffer size of all immediate-mode code restricts the number of immediate-mode statements that can be active at one time. This buffer holds only 128 executable instructions. Therefore

$A(I) = B(I) + \text{FNX}(A)$

USING BASIC-PLUS-2

Notice that BASIC prints the following message when you run out of buffer space:

```
?Too many immediate mode statements - use SCRATCH RESET
```

If you want to erase all immediate-mode code but you want to retain the present variable values, use the SCRATCH RESET command as shown in the last example.

2.5 IMMEDIATE MODE AND BASIC-PLUS-2 COMMANDS

Certain BASIC-PLUS-2 statements, such as SCRATCH, are also the names of BASIC-PLUS-2 commands. When you type a statement or command in immediate mode, BASIC-PLUS-2 first looks for a command of that name. In the case of SCRATCH, for example, you would lose your compiled program by executing the SCRATCH command instead of performing a SCRATCH on the file with the BASIC-PLUS-2 SCRATCH statement.

You can avoid this problem by preceding the SCRATCH statement with a backslash, for example: \SCRATCH

The immediate-mode statements that are also names of BASIC-PLUS-2 commands are:

```
DELETE  
RESET  
SCRATCH
```

In addition, although any BASIC-PLUS-2 command name (including the ones reserved for DIGITAL) can be used as a variable name in a program, you may assign such a variable a value from immediate mode only if you precede it with a backslash (\) or a LET. For example:

```
\LIST = 75  
PRINT LIST  
75
```

CHAPTER 3

INPUT AND OUTPUT TO THE TERMINAL

This chapter describes the statements that enable you to supply data to a BASIC program and to print data on the terminal. These statements are described in the following sections:

- 3.1 Inputting Data
- 3.2 Printing Output - The PRINT Statement

3.1 INPUTTING DATA

BASIC has three methods of inputting data to a program:

1. The INPUT statements - require that you interact with the computer while the program is running.
2. The READ, DATA, RESTORE, and NODATA statements - require that you build a data block within the source program.
3. The file statements - require that you manipulate files outside the main program. See Chapter 8 for information on file input and output.

3.1.1 INPUT Statement

The INPUT statement allows you to enter and process data while the program is running.

The INPUT statement has the following format:

INPUT [prompt string { ; }] variable(s)

where:

prompt string is a quoted string constant followed by either a semicolon or a comma. For a description of the semicolon and the comma, see Section 3.2.1.

variable(s) is a list of real, integer, string, or subscripted variables or any combination of these separated by commas.

INPUT AND OUTPUT TO THE TERMINAL

The INPUT statement may be used as a means of assigning values to variables. When you run your program, BASIC stops at the line designated by the INPUT statement and prints a space, a question mark, and a space. BASIC then waits for you to type one value for each variable requested in the INPUT statement. When there is more than one variable requiring a value, separate each value with a comma. Enter a line terminator after you finish typing all the values.

The following example requires that you type three values after the question mark:

```
00010 INPUT A,B,C
00020 END
```

```
READY
RUNNH
```

```
? 5,6,7
```

The INPUT statement tells BASIC to accept data from the user terminal for the variables specified in the INPUT statement. BASIC accepts the values left to right. After you type all the necessary data, type a line terminator. The program continues using the values you supply. Therefore, in the previous example:

```
A=5
B=6
C=7
```

You must supply the same number of values as there are variables in the INPUT request. If you do not type enough data, BASIC lets you know by printing a message (shown in the example below) and another question mark when you press the RETURN key. After the second question mark, you can complete entering the correct amount of data. BASIC will then continue accepting data until the rest of the items specified in the INPUT request have been entered.

```
00010 INPUT A,B
00020 END
```

```
READY
RUNNH
```

```
? 5
```

```
? 5? Insufficient data at line 00010 of MAIN PROGRAM
? 6
```

On the other hand, if you supply more values than there are variables to be defined, BASIC ignores the excess and prints a warning message, as follows:

```
00010 INPUT A,B,C
00020 PRINT A,B,C
00030 END
```

```
READY
RUNNH
```

```
? 5,6,7,8
```

```
% 436 Too much data present - ignored at line 00010 of MAIN PROGRAM
5          6          7
```

The extra value entered (8) is ignored. (The PRINT statement in line 20 caused the accepted data to be printed.)

INPUT AND OUTPUT TO THE TERMINAL

The values you supply must be the same data type as the variables in the INPUT statement, that is, strings for string variables, integers for integer variables, and so forth. You can type strings with or without quotation marks in response to the question mark. If you include quotation marks, be sure to type both beginning and ending delimiters. If you forget the end quotation mark, BASIC generates an error message. If you exclude quotation marks, BASIC ignores any leading and trailing spaces or tabs that may be part of the string. If you want the null string to be valid input to a string variable, you must use the LINPUT statement described in the next section.

Including a string constant in an INPUT statement allows you to prompt the user for the input data. Separate the string constant from the variable list with a comma or semicolon. For example:

```
00020 INPUT "PLEASE TYPE 3 INTEGERS";B%,C%,D%
00030 AZ=B%+C%+D%
00040 PRINT AZ
00050 END

READY
RUNNH
PLEASE TYPE 3 INTEGERS? 25,50,75
150
READY
```

The INPUT statement with a string constant but without a variable list functions the same as a PRINT statement, for example:

```
00010 INPUT "BASIC--PLUS-2"

READY
RUNNH
BASIC--PLUS-2
READY
```

If the variable list specified for the INPUT statement is terminated with a comma, BASIC assumes that the data line entered in response to the INPUT request will contain more items than the number of variables specified. In this case, the excess data will not be ignored, and it will be used to satisfy any subsequent INPUT requests. For example:

```
00010 INPUT A,B,C,
00020 PRINT A,B,C
00030 INPUT D
00040 PRINT D

READY
RUNNH
? 1,2,3,4
1          2          3
4
READY
```

INPUT AND OUTPUT TO THE TERMINAL

3.1.2 INPUT LINE Statement and LINPUT Statement

The INPUT LINE statement and the LINPUT statement have essentially the same function as the INPUT statement. However, the INPUT LINE statement and the LINPUT statement are used exclusively for string data and read the entire line as the value for a variable. Use the following format:

```
INPUT LINE [ prompt string {; } ] string variable(s)
```

```
LINPUT [ prompt string {; } ] string variable(s)
```

where:

INPUT LINE requires a space between the keywords INPUT and LINE.

LINPUT is one word.

prompt string is a quoted prompt string constant followed by a semicolon or a comma. For a description of the semicolon and the comma, see Section 3.2.1.

string variable(s) are the only variables allowed in INPUT LINE and LINPUT statements.

The INPUT LINE statement accepts and stores all characters including quotation marks and commas, up to and including the first line terminator. LINPUT accepts all characters up to the line terminator, but does not include the line terminator. For example:

```
00010 LINPUT B$
00020 PRINT B$
00030 END

READY
RUNNH
?"NOW, LOOK HERE!", SAID JOHN
"NOW, LOOK HERE!", SAID JOHN
READY
```

If you try to type the string shown above in response to an INPUT statement (See Section 3.1.1), you will receive a warning message. The INPUT would take the comma after the word "HERE!", as the delimiter of the string.

INPUT AND OUTPUT TO THE TERMINAL

Data lines to the INPUT LINE and LINPUT statements can be continued by typing an ampersand (&) as the last character in the line. For example:

```
LISNH
 00010 LINPUT B&
 00020 PRINT
 00030 PRINT B&
```

```
READY
RUNNH
 ?THIS EXAMPLE SHOWS HOW &
 THE AMPERSAND IS USED TO &
 CONTINUE A DATA INPUT LINE
```

```
THIS EXAMPLE SHOWS HOW THE AMPERSAND IS USED TO CONTINUE A DATA
INPUT LINE
```

```
READY
```

3.1.3 READ, DATA, RESTORE, and NODATA Statements

Another way you can supply data to a program is to store data within the program for BASIC to read during execution. This means that you do not interact with BASIC while the program is running. Instead, you supply data to the program in advance. Two statements are involved in this process: READ and DATA. In addition to the READ and DATA statements, the RESTORE and NODATA statements enable you to re-read data in a data statement and transfer control to a specified line after all data is read.

3.1.3.1 The READ Statement - The READ statement has the following format:

```
READ variable(s)
```

where:

variable(s) is one or a list of variables consisting of numeric, string, subscripted variables, or a combination of these. All variables should be separated by commas. For example:

```
10 READ A, B%, C$, D(5), E
```

The READ statement directs BASIC to read from a list of values built into a data block by a DATA statement.

INPUT AND OUTPUT TO THE TERMINAL

3.1.3.2 The DATA Statement - The DATA statement has the following format:

DATA constant(s)

where:

constant(s) is one or more real, integer, or string constants (quoted or unquoted) listed in the same order of data type as the variable requested in the READ statement. All constants are separated by commas.

Programs run faster with READ and DATA than with the INPUT statements because you do not have to wait the extra time it takes for BASIC to stop and request data; the data is already within the program.

A READ statement causes the variables listed in it to be given the next available constants in sequential order from the collection of DATA statements. BASIC has a data pointer to keep track of the data being read by the READ statement. Each time the READ statement requests data, BASIC retrieves the next available constant indicated by the data pointer.

A READ statement is not valid without at least one DATA statement. You can, however, have more than one DATA statement. Without a corresponding READ statement, BASIC ignores the data.

A READ statement can be placed anywhere in a multi-statement line. A DATA statement, however, must be the last or only statement on a line. Each list of constants in a DATA statement is local to a program or subprogram.

BASIC generates error messages during program execution if:

1. You have a READ statement without a DATA statement.
2. You assign a string constant to a numeric variable.
3. You place more variables in the READ statement than you supply data to define them.

You can READ a numeric variable into a string variable. For example:

```
00010 READ A$
00020 PRINT A$
00030 DATA 8.25
00040 END
```

```
READY
RUNNH
8.25
```

The following example shows a READ and DATA sequence:

```
00010 READ A,B,C1,D2,E4,Y$,Z$,Z1$
00020 DATA 2.3,-4.2654,3,-6,12,'CAT',DOG,'MOUSE'
00030 PRINT A;B;C1;D2;E4;Y$,Z$,Z1$
```

INPUT AND OUTPUT TO THE TERMINAL

BASIC assigns values as follows:

```
A=2.3
B=-4.2654
C1=3
D2=-6
E4=12
Y$=CAT
Z$=DOG
Z1$=MOUSE
```

```
RUNNH
  2.3 -4.2654  3 -6  12 CAT  DOG          MOUSE
```

READY

3.1.3.3 THE RESTORE Statement - The format of the RESTORE statement is:

```
RESTORE [ line number ]
```

The RESTORE statement resets the data pointer to the beginning of the first DATA statement in the program. (RESET is an equivalent statement.) The values are read as though for the first time; therefore, the same variable names may be used the second time through the data. If the line number is specified, however, the data pointer is reset instead to the DATA statement on that line. Consider this example:

```
00010 READ B,C,D
00015 PRINT B,C,D
00020 RESTORE
00030 READ E,F,G
00035 PRINT E,F,G
00040 DATA 6,3,4,7,9,2
00050 END
```

The READ statement in line 10 reads the first three values in the DATA statement on line 40. The values for B,C, and D are, therefore as follows:

```
B=6
C=3
D=4
```

Then the RESTORE statement on line 20 resets the pointer to the beginning of line 40, so that the second READ statement on line 30 also reads the first three values, as follows:

```
E=6
F=3
G=4
```

If the RESTORE statement were not there, the READ statement on line 30 would read the last three values, as follows:

```
E=7
F=9
G=2
```

The RESTORE statement affects only the program or subprogram in which it is contained. Also, if you have no DATA statements in your program, the RESTORE statement has no effect.

INPUT AND OUTPUT TO THE TERMINAL

3.1.3.4 The NODATA Statement - The NODATA statement has the following format:

```
NODATA line number
```

where:

line number is any valid line number in the program.

The NODATA statement transfers program control to the specified line numbers if there is no more data in the program or subprogram.

3.2 PRINTING OUTPUT - THE PRINT STATEMENT

The PRINT statement has the following format:

```
PRINT [expression(s)]
```

where:

expression(s) can be one or more numeric or string elements separated by commas or semicolons.

The PRINT statement prints the specified element(s) on the terminal when you execute your program. In this way, you can see the results of your computations or add program prompts which clarify your requests for input. (The PRINT statement can be placed anywhere within a multi-statement line.)

You can include blank lines in your output to provide better formatting. Use the PRINT statement without arguments to output a blank line, for example:

```
00010 PRINT "THIS EXAMPLE LEAVES A BLANK LINE"  
00020 PRINT  
00030 PRINT "BETWEEN TWO LINES"  
00040 END
```

```
READY  
RUNNH  
THIS EXAMPLE LEAVES A BLANK LINE
```

```
BETWEEN TWO LINES
```

```
READY
```

NOTE

For greater control over the format of the output, use the PRINT USING statement described in Chapter 10.

INPUT AND OUTPUT TO THE TERMINAL

If an element in the list is an expression rather than a simple variable or constant, BASIC evaluates the expression before printing the value. The PRINT statement can therefore perform two operations in a single statement: calculate expressions and print results. For example:

```
00010 A = 45 \ B = 55
00020 PRINT A + B
00030 END
```

```
READY
RUNNH
100
```

After running this program, BASIC prints 100 on the terminal, not 45+55. If you put quotes around the variables, BASIC treats them as a string literal, as follows:

```
00010 A = 45 \ B = 55
00020 PRINT "A + B"
00030 END
```

```
READY
RUNNH
A + B
```

You can use the PRINT statement to provide instructions for using a BASIC program. To use the PRINT statement in this way, include literal strings in the PRINT statement, as shown in the following example:

```
00010 PRINT "WHAT ARE YOUR VALUES OF X,Y,Z"
00020 INPUT X,Y,Z
00030 LET R = SQR(X^2 + Y^2 + Z^2)
00040 PRINT "THE RADIUS OF THE VECTOR EQUALS";R
00050 END
```

When you run this program, the PRINT statement prompts you to type values for X, Y, and Z, as follows:

```
RUNNH
WHAT ARE YOUR VALUES OF X,Y,Z
? 24,40,50
THE RADIUS OF THE VECTOR EQUALS 68.38128

READY
```

Note that you enclose the strings in quotation marks so that BASIC prints them exactly as you type them in.

3.2.1 Formatting with the Comma and Semicolon

Each line printed by the PRINT statement consists of a number of zones. Each zone is 14 spaces wide. The default TOPS-20 line, which is 80 characters long, contains five full print zones on each line.

INPUT AND OUTPUT TO THE TERMINAL

NOTE

The number of print zones per line is directly proportional to the currently defined TOPS-20 terminal width. (The TOPS-20 TERMINAL command, used to change the terminal line width, is described in the TOPS-20 User's Guide.)

You can control the placement of your output within the print zones by using the separators: comma (,) and semicolon (;).

The comma signals BASIC to start printing at the beginning of the next print zone. If the last print zone on the line is filled, BASIC prints the output beginning at the first print zone on the next line. For example:

```
00005 INPUT A,B,C,D,E,F
00010 PRINT A,B,C,D,E,F
00020 END
```

```
READY
RUNNH
? 5,10,15,20,25,30
5           10           15           20           25
30
```

If you place more than one comma between list elements, you will skip one print zone for each extra comma. The following example prints the value of A in the first zone and the value of B in the third zone:

```
00010 A=5\B=10
00020 PRINT A,,B
00030 END
```

```
READY
RUNNH
5           10
```

To print an output line in a more compact format, ignoring print zones, use a semicolon as the separator between variables. A semicolon causes BASIC to print a number with a preceding and a following space; thus, two spaces occur between numbers. For example:

```
00010 PRINT 10;20
00020 END
```

```
READY
RUNNH
10  20
```

BASIC does not print spaces between strings separated by a semicolon:

```
00010 PRINT "10";"20"
00020 END
```

```
READY
RUNNH
1020
```

INPUT AND OUTPUT TO THE TERMINAL

Placing a comma or semicolon after the last item in a PRINT statement causes the terminal printer to remain at the same print position in anticipation of another PRINT or INPUT statement. If, however, the next item to be printed will not completely fit in the space remaining between the current print position and the right margin, BASIC will print the additional output at the beginning of the next line.

In the following example, BASIC prints the current values of X,Y and Z on one terminal line because a comma appears as the last item in line 20:

```
00010 INPUT X,Y,Z
00020 PRINT X,Y,
00030 PRINT Z
00040 END
```

```
READY
RUNNH
? 5,10,15
   5          10          15
```

The following example illustrates the three options you have for placing either a comma, a semicolon, or a line terminator after the last item of the PRINT statement:

```
00010 FOR I=1 TO 10
00020 PRINT I      !LINE TERMINATOR
00030 NEXT I\PRINT
00040 FOR J=1 TO 10
00050 PRINT J,    !A COMMA
00060 NEXT J\PRINT
00070 FOR K=1 TO 10
00080 PRINT K;    !A SEMICOLON
00090 NEXT K
00100 END
```

```
READY
RUNNH

1
2
3
4
5
6
7
8
9
10

1          2          3          4          5
6          7          8          9          10
1 2 3 4 5 6 7 8 9 10
```

Commas and semicolons also allow you to use or not use print zones for printing string output. For example:

```
00010 PRINT "FIRST ZONE",,"THIRD ZONE",,"FIFTH ZONE"
00020 END
```

```
READY
RUNNH
FIRST ZONE          THIRD ZONE          FIFTH ZONE
```

INPUT AND OUTPUT TO THE TERMINAL

Because of the extra comma between strings, BASIC skips every other printing zone before stopping to print each string.

3.2.2 Output Format for Numbers and Strings

BASIC prints numbers and strings according to a specific format. Strings are printed exactly as you type them in with no leading or trailing spaces. Leading and trailing spaces can, however, be added within the quotation marks by using the keyboard space bar. (Quotation marks are not printed unless delimited by another pair of quotation marks.)

```
00010 PRINT 'PRINTING "QUOTATION" MARKS'  
00020 END
```

```
READY  
RUNNH  
PRINTING "QUOTATION" MARKS
```

BASIC precedes negative numbers with a minus sign and positive numbers with a space. A space is always placed after the rightmost digit of a number.

```
00010 PRINT -1  
0020 PRINT 25;50  
0030 END
```

```
READY  
RUNNH  
-1  
25 50
```

The number of spaces occupied by the decimal representation of a number varies according to the magnitude and type of the number. BASIC prints the results of computations as decimal numbers (either integer or real) if they are within the range: $.0001 < n < 999999$, where n is the number BASIC prints. Otherwise, BASIC prints them in E notation.

BASIC prints decimal digits as illustrated below:

Value You Type	Value BASIC Prints
.000099	9.9E-05
.0001	0.0001
.01	0.01
999999	999999
1000000	1E+06

If more than six digits are generated during a computation, BASIC prints the result of that computation in E notation.

3.2.3 The TAB Function

Another method of positioning the terminal printer is to use the TAB function in conjunction with the PRINT statement.

This function has the following format:

```
PRINT TAB(n) { ; }  
          { , }
```

INPUT AND OUTPUT TO THE TERMINAL

where:

- n is an expression indicating the desired print position. BASIC evaluates the expression and truncates the result to an integer.
- ;
- ;
- ,

The TAB function does not cause characters to be printed: it only returns a string of spaces. The PRINT statement then prints those spaces returned by the TAB function. The number of spaces returned by TAB is n minus the current column number. If n is less than the current column number, the TAB function returns a null string.

With the TAB function, you move the terminal printer to the right to any desired column. The first column at the left margin is column 0. Therefore, n can be 0 to whatever the right margin is.

The TAB function can be used only to position the terminal printer from left to right, not right to left. If you specify a column that is to the left of the current column position, BASIC returns a null string.

You can use more than one TAB function in the same PRINT statement by placing them between elements. The following examples contain several TAB functions in conjunction with one PRINT statement:

```
00010 PRINT "NAME" ;TAB(15);"ADDRESS";TAB(30);"PHONE NO."  
00020 END
```

```
READY  
RUNNH
```

```
NAME           ADDRESS           PHONE NO.
```

Without tabs 15 and 30, BASIC would print:

```
NAMEADDRESSPHONE NO.
```

Here is an example of using the TAB function to position numbers:

```
00010 A=100\B=29\C=35  
00020 PRINT A; TAB(15);B; TAB(30);C  
00030 END
```

```
READY  
RUNNH
```

```
100           29           35
```

```
(Column 0)   (Column 15)   (Column 30)
```

Notice that semicolons act as separators in the preceding example; but the TAB function determines the position at which the numbers are printed.

INPUT AND OUTPUT TO THE TERMINAL

Compare the following examples. The first one uses commas as separators; the second one uses semicolons.

```
00010 A=100
00020 B=200
00030 C=300
00040 PRINT A,TAB(30),B,TAB(40),C
00050 PRINT A;TAB(30);B;TAB(40);C
00060 END
```

```
READY
RUNNH
```

```
100                                200                300
100                                200                300
```

You can also place a TAB function outside a PRINT statement. However, the value of the function depends on whether or not you did place it in a PRINT statement:

1. If TAB has not been placed in a PRINT statement, then

TAB(n) is the same as SPACE\$(n).

The SPACE\$ function allows you to add spaces in a string; see Section 6.2.7.

2. If a PRINT statement is executed with TAB, TAB(n) is a string of spaces whose length is equal to the number of spaces between the last print position and n.

CHAPTER 4

PROGRAM CONTROL

The order in which BASIC executes statements in a program is termed "program control." If you do not include any statements in your program that alter the order in which BASIC executes the statements, BASIC executes the statements from top to bottom.

There are, however, a variety of statements that allow you to alter the default top-to-bottom program control. For example, the simplest statement for altering the program control is the unconditional GOTO. This statement directs BASIC to transfer execution to the line number specified in the GOTO statement.

BASIC also has a variety of more complicated program control statements that allow you to:

- transfer control based on the value of an expression
- create sequences of statements that are repeated a specified number of times
- halt and end program execution
- create subroutines
- transfer control to a sequence of statements if an error occurs in the program

In addition to the program-control statements themselves, BASIC has a set of statement modifiers, which qualify or restrict the execution of the statements they modify. The BASIC program-control statements and modifiers are described in the following sections:

- 4.1 Unconditional Transfer - The GOTO Statement
- 4.2 Multiple Branching - The ON-GOTO Statement
- 4.3 Conditional Transfer - The IF-THEN-ELSE Statement
- 4.4 Loop Execution
- 4.5 Time-Limit Statements
- 4.6 Stopping Program Execution - The STOP and END Statements
- 4.7 Using Subroutines
- 4.8 Error Handling
- 4.9 Statement Modifiers

4.1 UNCONDITIONAL TRANSFER - THE GOTO STATEMENT

The GOTO statement causes control to be transferred to the statement it specifies.

PROGRAM CONTROL

The format of the GOTO statement is:

```
GOTO line number
```

where:

line number is the line to be executed next.

This line number can be smaller or larger than the line number of the GOTO statement. Therefore, you have the option to skip any number of lines in either direction. BASIC executes the statement at the line number specified by GOTO and continues the program from that point. The line number must be accessible to this program or subprogram (for example, a GOTO cannot direct control from a main program to a subprogram). Consider the example:

```
00030 GOTO 110
```

When BASIC executes line 30, it transfers control to line 110. BASIC interprets the statement exactly as it is written: go to line 110.

Consider the following sample program with a GOTO statement:

```
00010 A =2
00020 GOTO 40
00030 A = SQR(A+14)
00040 PRINT A,A*A
00050 END
```

```
READY
RUNNH
  2          4
```

In this program, control passes in the following sequence:

1. BASIC starts at line 10 and assigns the value 2 to the variable A.
2. Line 20 sends BASIC to line 40.
3. BASIC executes the PRINT statement.
4. BASIC ends the program at line 50.

Notice that line 30 is never executed.

Make sure that the GOTO statement is either the only statement on the line or the last statement in a multi-statement line. If you place a GOTO in the middle of a multi-statement line, BASIC does not execute the rest of the statements on the line. For example:

```
00025 A = ATN(B2)\GOTO 50\PRINT A
```

BASIC never executes the PRINT statement on line 25 because the GOTO statement transfers control to line 50.

PROGRAM CONTROL

If you specify a line number in a GOTO statement and that line contains a non-executable statement, such as a REM statement, BASIC transfers control to the next executable statement after the one specified. For example:

```
00010 REM SQUARE ROOT FUNCTION
00020 INPUT A
00030 IF A=0 GOTO 60
00040 PRINT "THE SQUARE ROOT OF";A;"IS";SQR(A)
00050 GOTO 10
00060 END
```

At line 40, BASIC transfers control to line 10. Because line 10 is a non-executable statement, BASIC ignores it and control passes to line 20.

4.2 MULTIPLE BRANCHING - THE ON-GOTO STATEMENT

The ON-GOTO statement is another means of transferring control within a program. Like the GOTO statement, ON-GOTO allows you to transfer control to another line of the program; however, ON-GOTO also allows you to specify several line numbers as alternatives, depending on the result of the expression.

The ON-GOTO statement has the following format:

```
ON numeric expression {GOTO} line number(s)
                      {THEN}
```

where:

numeric expression is any valid BASIC numeric expression.

GOTO
THEN are interchangeable keywords.

line number(s) is one or more line numbers. Line numbers must be separated by commas.

The ON-GOTO statement is also known as a computed GOTO because of its dependency on the value of the numeric expression. When BASIC executes the ON-GOTO statement, it first evaluates the numeric expression. The value is then truncated to integer (if necessary). If the value of the expression is equal to 1, BASIC passes control to the first line number in the list, if the value of the expression is equal to 2, BASIC passes control to the second line number in the list and so on. This process continues until the list is exhausted or there are no more values. If the value is less than 1 or greater than the number of line numbers in the list, BASIC prints an error message.

In the following example:

```
200 ON A GOTO 50,20,100,300
```

if A=1, GOTO line 50 (first line number in the list),
if A=2, GOTO line 20 (second line number in the list),
if A=3, GOTO line 100 (third line number in the list),
if A=4, GOTO line 300 (fourth line number in the list),

PROGRAM CONTROL

If A is equal to 3 (that is, if the relation is true), control passes to line 200. If A is not equal to 3, control does not pass to line 200. Instead, control passes to the next sequential instruction after line 20.

Here is a complete program illustrating the IF-THEN-ELSE statement:

```
00005 REM PROGRAM TO COMPARE TWO NUMBERS
00010 PRINT "INPUT VALUE OF A";\INPUT A
00015 PRINT "INPUT VALUE OF B";\INPUT B
00020 IF A=0 AND B=0 THEN 80
00030 IF A=B THEN PRINT "A EQUALS B"\GOTO 75
00040 IF A<B THEN 60
00050 PRINT "B IS LESS THAN A"\GOTO 75
00060 PRINT "A IS LESS THAN B"
00075 IF A*B>=B*(B+1) THEN LET D4=D4+1\GOTO 10
00080 END
```

```
READY
RUNNH
INPUT VALUE OF A ? 25
INPUT VALUE OF B ? 43
A IS LESS THEN B
```

If you include the ELSE clause, as in the third format, BASIC executes the ELSE clause if the THEN or GOTO clause preceding it is not executed. This means that if the conditional expression is false, BASIC executes the ELSE clause.

The following example illustrates the ELSE clause:

```
00010 IF A>=90 THEN G$="A" ELSE G$=F
```

You can also use string expressions, as in this example:

```
00300 IF C$="OUTPUT" GOTO 10
```

If the value of the string variable C\$ is equal to the string "OUTPUT", control passes to line 10.

Be careful when placing the IF-THEN-ELSE statement in a multi-statement line. The following rules govern the transfer of control:

1. Execution of the physically last THEN or ELSE clause determines the execution of the rest of the statements on the line. If the THEN or ELSE clause is executed, the next statement or statements following it are executed. If the THEN or ELSE clause is not executed, the statements following it are not executed, and control passes to the next line number. For example:

```
00005 INPUT A
00010 IF A=1 THEN PRINT A;\PRINT "TRUE CASE"\GOTO 20
00015 PRINT "NOT=1"
00020 END
```

If A is equal to 1, BASIC prints:

```
RUNNH
? 1
1 TRUE CASE
```

PROGRAM CONTROL

Because the relation is true, BASIC executes the rest of line 10, which includes a branch to line 20.

If A is not equal to 1, BASIC prints:

```
RUNNH
? 5

NOT=1
```

Because the relation is false, BASIC skips the rest of the statements on line 10 following the keyword THEN and proceeds to execute line 15.

2. All other ELSE and THEN clauses in a multi-statement line must be true for the remainder of that line to be executed. If any THEN or ELSE clause is not true, then control passes to the next line number.

```
00010 INPUT A,B,C
00020 IF A>B THEN IF B<C THEN PRINT "B<C"\GOTO 30
00025 PRINT "A<B"
00030 END
```

The statement GOTO 30 is executed only if A is greater than B and B is less than C. If A is either less than or equal to B or B is greater than or equal to C, then line 25 is executed:

```
RUNNH
? 10,15,20
A<B
```

3. If the statement following the THEN or ELSE clause is a FOR statement, you must include the corresponding NEXT statement in the same THEN or ELSE clause. In the following example, the FOR statement creates a program loop (See Section 4.4) on the second line after the ELSE statement; the corresponding NEXT statement occurs at the end of the second line:

```
00010 IF A=3 THEN FOR X=1 TO 10\B(X)=B(X)*A\NEXT X&
ELSE FOR X=1 TO 10\B(X)=B(X)+C\NEXT X
```

4. An ELSE clause is paired with the last unmatched IF-THEN clause. In the following example, the ELSE clause is paired with the second IF-THEN clause and is executed only if A is greater than 6:

```
00010 IF A>5 THEN IF A=6 THEN PRINT '=6' ELSE &
PRINT '<6'
```

A null ELSE clause may be used if the first IF-THEN clause requires an ELSE clause and the second IF-THEN clause has no ELSE clause paired with it. For example:

```
00010 IF A>5 THEN IF A=6 THEN PRINT '=6'&
ELSE&
ELSE PRINT '<6'
```

PROGRAM CONTROL

4.4 LOOP EXECUTION

A loop is the repeated execution of a set of statements. Placing a loop in a program saves you from duplicating and enlarging a program unnecessarily. The following section describes how to build a loop with the FOR and NEXT statements.

4.4.1 The FOR and NEXT Statements

Without some sort of terminating condition, a program can run through a loop indefinitely. The FOR and NEXT statements allow you to design a loop wherein BASIC tests for a condition each time it runs through the loop. You decide how many times you want the loop to run, and you set the terminating condition accordingly.

The FOR statement has the following format:

$$\text{FOR variable} = \text{num expr 1 TO num expr 2} \left[\begin{array}{l} \{\text{STEP}\} \text{num expr 3} \\ \{\text{BY}\} \end{array} \right]$$

where:

variable	is a numeric variable known as the loop index. It represents the counter for the loop.
num expr 1	is the first numeric expression - the initial value of the index.
num expr 2	is the second numeric expression - the maximum value of the index.
num expr 3	is the incremental value of the loop index known as the step size. This value is optional; if specified, it can be positive or negative. If not specified, the default is +1.

BASIC evaluates all numeric expressions in the FOR statement before assigning a value to the loop variable. For example:

```
00010 T = 3
00020 FOR M = 10*T TO 30*T STEP T
00030 NEXT M
```

M, the loop index, is given the initial value of 30, and BASIC tests to determine if M is less than or equal to the terminating value of 90. The loop is executed because M is less than 90. When the NEXT statement is encountered, the value of M is incremented by 3. BASIC tests again to see if M is greater than or equal to 90. When the value of M is greater than 90, control passes to the statement following the NEXT statement.

The NEXT statement has the following format:

```
NEXT numeric variable(s)
```

PROGRAM CONTROL

where:

numeric variable(s) must be the same loop index(es) named in the corresponding FOR statement. With multiple variables, the last loop variable must be specified first. Refer to Section 4.4.2 for nested loops.

The FOR and NEXT statements must be used together. If you use one without the other, an error condition results. The FOR statement defines the beginning of the loop; the NEXT statement defines the end. You are building a counter into your program to determine the number of times the loop is to execute.

Place the statements you want repeated between the FOR and NEXT statements. Consider the following example:

```
00010 FOR I = 1 TO 10
00020 PRINT I
00030 NEXT I
00040 PRINT I
00050 END
```

In this program, the initial value of the loop index is 1. The terminating value is 10, and the STEP size is +1 (the default).

Every time BASIC goes to line 30, it increments the loop index by 1 (the STEP size) until the terminating condition is met. Therefore, this program prints the values of I ten times. When the loop is completed, execution proceeds to line 40. The following is the resulting output:

```
RUNNH
 1
 2
 3
 4
 5
 6
 7
 8
 9
10
10
```

Notice that when control passes from the loop, the last value of the loop index is retained. Therefore, I equals 10 on line 40.

You can modify the loop index within the loop:

```
00010 FOR I = 2 TO 44 STEP 2
00020 LET I = 44
00030 NEXT I
00040 END
```

The loop in this program only executes line 20 once because at line 20 the value of I is changed to 44 and the terminating condition is reached.

If the initial value of the loop index is greater than the terminal value and the step size is positive, the loop is never executed.

```
00010 FOR I = 20 TO 2 STEP 2
```

PROGRAM CONTROL

This loop cannot execute because you cannot decrease 20 to 2 with increments of +2. You can, however, accomplish this with decrements of -2.

```
00010 FOR I = 20 TO 2 STEP -2
```

The STEP size can also be a number with a fractional part:

```
00010 FOR K = 1.5 TO 7.7 STEP 1.32
```

NOTE

You should not transfer control into a loop that has not been initialized by a FOR statement. The results would be unpredictable. The following is not recommended in a BASIC program:

```
00010 REM THIS IS A POOR PROGRAM
00020 GOTO 40
00030 FOR I = 1 TO 20
00040 PRINT I
00050 NEXT I
00060 END
```

Line 20 transfers control to line 40, bypassing line 30. This is invalid in BASIC.

You can place FOR and NEXT statements anywhere in a multi-statement line. For example:

```
00010 FOR I = 1 TO 10 STEP 5\PRINT "I = ";I\NEXT I
00020 END
RUNNH
I = 1
I = 6
```

The calculation of the index values (initial, final, and step size) is subject to the precision limitations of the computer. These index values are represented in the computer by binary numbers. When the values are integer, they can be represented exactly in binary; however, it is not always possible to represent decimal values exactly in binary when they contain a fractional part. Consider the following example:

```
00020 FOR X = 0 TO 10 STEP 0.1
00030 A = 75\B = 473\C = A/B
00040 PRINT "THE ANSWER IS ";C
00050 NEXT X
00060 END
```

The loop established in line 20 executes 100 times instead of 101 because the internal value of 0.1 is not exactly 0.1. After the 100th execution of the loop, X is not exactly equal to 10. It is slightly larger than 10, so the loop stops. Whenever possible, it is advisable to use indexes that have integer values which ensure that the loop is executed the correct number of times.

PROGRAM CONTROL

Note that changing the termination value of a loop within the loop has no effect. For example:

```
00010 K = 10
00020 FOR I = 1 TO K
00030 K = 5
00040 PRINT I;
00050 NEXT I
```

```
READY
RUNNH
 1 2 3 4 5 6 7 8 9 10
```

4.4.2 Nested Loops

A loop can contain one or more loops provided that each inner loop is completely contained within the outer loop. Using one loop within another is called nesting. Each loop within a nest must contain its own FOR and NEXT statements, and the inner loop must terminate before the outer loop, that is, the one that starts first must be completed last. Loops cannot overlap.

The following example shows valid and invalid forms of nested loops:

Valid	Valid	Invalid
<pre>10 FOR A = 1 TO 10 20 FOR B = 2 TO 20 30 NEXT B 40 NEXT A</pre>	<pre>10 FOR A = 1 TO 10 20 FOR B = 2 TO 20 30 NEXT B 40 FOR C = 3 TO 30 50 FOR D = 4 TO 40 60 FOR E = 5 TO 50 70 NEXT E 80 NEXT D 90 NEXT C 100 NEXT A</pre>	<pre>10 FOR M = 1 TO 10 20 FOR N = 2 TO 20 30 NEXT M 40 NEXT N</pre>

The following is a program with a nested loop:

```
00010 PRINT "I","J"
00015 PRINT
00020 FOR I=1 TO 2
00030 FOR J=1 TO 3
00040 PRINT I,J
00050 NEXT J
00060 NEXT I
00070 END
```

```
READY
RUNNH
```

```
 I      J
 1      1
 1      2
 1      3
 2      1
 2      2
 2      3
```

PROGRAM CONTROL

FOR and NEXT statements are commonly used to initialize arrays, as illustrated in this example:

```
00005 DIM X(5,10)
00010 FOR A=1 TO 5
00020 FOR B=2 TO 10 STEP 2
00030 X(A,B)=A+B
00040 NEXT B,A
00055 PRINT X(5,10)
00060 END
```

```
READY
RUNNH
15
```

4.4.3 The Conditional FOR Statement

Another method of creating loops in a program is to use the conditional FOR statement. The conditional FOR statement has the following format:

```
FOR variable=num expr 1 [ {STEP} num expr 2 ] {WHILE} conditional expr
                        [ {BY } ] [ {UNTIL} ]
```

where:

variable	is a numeric variable known as the loop index. It represents the counter for the loop.
num expr1	is the first numeric expression which determines the initial value of the index.
num expr2	is the incremental value of the loop index known as the step size. This value is optional; if specified, it can be positive or negative. If not specified, the default is +1.
conditional expr	is the condition tested. The expression can be a relational or logical expression.

This form of loop is similar to the normal FOR statement. The difference lies in the termination test for the loop. Each time the loop is about to begin, BASIC evaluates the conditional expression and tests it for its truth value. The loop terminates if the conditional expression is true and the clause is an UNTIL clause, or if the conditional expression is false and the clause is a WHILE clause. The NEXT statement is required with this form of the FOR statement.

PROGRAM CONTROL

When BASIC exits from the conditional loop, the value of the loop index is the value that terminates the loop. In contrast, when the normal FOR-NEXT loop terminates, the value of the loop index is the last value used in the FOR statement, not the terminating value. Consider the following example:

```
00010 FOR I=1 TO 10           !NORMAL FOR LOOP
00015 PRINT I#
00020 NEXT I
00025 PRINT "I=";I
00030 FOR I=1 UNTIL I>10     !CONDITIONAL FOR LOOP
00035 PRINT I#
00040 NEXT I
00045 PRINT "I=";I
00050 END
```

```
READY
RUNNH
 1  2  3  4  5  6  7  8  9 10 I= 10
 1  2  3  4  5  6  7  8  9 10 I= 11
```

Both loops print the numbers 1 through 10. Notice, however, the difference in the values of I. When the first loop terminates, the loop variable is set to the last value BASIC used (10). In the second loop beginning on line 30, the loop variable is set to the value which caused the loop to terminate (11).

The following example illustrates the WHILE clause:

```
00010 FOR I=1 WHILE I<10
00015 PRINT I#
00020 NEXT I
00025 PRINT "I=";I
00030 END
```

```
READY
RUNNH
 1  2  3  4  5  6  7  8  9 I=10
```

4.4.4 The FOR Statement with an Additional Termination Test

The FOR statement with an additional termination test has the following format:

FOR variable=num expr 1 TO num expr 2 $\left[\begin{array}{l} \{STEP\} \text{ num expr 3} \\ \{BY\} \end{array} \right] \left\{ \begin{array}{l} \{WHILE\} \text{ conditional expr} \\ \{UNTIL\} \end{array} \right\}$

where:

variable	is a numeric variable (loop index).
num expr 1	is the initial value of the index variable.
num expr 2	is the terminating value of the index variable.
STEP num expr 3 BY	is the increment value of the index. This is optional; the default is +1.
WHILE conditional expr UNTIL	is a logical or relational expression. This is the additional termination test.

PROGRAM CONTROL

This type of loop is equivalent to the normal FOR-NEXT statements except for the addition of the conditional test. Each time BASIC encounters the NEXT statement, the loop index is incremented. After incrementing the loop index, BASIC checks it to see if the TO expression has been exceeded. If the TO expression has not been exceeded, BASIC checks the conditional expression. The termination test on the conditional expression is the same as in the Conditional FOR statement, Section 4.4.3.

Consider the following example:

```
00010 Y=0
00020 FOR I=1 TO 10
00030 PRINT I;                !NORMAL FOR LOOP
00040 Y=I
00050 NEXT I
00060 PRINT "I=";I
00070 FOR I=1 TO 10 UNTIL Y> 10 !FOR WITH ADDITIONAL TEST
00080 PRINT I;
00090 Y=I*2
00100 NEXT I
00110 PRINT "I=";I
00120 END
```

```
READY
RUNNH
```

```
1 2 3 4 5 6 7 8 9 10 I= 10
1 2 3 4 5 6 I= 7
```

In the above example, the normal FOR loop prints the numbers 1 through 10. Line 110 of the conditional FOR statement, however, causes the loop to print the numbers 1 through 6. Before the seventh iteration of the loop, the value of Y is greater than 10, making the conditional expression in line 70 true.

4.4.5 The WHILE and UNTIL Statements

The WHILE and UNTIL statements have the following format:

```
WHILE conditional expression
```

```
UNTIL conditional expression
```

where:

```
conditional expression      is any numeric, logical, or
                             relational expression.
```

Like the FOR statement, the WHILE and UNTIL statements require a corresponding NEXT statement. However, the NEXT statement, in this case, does not contain a variable; rather, by itself it acts as the loop terminator.

PROGRAM CONTROL

The conditional expression is evaluated before each loop iteration. If the expression is true, BASIC executes the statements within the loop. If the expression is false, BASIC executes the statement following the NEXT statement. For example:

```
00010 WHILE A%<10%
00020 LET A%=A%+I%
00030 I%=1 TO 5
00040 NEXT
00050 PRINT A%
00060 END
```

As long as A% is less than 10%, BASIC will execute the statements within the loop.

With the UNTIL statement, the loop executes until the expression is true. For example:

```
00010 I=12
00020 UNTIL I=0
00030 PRINT I;
00040 I=I-1
00050 NEXT
```

4.5 TIME-LIMIT STATEMENTS

BASIC provides statements to suspend program execution for a specified amount of time. These statements are SLEEP and WAIT.

4.5.1 The SLEEP Statement

The SLEEP statement has the following format:

```
SLEEP numeric expression
```

where:

numeric expression is the number of seconds to delay further execution of the program or subprogram.

For example:

```
00010 SLEEP 120*10
```

At the end of 1200 seconds (20 minutes) BASIC continues execution.

4.5.2 The WAIT Statement

The WAIT statement has the following format:

```
WAIT numeric expression
```

PROGRAM CONTROL

where:

numeric expression specifies the maximum number of seconds BASIC will wait for input from the terminal before an error condition is signaled.

For example:

```
00010 WAIT 60
```

When this line is executed, BASIC waits for 60 seconds for the current terminal input request to be satisfied before generating an error. The input request is satisfied only when a line terminator is entered. If data is partially entered (but no line terminator is typed) and the time specified in the WAIT statement elapses, all data is discarded.

The WAIT statement is used in conjunction with the INPUT statement so that you can set time limits for responses to your program.

A WAIT statement with a value of 0 or no value, indicates that no WAIT error condition exists no matter how long it takes for a response. Thus, WAIT 0 turns off a previous WAIT.

You must place the WAIT statement before the respective INPUT statement. For example:

```
00010 WAIT 15
00020 INPUT A,B,C
00030 D=A*B/C
00040 PRINT D
```

BASIC waits 15 seconds for a response to the INPUT statement. If no response is typed, BASIC prints an error message.

4.6 STOPPING PROGRAM EXECUTION - THE STOP AND END STATEMENTS

There are three methods of halting program execution:

1. Executing all the statements
2. Using the STOP statement
3. Using the END statement

The first method is shown in the following example. The program executes completely and then BASIC closes all files:

```
00010 FOR I=1 TO 10
00020 PRINT I
00030 NEXT I
```

The STOP statement has the following format:

```
STOP
```

This statement causes program execution to halt, at which point BASIC prints a message:

```
STOP at line n in program segment
```

PROGRAM CONTROL

where:

n is the line number of the STOP statement.

program segment is either MAIN PROGRAM or the name of the function or subprogram in which the program stopped.

You can place several STOP statements at various points in a single program. This is a useful debugging tool in determining program flow in large programs.

The STOP statement halts execution but it does not close files. When you use STOP and you want BASIC to close files at program termination, you must use the END statement or explicitly close the file with the CLOSE statement (See Section 9.5.2).

The END statement has the following format:

```
END
```

The END statement is optional unless there are subprograms in the program. See Section 5.1 for information on subprograms. If you include an END, it must have the largest line number in the main program. Any reference to an END statement via a GOTO or IF-THEN-ELSE statement terminates program execution and closes all files.

An END statement does not cause BASIC to print a message on the terminal. If a message is desired, use the STOP statement.

If you do not include a STOP or an END statement in a program, the execution of the last statement of the program terminates program execution and closes all files. That is, the STOP statement is optional, and END is necessary only when you reference the end of the program with a transfer statement or place subprograms after the main program.

The following examples show all three options of ending a program. In the first example, BASIC executes all statements and closes all files:

```
00010 READ A,B,C
00020 PRINT "A=";A
00030 PRINT "B=";B
00040 PRINT "C=";C
00050 DATA 100,300,450
```

```
READY
RUNNH
A= 100
B= 300
C= 450
```

PROGRAM CONTROL

In the second example, BASIC executes all statements and stops executions at line 60; the files are not closed:

```
00010 READ A,B,C
00020 PRINT "A=";A
00030 PRINT "B=";B
00040 PRINT "C=";C
00050 DATA 100,300,450
00060 STOP
```

```
READY
RUNNH
A= 100
B= 300
C= 450
```

STOP at line 00060 of MAIN PROGRAM

In the third example, BASIC executes all statements and closes all files:

```
00010 READ A,B,C
00020 PRINT "A=";A
00030 PRINT "B=";B
00040 PRINT "C=";C
00050 DATA 100,300,450
00060 END
```

```
READY
RUNNH
A= 100
B= 300
C= 450
```

As you can see, the first, second, and third examples have the same output.

4.7 USING SUBROUTINES

Subroutines are like functions in that you reference them in another part of the program. However, they are unlike functions in that you do not name a subroutine or specify an argument. Instead, you include the GOSUB statement, which transfers control of the program to a subroutine, and the RETURN statement, which returns control from that subroutine back to normal program execution.

In BASIC, you can enter more than one subroutine in the same program. Subroutines are easier to locate (for debugging purposes) if you place them near the end of the program, before any DATA statements and before the END statement (if present). You can also assign distinctive line numbers to subroutines. For example, if the main program has line numbers ranging from 10 to 190, begin the subroutines with line numbers 200, 300, 400 and so on.

The first line of a subroutine can be any valid BASIC statement, including a REM statement. You do not have to transfer to the first line of the subroutine. Instead, you can include several entry points and RETURNS into and out of the same subroutine. Similarly, you can nest subroutine calls (one subroutine within another) up to a system-defined limit.

PROGRAM CONTROL

The following sections describe the building of subroutines with the GOSUB and RETURN statements.

4.7.1 The GOSUB and RETURN Statements

When BASIC begins executing a program, it continues in sequence until it encounters a transfer statement, such as a GOSUB statement. The GOSUB statement has the following format:

```
GOSUB line number
```

where:

```
line number      is either the first line of the subroutine or
                  a line used as an entry point into the
                  subroutine.
```

BASIC transfers control to the specified line. For example:

```
00010 GOSUB 200
```

BASIC stops executing sequentially at line 10 and transfers control to line 200. BASIC executes the subroutine until it encounters a RETURN statement, which causes BASIC to transfer control back to the statement immediately following the calling GOSUB statement.

A subroutine can be exited only through a RETURN statement. The RETURN statement has the following format:

```
RETURN
```

Before it transfers control to a subroutine, BASIC records the next sequential statement following the GOSUB statement. The RETURN statement signals BASIC to return to the statement previously recorded. In this way, no matter how many subroutines there are or how many times they are called, BASIC always knows where to transfer control. For example:

```
00010 INPUT A,B,C
00020 GOSUB 40
00030 PRINT D
00035 GOTO 70
00040 REM THIS IS A SUBROUTINE
00050 D = A * B - C
00060 RETURN
00070 END
```

Line 20 sends BASIC to line 40; then line 60 returns execution to line 30. The resulting output is:

```
RUNNH
? 5,10,15
35
```

PROGRAM CONTROL

The following is an example of several calls to the same subroutine:

```
00010 DIM B(100)
00020 GOSUB 60
00030 GOSUB 60
00040 GOSUB 60
00050 GOTO 110
00060 LET A = 0
00070 FOR I = 1 TO 5
00080 LET A = A+B(I)
00090 NEXT I
00100 RETURN
00110 END
```

The same subroutine on line 60 is called three times. Notice that only one RETURN statement is necessary.

4.7.2 The ON-GOSUB Statement

The ON-GOSUB statement is used to conditionally transfer control to one of several subroutines or to one of several entry points in one or more subroutines. The ON-GOSUB statement has the following format:

ON numeric expression GOSUB line number(s)

where:

numeric expression is any valid BASIC numeric expression.

line number(s) one or a list of line numbers contained
in the program, separated by commas.

The ON-GOSUB statement works like the ON-GOTO statement (See Section 4.2). When BASIC executes the ON-GOSUB statement, it first evaluates the numeric expression. The value is then truncated to integer, if necessary.

If the value of the expression is

- 1, control passes to the first line number specified.
- 2, control passes to the second line number specified.
- 3, control passes to the third line number specified, and so on.

If the expression is less than 1 or greater than the number of line numbers in the list, BASIC prints an error message to that effect. The following is an example of an ON GOSUB statement:

```
00020 ON A+B GOSUB 200,300,120
```

When:

A+B=1, go to the subroutine on line 200, (first line number in
the list)

PROGRAM CONTROL

A+B=2, go to the subroutine on line 300, (second line number
in the list)

A+B=3, go to the subroutine on line 120, (third line number
in the list)

(A+B)<1, print error message,

(A+B)>4, print error message.

The line number to which BASIC branches can be either the first line of a subroutine or a line number used as an entry point into a subroutine.

4.8 ERROR HANDLING

The BASIC system error handler detects errors that occur during program execution. When an error occurs in an executing program, the system error handler, depending on the type of error detected, either prints a warning message (preceded by a %) and continues execution, or prints a fatal error message (preceded by a ?) and terminates execution.

There are two categories of execution errors: nontrappable and trappable. The nontrappable errors always cause an immediate termination of program execution. The trappable errors include all non-fatal errors (warnings) and certain fatal errors, which can be detected by an error routine that you have built into the program. (The trappable and nontrappable error messages are listed in Sections A.3.1 and A.3.2, respectively.)

By using a combination of BASIC statements, you can set up error-handling routines within your program that take precedence over the system error handler when a trappable error occurs. These routines enable the program to evaluate and optionally proceed with program execution. BASIC supplies the following elements for creating error routines:

1. the ONERROR GOTO statement
2. the ONERROR GO[TO] BACK statement
3. the RESUME statement
4. the BASIC error variables: ERR, ERL, and ERN\$
5. the LINO function

4.8.1 The ONERROR GOTO Statement

The ONERROR GOTO statement indicates to BASIC that an error-handling routine exists in your program at a specified line number. The ONERROR GOTO statement has the following format:

ONERROR GOTO line number

PROGRAM CONTROL

where:

line number specifies the number of the line that begins the routine. If you specify 0, then the system error handler responds to any errors.

If a trappable error occurs before an ONERROR statement is executed, the system error handler responds to the error. If a trappable error occurs after an ONERROR statement (with a nonzero line number) is executed, BASIC transfers control to the line specified in the ONERROR statement.

If an error occurs within your error-handling routine, then the system error handler responds to both errors by printing the following message and terminating program execution:

```
? 437 error er2 occured while processing error er1 at line nn of name
```

where:

er2 is the number of the error that occurred in your error routine. The trappable error messages and their numbers are listed in Section A.3.1.

er1 is the number of the error that originally caused BASIC to execute your error routine.

nn is the line number in your error routine where the error (er2) occurred.

name is the name of the program segment in which er2 occurred.

A typical situation in which you receive this error message is if your program stops because of an error and you execute an immediate-mode statement that causes a second error.

The ONERROR GOTO statement is local to the program in which it is contained. If an error occurs in a function that does not have an error routine, BASIC automatically checks the calling program for an error routine. If BASIC does not find an error routine in the calling program, the system error handler responds to the error.

If an error occurs within a subprogram, BASIC checks only that subprogram for an error routine before handling the error itself. You can, however, direct BASIC to check the calling program when an error occurs in a subprogram. To do this, include the following statement in the subprogram:

```
ONERROR GO[TO] BACK
```

PROGRAM CONTROL

The ONERROR GO statement with the BACK clause passes error control from a subprogram to the calling program. For example:

```
READY
LISNH
00100 ONERROR GOTO 150
00110 PRINT "FIND THE ROOT OF";
00120 INPUT N%
00130 CALL SQUARE(N%)
00140 GOTO 170
00150 PRINT "INVALID ENTRY ---- TRY AGAIN"
00160 RESUME 110
00170 END
00180 SUB SQUARE(N%)
00190 ONERROR GO BACK
00198 M%=SQR(N%)
00200 PRINT "THE SQUARE ROOT IS";M%
00210 SUBEND
```

```
READY
RUNNH
FIND THE ROOT OF ? -9
INVALID ENTRY ---- TRY AGAIN
FIND THE ROOT OF ? A STRING
INVALID ENTRY ---- TRY AGAIN
FIND THE ROOT OF ? 81
THE SQUARE ROOT IS 9
```

READY

In this example, the ONERROR GO BACK statement sends BASIC back to the error routine in the main program when an error occurs in subprogram ROOT. The RESUME statement (line 160) is described in the next section.

4.8.2 The RESUME Statement

The RESUME statement must be the last statement of your error-handling routine. When executed, the RESUME statement terminates the error-handling routine, and either returns control to a specified line number or back to the beginning of the line that caused the error. The RESUME statement has the following format:

```
RESUME line number
```

where:

line number	is an optional line number specifying the line to which control passes. If no line number is specified, control passes to the beginning of the line where the error occurred.
-------------	---

PROGRAM CONTROL

If the line to which control passes is a multi-statement line, BASIC begins execution at the beginning of the line. For example:

```
READY
OLD MULT,B20
```

```
READY
LISNH
00010 ONERROR GOTO 40
00020 INPUT A \ B=A*A \ PRINT A;"SQUARED IS";B
00030 GOTO 60
00040 PRINT "INVALID ENTRY ---- TRY AGAIN"
00050 RESUME
00060 END
```

```
READY
RUNNH
? STRING
INVALID ENTRY ---- TRY AGAIN
? 8
8 SQUARED IS 64
```

```
READY
```

In this example, the error occurs at line 20. The RESUME statement (line 50) returns execution to the first statement on line 20.

4.8.3 The BASIC Error Variables

When BASIC detects an error in a program, it supplies three variables with data regarding the error. These variables are:

ERR contains the number of the trappable error message. (These messages are listed in Section A.3.1.)

ERL contains the number of the line at which the error occurred. (The LINO function, described in Section 4.8.4, is used in expressions with the ERL function to maintain correct line-number references after a RESEQUENCE command.)

ERN\$ contains the name of the program or subprogram in which the error occurred.

You can use these variables within your error-handling routine. For example:

```
READY
LISNH
00010 ONERROR GOTO 50
00020 INPUT "TYPE A FLOATING-POINT VALUE";A
00030 PRINT A
00040 GOTO 70
00050 IF ERR = 52 THEN PRINT "STRING VALUE NOT ALLOWED&
    AT LINE";ERL;"OF ";ERN$;" ---- TRY AGAIN"
00060 RESUME
00070 END
```

PROGRAM CONTROL

```
READY
RUNNH
TYPE A FLOATING-POINT VALUE ? A STRING
STRING VALUE NOT ALLOWED AT LINE 20 OF MAIN PROGRAM --- TRY AGAIN
TYPE A FLOATING-POINT VALUE ? ANOTHER STRING
STRING VALUE NOT ALLOWED AT LINE 20 OF MAIN PROGRAM --- TRY AGAIN
TYPE A FLOATING-POINT VALUE ? 5.678
5.678
```

READY

This example makes use of all three BASIC error variables when the program traps an error. The IF statement in line 50 uses the ERR variable to determine if the trapped error was error number 52. If you type a string into an INPUT statement when the variable in the INPUT statement takes only a floating-point value, error number 52 is generated.

If you do not include an error routine in this program and you type a string when a floating-point value is required, BASIC prints error message 52 and terminates the program, as follows:

```
READY
LISNH
00010 INPUT "TYPE A FLOATING-POINT VALUE";A
00020 PRINT A
00030 END
```

```
READY
RUNNH
TYPE A FLOATING-POINT VALUE ? A STRING

? 52 Invalid floating point number at line 00010 of MAIN PROGRAM

READY
```

4.8.4 The LINO Function

The LINO function indicates to BASIC that a number represents a statement line number. For the RESEQUENCE command to properly alter a line number that appears in an expression, the line number must appear as an argument to the LINO function.

Although you can use the LINO function in any expression, it is most useful when used in conditional expressions involving the ERL error variable. The format of this function is as follows:

```
LINO(line number)
```

where:

```
line number          is a valid line number.
```

PROGRAM CONTROL

The following example shows the LINO function used to maintain the same line reference before and after a RESEQUENCE command.

```
READY
OLD LINO.B20
```

```
READY
LISNH
00010 ONERROR GOTO 50
00020 INPUT "PLEASE ENTER A VALUE";A
00030 PRINT A
00040 GO TO 70
00050 IF ERL = LINO(20) THEN PRINT "TRY AGAIN"
00060 RESUME
00070 END
```

```
READY
RESEQUENCE
```

```
READY
LISNH
00100 ONERROR GOTO 140
00110 INPUT "PLEASE ENTER A VALUE";A
00120 PRINT A
00130 GO TO 160
00140 IF ERL = LINO(110) THEN PRINT "TRY AGAIN"
00150 RESUME
00160 END
```

```
READY
```

Note that after the RESEQUENCE command is given, the expression in line 140 refers to the same line as it did before the RESEQUENCE.

4.9 STATEMENT MODIFIERS

THE BASIC statement modifiers are a collection of five keywords that are used with certain BASIC statements to qualify or restrict the execution of that statement. These modifiers enable you to:

1. Indicate conditional execution of a statement (IF, UNLESS, WHILE, UNTIL modifiers)
2. Create an implied loop (FOR modifier)

These statement modifiers cannot stand alone; they must be appended to a statement. Most BASIC statements can have modifiers. There are some, however, that cannot. Table 4-1 lists the BASIC statements that can and cannot have modifiers.

PROGRAM CONTROL

Table 4-1
BASIC Statements

Can Have Modifiers		Cannot Have Modifiers
CALL	NAME-AS	COMMON
CHAIN	NODATA	DATA
CHANGE	ONGOSUB	DEF
CLOSE	ONGOTO	DIM
DELETE	ONERROR	END
FIND	OPEN	FNEND
FNEXIT	PAGE	FOR
GET	PRINT	IFEND
GOSUB	PUT	IFMORE
GOTO	RANDOMIZE	IMAGE
IF-THEN-ELSE	READ	MAP
INPUT	RESET	NEXT
INPUT LINE	RESTORE	REM
KILL	RESUME	SUB
LET	RETURN	SUBEND
LINPUT	SCRATCH	UNTIL
MARGIN	SLEEP	WHILE
Matrix	STOP	
Initialization	SUBEXIT	
MAT INPUT	WAIT	
MAT PRINT	UPDATE	
MAT READ		
MOVE		

When you use statement modifiers with the various formats of the IF statement, the following rules apply:

1. Append statement modifiers to either the THEN clause or the ELSE clause of an IF statement.
2. The statement modifier applies only to the clause it is appended to and not to the statement as a whole.

If you have more than one statement on a line, the modifier applies only to the statement immediately preceding it. You can also append more than one statement modifier to a single statement. In this case, BASIC processes the modifiers from right to left. Statement modifiers are reserved words. See Appendix D for a list of BASIC reserved words.

4.9.1 The IF Modifier

The IF modifier has the following format:

statement IF condition

where:

statement is any statement from the first two columns of the above table.

condition is any numeric expression.

PROGRAM CONTROL

BASIC tests to see if the condition is true or false; the statement executes only if the condition is true. For example:

```
00010 PRINT X IF X <> 0
```

BASIC prints the value X only if X is not equal to 0. The example is the same as the IF-THEN-ELSE statement:

```
00010 IF X <> 0 THEN PRINT X
```

You cannot add an ELSE or a THEN clause to the IF modifier. However, the reverse is true: you can add the IF modifier in a THEN or ELSE clause. The IF modifier in the following example applies only to the statement PRINT B:

```
00010 IF A=B THEN PRINT B IF B<100
```

BASIC prints the value of B only when both the following conditions are true:

1. A is equal to B, and
2. B is less than 100.

4.9.2 The UNLESS Modifier

The UNLESS modifier has the following format:

```
statement UNLESS condition
```

where:

statement is any statement from the first two columns of the Table 4-1.

condition is any valid numeric expression.

BASIC tests to see if the condition is true or false; the statement executes only if the condition is false. For example:

```
00010 PRINT A UNLESS A=0
```

BASIC prints the value of A only if A is not equal to 0.

The following statements each produce the same results as the UNLESS modifier:

```
PRINT A IF NOT A=0
```

```
IF NOT A=0 THEN PRINT A
```

```
IF A <> 0 THEN PRINT A
```

The UNLESS modifier simplifies the negation of a logical condition.

PROGRAM CONTROL

4.9.3 The WHILE Modifier

The WHILE modifier has the following format:

```
statement WHILE condition
```

where:

statement is any statement from the first two columns of Table 4-1.

condition is any valid numeric expression.

BASIC tests to see if the condition is true or false; the statement executes repeatedly as long as the condition is true.

For example:

```
00010 Y=2
00020 Y=Y^(2) WHILE Y<1E6
00030 PRINT Y
```

Line 20 executes over and over as long as Y^2 is less than $1E6$. When Y^2 is greater than or equal to $1E6$, BASIC executes line 30.

The WHILE modifier sets up a loop wherein one statement executes iteratively if the condition is true. There is no formal control variable as in a FOR-NEXT loop. Instead, the structure of the loop modifies the values which determine when to terminate the loop.

The previous example is equivalent to:

```
00010 Y=2
00020 Y=Y^(2)
00030 IF Y^(2)<1E6 GOTO 40 ELSE GOTO 50
00040 PRINT Y
00050 END
```

Be careful not to create an infinite loop with the WHILE modifier. The following sequence never terminates properly:

```
00010 X=X+1 WHILE I<1000
```

I is automatically set to 0 at the beginning of program execution as usual; therefore I is less than 1000. The condition of the WHILE modifier is unrelated to the assignment $X=X+1$. Because 0 is always less than 1000, the statement causes an infinite loop.

In the following example, line 10 reads the data in line 30 until it reaches the constant 10. Then the WHILE condition is no longer true, and line 20 executes.

```
00010 READ Z WHILE Z < 10
00020 IF Z >= 10 THEN PRINT "?WHILE ON READ FAILED."
00030 DATA 1,2,3,4,5,6,7,8,9,10
00040 END
```

```
READY
RUNNH
?WHILE ON READ FAILED.
```

PROGRAM CONTROL

4.9.4 The UNTIL Modifier

The UNTIL modifier has the following format:

```
statement UNTIL condition
```

where:

statement is any statement from the first two columns of Table 4-1.

condition is any valid numeric expression.

BASIC tests to see if the condition is true or false; the statement executes repeatedly as long as the condition is false. For example:

```
00005 X = 40
00010 X = X + 1 UNTIL X >795
00020 PRINT X
```

Line 10 executes repeatedly as long as X is less than or equal to 795. The statement continues until the condition becomes true.

The UNTIL modifier is similar to the WHILE modifier in that it does not need a formal control variable to determine loop termination.

The previous example is equivalent to:

```
00005 X = 40
00010 IF X > 795 THEN GOTO 25
00015 X = X + 1
00020 GOTO 10
00025 PRINT X
```

Be careful not to create an infinite loop with the UNTIL modifier:

```
00010 A=1\ B=2\ C=3
00020 LET D=C+2*A UNTIL D>=50
00030 IF D>=50 THEN PRINT D
```

In the above example, line 20 continues to execute as long as D is less than 50. Since D is never greater than or equal to 50, this example is an infinite loop at line 20.

4.9.5 The FOR Modifier

The FOR modifier has two formats:

```
statement FOR variable = num expr1 [ {STEP} num expr2 ] {WHILE} conditional expr
                                   {BY }                               {UNTIL}
```

```
statement FOR variable = num expr1 TO num expr2 [ {STEP} num expr3 ]
                                                  {BY }                               }
```

PROGRAM CONTROL

The FOR modifier is used to create an implied loop on a single line. For example:

```
00010 PRINT I, SQR(I) FOR I=1 TO 10
```

is equal to:

```
00010 FOR I= 1 TO 10
00020 PRINT I, SQR (I)
00030 NEXT I
```

By using the FOR modifier for simple loops, you eliminate the need for the FOR-NEXT statement. Keep in mind that the FOR modifier applies only to one statement on the line. Hence, it iterates only one statement, even in multi-statement lines. You can have many FOR modifiers in a single program.

The STEP and BY clauses increment the loop index just as they do in the FOR statement. The default is +1.

```
00010 PRINT A=B*C FOR I=1 TO 50 STEP 3
```

If you use the WHILE or UNTIL option, the loop continues as long as the WHILE condition is true, or as long as the UNTIL condition is false.

The following is an example of a FOR modifier and an IF modifier:

```
00010 DIM X (100)
00020 PRINT I, X(I) IF X(I) <>0 FOR I=1 TO 100
```

With more than one modifier, BASIC reads from right to left. Therefore, the implied loop, I=1 TO 100, executes first, then the IF modifier is tested. Appending more than one modifier to a statement is known as nesting modifiers.

In the following example, the modifiers are tested from right to left. If the first modifier fails, BASIC continues execution at the next statement of the program (not the next modifier on the line).

```
00010 LET A=A+J FOR J=1 TO 10 IF A+J<10
00020 LET B=B-J FOR J=1 TO 10 UNLESS B>-10
00030 LET C=C+J*2 FOR J=1 TO 4 WHILE C<10
00040 LET D=D-J FOR J=2 TO 10 STEP 2 UNTIL D=-10
00050 LET F=I+J FOR I=1 TO 5 FOR J=2 TO 6
00060 END
```

CHAPTER 5

PROGRAM SEGMENTATION

The methods of dividing large programming tasks into a series of smaller and more easily managed modules are referred to as program segmentation. Program segmentation is useful when you have a large program that you want to divide into a series of smaller program segments or discrete programs. BASIC offers two methods for dividing large programs. They are subprogramming and chaining.

A subprogram is a series of statements which are executed only when called from the main program or another subprogram. Often an efficient way to structure a BASIC program is to have a main program that calls individual subprograms when certain tasks must be performed. Structuring a program in this way makes it clear and easy to debug.

The second method of program segmentation, chaining, enables you to transfer control from the current program in the work area to a second program, which is brought into the work area to replace the first program. By dividing a program into two separate programs, connected by a CHAIN statement, you can avoid any size limitations imposed by the system.

You can define common variable storage in memory that enables you to retain variable values when control passes from a one program segment to another, or when control passes from one program to another. Program segmentation is described in the following sections:

- 5.1 Using Subprograms
- 5.2 Transferring Control to Another Program - The CHAIN Statement
- 5.3 Declaring Common Variable Storage - The COMMON Statement

5.1 USING SUBPROGRAMS

A subprogram allows you to divide a large task into smaller, more manageable units which in turn can be accessed individually.

You can use subprograms in two ways:

1. As a segment of a main program which can be called several times from the main program
2. As a mini program, which can be called by several different main programs

In both cases, the subprogram is executed by a CALL statement contained in the main program.

PROGRAM SEGMENTATION

The SUB statement is the first line of a subprogram and has the following format:

```
SUB name [(dummy argument(s))]
```

where:

name is the unique name for the subprogram. (The length of the name is system-specific.)

(dummy argument(s)) represent one or more parameter variables and file references separated by commas. The variables must agree in type and number with those of the calling sequence.

If you use the SUB statement in a multi-statement line, it must be the first statement in that line.

The body of the subprogram may contain any valid BASIC statement except for statements which affect transfer out of the subprogram. Transfers into a subprogram must be to a SUB statement; transfers out of a subprogram must be from a SUBEXIT or SUBEND statement.

All variables in a subprogram are local to that subprogram. These local variables are initialized to 0 or a null string upon each entry to the subprogram. Also, any data used from DATA statements are local to the subprogram. The DATA pointer in the main program is not affected by subprogram values.

To exit from a subprogram, use a SUBEND statement or SUBEXIT statement. The SUBEND statement has the following format:

```
SUBEND
```

The SUBEND statement must be the final statement of a subprogram.

The SUBEXIT statement has the following format:

```
SUBEXIT
```

SUBEXIT returns control to the calling program via the SUBEND statement. It has the same effect as GOTO n, where n is the line number of the appropriate SUBEND statement.

SUBEXIT is invalid in a main program or multi-line function definition.

5.1.1 Executing a Subprogram - The CALL Statement

The CALL statement transfers control to the subprogram, provides transfer of parameter values, and saves the state of the calling program.

The CALL statement has the following format:

```
CALL name [(actual argument(s))]
```

PROGRAM SEGMENTATION

where:

name is the subprogram name defined in the SUB statement.

(actual argument(s)) is one or more variables, constants, and expressions, separated by commas. These parameters must agree in position, type, and number with the dummy list in the SUB statement.

You can place the CALL statement anywhere in a main program, subprogram, or multi-line DEF. When you reference a subprogram with the CALL statement, BASIC replaces the dummy arguments with the corresponding actual arguments listed with the CALL. The subprogram then works with these parameters.

The following is a SUB statement:

```
00500 SUB TEST (A,B$)
```

This is a corresponding CALL statement that transfers control to the subprogram TEST:

```
00050 CALL TEST (C,A$)
```

Upon returning to the main program, BASIC executes the statement following the CALL statement.

5.1.2 Using Dummy and Actual Arguments

Because you can reference subprograms at more than one point throughout a program, many of the values used by the subprogram may change each time it is used. Dummy arguments in subprograms take the place of the actual values passed to the subprogram when it is called.

Dummy arguments indicate the data type of the actual arguments they represent. The position, number, and type of each dummy argument in the SUB statement list must agree with the position, number, and type of each actual argument in the CALL statement list.

Items passed to subprograms can be any valid variable, constant, expression, array, or array element. The value of any item can be used as a file number in the subprogram. BASIC passes items from the main program to the subprogram either by (1) value or by (2) reference or address.

When passing by value, BASIC makes a temporary copy of the value in the calling program and uses the copy for calculations in the subprogram. The value in the calling program remains unchanged. The following items are passed by value:

1. constants
2. expressions
3. array elements

PROGRAM SEGMENTATION

When passing by reference or address, BASIC takes the actual value from the location in the main program, uses the value in the subprogram, then replaces the value in the main program. In this case, because of calculations in the subprogram, the value passed by reference could change in the main program. The following items are automatically passed by reference:

1. variables
2. entire arrays

It is not possible to pass entire arrays by value. Individual elements of a list or table, however, are always passed by value. When an individual entry in an array is passed to a subprogram, it is received as a numeric or string variable depending on its type. For example:

```
20 CALL ELEMENTS (BCD(5))  
50 SUB ELEMENTS (ARRAY)
```

The CALL passes the copy of the value in array element BCD(5) to the subprogram. The SUB statement, in subprogram ELEMENTS, accepts the value in the variable name ARRAY.

If you specify an entire array in either argument list, you do not include the subscript. For example:

```
C( ) is a list, that is, a one-dimensional array  
C(,) is a matrix, that is, a two-dimensional array
```

When you pass an entire array to a subprogram, its dimensions remain the same as in the main program, although its values may have changed. It is invalid to use a DIM (dimension) statement on an array you specify in the SUB statement. You can, however, redimension such an array with a MAT statement (See Section 7.1). The array will then be redimensioned in the main program as well. Arrays local to the subprogram must appear in DIM statements within the subprogram.

Functions can be defined inside subprograms. A function definition is local to the subprogram in which it is defined. However, you can pass the value of a function as an expression.

You can also pass files to a subprogram. BASIC passes the position of the file pointer to the subprogram unchanged from its position after the last operation affecting the file in the main program. Any operation on a file in a subprogram also affects the file in the main program.

You can also open a file within a subprogram. The file remains open after BASIC returns to the main program. When you include a file reference in a SUB statement, the reference must be a variable name.

PROGRAM SEGMENTATION

The following example illustrates argument lists in the SUB and CALL statements:

```

LISNH
00010 AZ=5%\B%=10%\C%=15%
00020 CALL ARG (B%)
00030 CALL ARG (C%)
00040 PRINT 'AZ= ';AZ; 'B%= ';B%; 'C%= ';C%
00050 END
00060 SUB ARG (D%)
00070 AZ = 30%
00080 D% = D%*AZ
00090 SUBEND

READY
RUNNH
AZ= 5 B%= 300 C%= 450
    
```

The subprogram ARG is called twice in this program with the CALL statements on lines 20 and 30. The first time the subprogram is referenced, the value stored in B% is passed to the integer variable D%. The second CALL statement passes the value stored in C% to the integer variable D%. Notice that variables are local to the subprogram. Consequently, you can use the same variable name in a main program and a subprogram without interference.

The following table summarizes the proper form for variable names, functions, arrays, and files references in SUB and CALL statements.

Table 5-1
Argument Data Types

Data Type	Dummy Argument SUB Statement	Actual Argument CALL Statement
real	A	B
integer	A%	B%
string	A\$	B\$
entire list	A(), A%(), A\$()	B(), B%(), B\$()
entire matrix	A(,), A%(,), A\$(,)	B(,), B%(,), B\$(,)
file	A,C	l, N
array element	A	D(I)

5.2 TRANSFERRING CONTROL TO ANOTHER PROGRAM - THE CHAIN STATEMENT

The CHAIN statement transfers control from the current program to a program stored in a file. CHAIN first closes all files and erases all program lines, arrays, and variables. It does not necessarily erase variables in blank common; see Section 5.3.2. Finally, it loads the program from the file, compiles it if necessary, and starts the execution of the new program. Thus, the CHAIN statement has the same effect as an END statement followed by an OLD command and then a RUN command.

The format of the CHAIN statement is:

```
CHAIN string [LINE line number]
```


PROGRAM SEGMENTATION

where:

string is a file specification for the file containing the new program. The string can be any string expression. If no file type is specified, .EXE is assumed.

LINE line number specifies the line to start execution in the new program. If no line number is specified, then execution starts at the lowest numbered line.

Consider the following example:

The file specified by "SEG1" contains:

```
00005 PRINT "SEG1 IS WORKING"           !PRINTS IDENTIFYING MESSAGE
00010 OPEN "DATA1" FOR OUTPUT AS #1     !OPENS OUTPUT FILE
00020 FOR I=1 TO 100                    !WRITES OUT ALL THE
00030 PRINT #1, I*2                     !EVEN NUMBERS 2 TO 200
00040 NEXT I                             !TO THE FILE
00050 CLOSE #1                           !CLOSES THE FILE
00060 CHAIN "SEG2.B20"                  !CHAINS TO THE NEXT
00070 END                                !SEGMENT
```

The file specified by "SEG2" contains:

```
00005 PRINT "SEG2 IS WORKING"           !PRINTS IDENTIFYING MESSAGE
00010 OPEN "DATA1" FOR INPUT AS #1      !OPENS EXISTING FILE
00020 FOR I=1 TO 100                    !INPUTS THE NUMBERS
00030 INPUT #1, J                        !FROM THE FILES
00040 T=T+J                              !AND ADDS THEM TOGETHER
00050 NEXT I                             !STORING THE TOTAL IN T
00060 PRINT "THE TOTAL IS";T            !PRINTS THE TOTAL ON THE
00070 CLOSE #1                           !CLOSES INPUT FILE
00080 END
```

A run of these programs produces the following output:

```
RUNNH
SEG1 IS WORKING
SEG2 IS WORKING
THE TOTAL IS 10100
```

If the file specified by the CHAIN statement does not exist, BASIC prints an error message. To allow error recovery, BASIC does not erase the current program lines, variables, or arrays from the work area. However, all files are closed as they normally would be.

Be sure to SAVE a program containing a CHAIN statement before running it; otherwise, the program will erase itself from memory.

5.3 DECLARING COMMON VARIABLE STORAGE - THE COMMON STATEMENT

The COMMON statement defines variables whose values are shared among program segments (for example, a main program and subprograms) connected by a SUB statement or between two programs connected by a CHAIN statement.

PROGRAM SEGMENTATION

The variables declared within a COMMON statement (COMMON variables) may be any BASIC variable, including arrays. Arrays are declared in a COMMON statement the same way they are declared in a DIMENSION statement. Do not, however, define the same array in the same program segment using both a DIMENSION statement and a COMMON statement. This redundancy causes a fatal compiler error.

The format of the COMMON statement is as follows:

```
COM [MON] [(name)] var [, var, var...]
```

where:

COM is the unique abbreviation.

name is an optional variable name given to the COMMON area. If a name is included, the resulting area defined by the statement is termed "named" COMMON. If a name is not included, the resulting area defined by the statement is termed "blank" COMMON.

var is a real, integer, string, or array variable, or any combination of these. Separate multiple variable declarations with commas.

The conventions to follow when using the COMMON statement vary depending on whether you want to declare COMMON variables whose values are retained across a CALL statement or across a CHAIN statement.

To declare COMMON variables across a CALL statement, see Section 5.3.1. To declare COMMON variables across a CHAIN statement, see Section 5.3.2.

5.3.1 Sharing COMMON Variables across a CALL Statement

The COMMON statement declares variables whose values are retained when one program segment transfers control to another program segment using a CALL statement. Unlike when arguments are passed to a subprogram using the CALL statement (Section 5.1.2), the COMMON statement declares that variables are local to all program segments that include a COMMON statement accessing the same area.

The following list of rules pertains to sharing variable values across a CALL statement.

1. Each program segment must contain a COMMON statement referencing a COMMON area by name. All COMMON statements referencing the blank COMMON area must have no name.
2. The variables in corresponding COMMON statements must agree positionally in data type, although the variable names may be different. If the data types do not match positionally, BASIC equivalences the data values without issuing an error message.
3. A variable used in a COMMON statement must be declared in the COMMON statement before it can be used anywhere else in the program segment.
4. Variables in a COMMON area retain their values when a program segment is called that contains a COMMON statement declaring the same area.

PROGRAM SEGMENTATION

5. COMMON string variables are fixed length. The string length must be specified in the same manner as it is specified in the MAP statement, for example:

```
00010 COMMON LINE$=80,MY.ARRAY$(5%)=10,XX$
```

In this example, LINE\$ has a fixed length of 80 characters; each element of MY.ARRAY\$ has a length of 10 characters; the variable XX\$, however, has a default length of 16 characters.

If the string is larger than the defined length, BASIC left-justifies and truncates the string to the defined length. If the string is smaller than the defined length, BASIC left-justifies and pads the string on the right with blanks.

6. COMMON variables retain their values until BASIC executes an END or a CHAIN statement.
7. The initial COMMON statement must be larger than all other statements referencing the same COMMON area.
8. COMMON statements that are on multiple lines but that declare the same COMMON area can have no other intervening BASIC statements or COMMON statements declaring other COMMON areas. For example:

Valid

```
00010 COMMON (FIRST) A,B
00020 COMMON (FIRST) C
00030 COMMON (SECOND) Z$
```

Invalid

```
00010 COMMON (FIRST) A,B
00020 COMMON (SECOND) Z$
00030 COMMON (FIRST) C
```

In the valid example, the declaration of COMMON area FIRST is equivalent to:

```
00010 COMMON (FIRST) A,B,C
```

The following example shows how the COMMON statement is used to share variable values between a program and a subprogram:

```
00010 COM (CAT) A,X$,M%
00020 COM (DOG)Z
00030 A=1
00040 X$='HELLO'
00050 M%=7
00060 PRINT A;X$;M%
00070 CALL SUBPRG
00080 END
00090 !
00100 SUB SUBPRG
00110 COM (CAT)F,T$,L%
00120 PRINT F;T$;L%
00130 SUBEND
```

PROGRAM SEGMENTATION

Note that the variable names in the two COMMON statements do not need to be the same. Execution of this program demonstrates that the three variables have been passed from the main program to the subprogram. The program also contains the variable Z in the main program. Since COMMON (DOG) has not been declared in the subprogram, the subprogram cannot reference the location containing Z in the main program.

RUN

NONAME.B20

Friday, March 2, 1979 15:16:44

```
1 HELLO          7
1 HELLO          7
```

5.3.2 Sharing COMMON Variables across a CHAIN Statement

When creating COMMON statements to share variable values across a CHAIN statement, adhere to the rules listed in Section 5.3.1 in addition to the rules for using blank COMMON statements listed in this section.

The following list of rules pertains to using the COMMON statement to retain variable values across a CHAIN statement.

1. A blank COMMON statement must be the first COMMON statement in both programs for the variable values to be retained across a CHAIN statement.
2. The COMMON variable declarations in both programs must agree in data type, position, and number. If the variables do not agree in data type, position, and number, BASIC prints a warning message indicating that the variable values have not been preserved and initializes each variable in the second COMMON statement to zero, if the variable is numeric, or to a null string, if the variable is a string.
3. Multiple blank COMMON statements declaring variables whose values will be retained across a CHAIN statement can have no other intervening BASIC statements or any COMMON statements declaring other areas.

The following example demonstrates the use of blank COMMON statements for retaining variable values across a CHAIN statement:

NONAME.B20

Monday, March 5, 1979 00:51:44

```
00010 COMMON A$, B%, C
```

```
00020 A$ = 'A String'
```

```
00030 B% = 12345
```

```
00040 C = 123.34
```

```
00050 PRINT 'The lines below will be the same if COMMON preserved.'
```

```
00060 PRINT
```

```
00070 PRINT A$, B%, C
```

```
00080 CHAIN 'CHNPRG.B20'
```

```
00090 END
```

READY

RUN

NONAME.B20

Monday, March 5, 1979 00:51:49

PROGRAM SEGMENTATION

The lines below will be the same if COMMON is preserved.

A Strings	12345	123.34
A Strings	12345	123.34

Compile time: 0.129 secs
Run time: 0.217 secs Elapsed time: 0:00:01

READY
LIST

CHNPRG.B20
Monday, March 5, 1979 00:51:54

00010 COMMON A\$, B\$, C
00020 PRINT A\$, B\$, C
00030 PRINT
00040 END

In this example, a blank COMMON area is used in the first program to declare variables that will be available to the program called by the CHAIN statement. The blank COMMON statement must be the first statement in any program which accesses the blank COMMON area.

CHAPTER 6
USING FUNCTIONS

Functions perform a series of numeric or string operations on the arguments you specify and return a result to BASIC. You can use functions which return numeric values in numeric expressions and functions which return string values in string expressions. BASIC provides numeric functions; string functions; conversion functions; date, time, and directory functions; terminal-format file functions; and user-defined functions.

The BASIC functions are described in the following sections:

- Section 6.1 Numeric Functions
- Section 6.2 String Functions
- Section 6.3 Conversion Functions
- Section 6.4 Date, Time, and Directory Functions
- Section 6.5 Terminal-Format File Functions
- Section 6.6 System Functions
- Section 6.7 User-Defined Functions

6.1 NUMERIC FUNCTIONS

The BASIC-PLUS-2 numeric functions perform standard mathematical operations.

BASIC provides the following trigonometric functions:

1. SIN - sine
2. COS - cosine
3. TAN - tangent
4. COT - cotangent
5. ATN - arctangent
6. ATN2 - two-argument arctangent

In addition, BASIC has a special function, PI, which returns the value of a transcendental number frequently used as a trigonometric constant.

USING FUNCTIONS

BASIC also has algebraic functions:

1. SQR - the square root of a number
2. EXP - the value of e, an algebraic constant, raised to any power
3. LOG and LOG10 - the natural and common logarithms of a number
4. INT - the integral part of a number
5. ABS - the absolute value of a number
6. SGN - the sign of a number
7. FIX - the truncated value of a number

All BASIC numeric functions return real numbers (internally) as opposed to integer numbers. A numeric argument to a function is converted to integer by truncation before it is used in calculations.

6.1.1 Trigonometric Functions (PI, SIN, COS, TAN, COT, ATN, ATN2)

The format of these functions is:

```
PI
SIN(expression)
COS(expression)
TAN(expression)
COT(expression)
ATN(expression)
ATN2(expression,expression)
```

BASIC provides functions, SIN and COS, to find the sine and cosine of an angle in radians. The PI function returns a numeric constant, 3.141593. Do not include an argument with PI; if you do, BASIC prints an error message.

Consider the following example that uses the SIN, COS, and PI functions:

```
READY
LISNH
00010 REM - CONVERT ANGLE (X) TO RADIANS, AND
00020 REM - FIND SIN AND COS
00030 PRINT "DEGREES","RADIANS","SINE","COSINE"
00040 INPUT X \ GOTO 999 IF X<0
00050 LET Y=X*PI/180
00060 PRINT X,Y,SIN(Y),COS(Y)
00070 GO TO 40
00999 END
```

USING FUNCTIONS

READY RUNNH DEGREES	RADIANS	SINE	COSINE
? 0			
0	0	0	1
? 10			
10	0.1745329	0.1736482	0.9848078
? 20			
20	0.3490658	0.3420201	0.9396926
? 30			
30	0.5235988	0.5	0.8660254
? 360			
360	6.283185	0	1
? 45			
45	0.7853982	0.7071068	0.7071068
? -1			

READY

Note that in this example, PI is used to convert degrees to radian measure (line 50).

The TAN function returns the tangent of the argument you supply. The TAN function has the following format:

TAN(expression)

where:

expression is given in radians.

The COT function returns the cotangent of the specified argument. The COT function has the following format:

COT(expression)

where:

expression is given in radians. The functions COT(X) and TAN(PI/2-X) are equivalent.

The ATN function returns the value in radians of the angle whose tangent is equal to the specified expression. The format of the ATN function is:

ATN(expression)

The ATN function returns a value in the range +PI/2 to -PI/2.

The ATN2 function returns the value of the angle whose tangent is equal to expression1 divided by expression2. The ATN2 function has the following format:

ATN2(expression1,expression2)

where:

expression (both expressions) must be given in radians. If the division causes an underflow, a warning message is printed and the function returns a zero.

USING FUNCTIONS

The following example tests the ATN function. The user inputs an angle in degrees, converts it to radians, and calculates the tangent of the angle according to this formula:

$$\text{TAN}(X) = \text{SIN}(X) / \text{COS}(X)$$

Then the program converts the tangent to an angle using the ATN function and prints the results. The angles returned by the ATN function should be the same as the angles supplied by the user.

```
LISTNH
20 PRINT "SUPPLY AN ANGLE IN DEGREES"
25 PRINT "ANGLE","ANGLE","TAN(X)","ATAN(X)","ATAN(X)"
26 PRINT "(DEGS)","(RADS)","(RADS)","(DEGS)"
30 INPUT X\GO TO 200 IF X<0
40 Y=X*PI/180
45 IF ABS(COS(Y))<.01 THEN 100
50 Z=SIN(Y)/COS(Y)
70 PRINT X,Y,Z,ATN(Z),ATN(Z)*180/PI !COMPUTE ARCTANGENT AND PRINT
85 PRINT ! RESULTS.
90 GO TO 30
100 PRINT "ANGLE ERROR"
110 GO TO 30
200 END
```

READY

```
RUNNH
SUPPLY AN ANGLE IN DEGREES
ANGLE          ANGLE          TAN(X)          ATAN(X)          ATAN(X)
(DEGS)         (RADS)                   (RADS)         (DEGS)
? 0
0              0              0              0              0
? 45
45            .785398            1              .785398         45
? 10
10            .174533            .176327        .174533         10
? -1
```

READY

6.1.2 Algebraic Functions

BASIC has several algebraic functions that you can use in calculations:

SQR	Square root function
EXP	Exponential function
LOG	Logarithm function
LOG10	Common logarithm function
INT	Integer function
ABS	Absolute value function
SGN	Sign function
FIX	Fix function

USING FUNCTIONS

6.1.2.1 Square Root Function (SQR) - The SQR function returns the square root of the expression you specify. The format of the SQR function is:

SQR T (expression)

The T in the function name is optional. If the value of the expression is negative, BASIC prints a warning message and the function returns the square root of the absolute value of the expression.

```
READY
LISNH
00010 INPUT X \ GOTO 999 IF X=-1
00020 LET Z=SQR(X)
00030 PRINT Z
00040 GO TO 10
00999 END
```

```
READY
RUNNH
? 16
4
? -100
```

```
% 54 Attempt to take SQR of a nesative argument at line 00020 of MAIN PROGRAM
10
? 1000
31.62278
? 12345
111.1081
? 23456
50
? 1970
44.38468
? -1
```

```
READY
```

6.1.2.2 Exponential and Log Functions (EXP, LOG, and LOG10) - The exponential function, EXP, returns e, an algebraic constant, raised to the power specified by the expression, where e is the base of the natural logarithm system. The value of e is approximately 2.71828.

The format of the exponential function is:

EXP(expression)

The logarithm function LOG returns the logarithm to the base e of the expression.

The format of the LOG function is:

LOG(expression)

EXP and LOG are related functions. Specifically, EXP is the inverse of LOG. The following formula describes their relationship:

$$\text{LOG}(\text{EXP}(X)) = X$$

USING FUNCTIONS

Consider the following examples. Note that the output from one example is used as the input for the other.

<pre> READY LISNH 00010 INPUT X\GOTO 999 IF X<0 00020 PRINT EXP(X) 00030 GO TO 10 00999 END </pre>	<pre> READY LISNH 00010 INPUT X\GOTO 999 IF X<0 00020 PRINT LOG(X) 00030 GO TO 10 00999 END </pre>
<pre> READY RUNNH ? 4 54.59815 ? 10 22026.47 ? 9.42100 12344.92 ? 4.6051704 100 ? 25 7.20049E+10 ? -1 </pre>	<pre> READY RUNNH ? 54.59815 4 ? 22026.47 10 ? 12344.92 9.421 ? 100 4.60517 ? 7.20049E10 25 ? -1 </pre>
<pre> READY </pre>	<pre> READY </pre>

The LOG10 function returns the common logarithm (base 10) of the specified value. The form of the LOG10 function is:

LOG10(expression)

Programs that require the computation of logarithm (base 10) do not have to use the conversion formula described above. For example:

```

READY
LISNH
00010 INPUT X
00020 PRINT " X", " LOG10(X)"
00030 FOR I=1 TO 5
00040 PRINT X^I,LOG10(X^I)
00050 NEXT I
00999 END
          
```

```

READY
RUNNH
? 5.732
X          LOG10(X)
5.732      0.7583062
32.85582   1.516612
188.3296   2.274919
1079.505   3.033225
6187.723   3.791531
          
```

READY

If the expression supplied for the EXP function is less than or equal to -89.415, BASIC prints a warning message and returns a zero. If the EXP expression is greater than or equal to 88.029, BASIC prints a warning message and returns the largest positive number.

If the expression supplied for the LOG or LOG10 function is equal to or less than zero, BASIC prints a warning message, and the function returns either a zero if the expression is negative or the smallest negative number.

USING FUNCTIONS

6.1.2.3 Integer Function (INT) - The integer function returns the value of the greatest integer that is less than or equal to the expression you specify. The format of the integer function is:

```
INT(expression)
```

For example:

```
READY
LISNH
00010 PRINT INT(34.47)
00020 PRINT INT(33000.9)
00999 END
```

```
READY
RUNNH
  34
33000
```

```
READY
```

The INT function always returns the value of the greatest integer that is less than the value of the specified expression when the value is positive; however, when you specify a negative number, INT produces a number whose absolute value is larger. For example:

```
READY
LISNH
00010 PRINT INT(-23.45)
00020 PRINT INT(-14.7)
00030 PRINT INT(-11)
00999 END
```

```
READY
RUNNH
-24
-15
-11
```

```
READY
```

Note that the value returned by INT is a real number.

You can use the INT function to round off numbers to the nearest integer by adding .5 to the argument. For example:

```
READY
LISNH
00010 PRINT INT(36.67+.5)
00020 PRINT INT(-5.1+.5)
00999 END
```

```
READY
RUNNH
  37
-5
```

```
READY
```

You can also use INT to round off a number to any given decimal place or any integral power of 10. Do this by using the formula:

$$\text{rounded off number} = \text{INT}(\text{number} * 10^P + .5) / 10^P$$

USING FUNCTIONS

where P represents the number of places of accuracy and is positive for accuracy to the right of the decimal point and negative for accuracy to an integral power of 10.

Consider the following example, which rounds numbers to the number of decimal places specified (line 80):

```
READY
LISNH
00010 REM PROGRAM TO ROUND OFF DECIMAL NUMBERS
00020 PRINT "WHAT NUMBER DO YOU WISH TO ROUND OFF";
00030 INPUT N
00040 IF N = -9999 GO TO 999
00050 PRINT "TO HOW MANY PLACES";
00060 INPUT P
00070 PRINT
00080 LET A=INT(N*10^P+.5)/(10^P)
00090 PRINT N;"=";A;"TO";P;"DECIMAL PLACES."
00100 PRINT
00110 GO TO 20
00999 END
```

```
READY
RUNNH
WHAT NUMBER DO YOU WISH TO ROUND OFF ? 56.1237
TO HOW MANY PLACES ? 2
```

56.1237 = 56.12 TO 2 DECIMAL PLACES.

```
WHAT NUMBER DO YOU WISH TO ROUND OFF ? 8.449
TO HOW MANY PLACES ? 1
```

8.449 = 8.4 TO 1 DECIMAL PLACES.

```
WHAT NUMBER DO YOU WISH TO ROUND OFF ? -9999
```

```
READY
```

6.1.2.4 Absolute Value Function (ABS) - The ABS function returns the absolute value of the specified expression. The form of the ABS function is:

ABS(expression)

USING FUNCTIONS

The absolute value of a number is always positive. If the expression is a positive number, the absolute value is equal to that number. If the expression is a negative number, the absolute value is equal to -1 times the number. For example:

```
READY
LISNH
00010 INPUT X \ GO TO 999 IF X=0
00020 X=ABS(X)
00030 PRINT X
00040 GO TO 10
00999 END
```

```
READY
RUNNH
? -35.7
35.7
? 2
2
? 25E20
2.5E+21
? 10555567
1.055557E+07
? 10.12345
10.12345
? -44.5556668
44.55567
? 0
```

READY

Note that the ABS function returns a real number even if the argument is an integer.

6.1.2.5 Sign(SGN) and Fix(FIX) Functions - The sign function determines whether an expression is positive, negative, or equal to 0. The format of the SGN function is:

SGN(expression)

If the expression is positive, SGN returns a value of +1. If the expression is negative, SGN returns a value of -1. If the expression is equal to zero, SGN returns a value of zero. For example:

```
READY
LISNH
00010 A=-7.32
00020 B=.44
00030 C=0
00040 PRINT "A=2;A,"B=2;B,"C=2;C
00050 PRINT "SGN(A)=";SGN(A),
00060 PRINT "SGN(B)=";SGN(B),
00070 PRINT "SGN(C)=";SGN(C)
00999 END

READY
RUNNH
A=-7.32      B= 0.44      C= 0
SGN(A)=-1    SGN(B)= 1     SGN(C)= 0

READY
```

USING FUNCTIONS

Note that the SGN function returns the values as real numbers.

The FIX function has the following format:

```
FIX(expression)
```

The FIX function returns the truncated value of the expression you supply as a real number not an integer. For example:

```
FIX(-.5) returns a 0.  
FIX(2.6) returns a 2.
```

The FIX function is equivalent to:

```
SGN(X)*INT(ABS(X))
```

6.1.3 Random Numbers (RND Function and RANDOMIZE Statement)

The RND function supplies a series of random numbers to a BASIC program. This function is useful if you want to simulate a situation that involves input of an unknown quantity, for example, a roll of the dice. When you include the RND function in a program, it produces a predictable sequence of numbers that are seemingly unrelated. Because a computer always produces the same results given the same starting conditions, the RND function does not create a truly random series of numbers. Every time you execute the same program you will receive the same series of random numbers. Therefore, the RND function is known as a pseudo random number generator.

The RND function has the following format:

```
RND
```

The RND function returns a random number between 0 and 1 but never returns the extremes of the range, 0 and 1. (This kind of range is called an open range, or open interval.) For example:

```
READY  
LISNH  
00010 PRINT RND,RND,RND,RND  
00999 END  
  
READY  
RUNNH  
0.1948187      0.7324636      0.6087399      0.3225784  
  
READY
```

The program requests four random numbers so BASIC prints 4 numbers in the open range 0 to 1.

The RND function has the same starting location each time you run the same program. However, you can change the starting point by adding the RANDOMIZE statement before the RND function in the program. Each time BASIC executes the RANDOMIZE statement, it starts the RND function at a new unpredictable location in the series. This location is determined by the current time of day according to the computer's clock.

USING FUNCTIONS

NOTE

You should not include the RANDOMIZE statement until you have debugged your program. If you do, you will not know if changes in the results are caused by changes in the program or changes in the starting location of the random number generator.

The RANDOMIZE statement has the following format:

```
RANDOM
RANDOMIZE
```

Consider the following examples which contrast RND without and with RANDOMIZE.

RND without RANDOMIZE

```
READY
LISNH
00010 PRINT RND,RND,RND,RND
00999 END
```

```
READY
RUNNH
0.1948187      0.7324636      0.6087399      0.3225784
```

```
READY
RUNNH
0.1948187      0.7324636      0.6087399      0.3225784
```

```
READY
RUNNH
0.1948187      0.7324636      0.6087399      0.3225784
```

READY

Notice every time the program without RANDOMIZE is run, RND produces the same series of values.

RND with RANDOMIZE

```
READY
LISNH
00010 RANDOMIZE
00020 PRINT RND,RND,RND,RND
00999 END
```

```
READY
RUNNH
0.5267071      0.6324583      0.6848936      0.04149863
```

```
READY
RUNNH
0.6112094      0.3614544      0.5410995      0.5208594
```


USING FUNCTIONS

```
READY
RUNNH
  0.1922475      0.5580706      0.8529094      0.4213777
```

READY

Each time the program with RANDOMIZE is run, RND produces a different random series of numbers.

You can also use the RND function to produce a series of random numbers over any given open range. To produce random numbers in the open range A to B, use the following general expression:

$(B-A) * \text{RND} + A$

For example, to produce ten numbers in the open range 4 to 6, use this program:

```
READY
LISNH
00010 FOR I=1 TO 10
00020 PRINT (6-4) * RND+4,
00030 NEXT I
00999 END
```

```
READY
RUNNH
  4.389637      5.464927      5.21748      4.645157      4.216904
  4.376965      4.123446      5.426511      5.275825      4.097507
```

READY

6.1.4 MOD Function

The MOD function has the following formats:

$\text{MOD}\%(A,B)$

$\text{MOD}(A,B)$

where:

A,B represent numeric constants you supply.

$\text{MOD}\%(A,B)$ returns the integer result of $A \bmod B$, which is the remainder of A/B .

$\text{MOD}(A,B)$ returns the real result of $A \bmod B$, which is equal to $A - B * \text{INT}(A/B)$. If A/B produces an underflow, the value of A is returned. If the division produces an overflow, or if the quotient is greater than or equal to $2^{**}26$, a warning message is printed and a value of zero is returned.

USING FUNCTIONS

6.2 STRING FUNCTIONS

BASIC provides string functions that allow you to modify strings. With these functions you can:

1. Determine the length of a string (LEN)
2. Trim off trailing blanks from a string (TRM\$)
3. Search for the position of a set of characters within a string (POS, INSTR)
4. Extract a segment from a string (SEG\$, MID\$, LEFT\$, RIGHT\$)
5. Create a string of a certain length (STRING\$)
6. Insert spaces into a string (SPACE\$)
7. Alter the contents of a string (EDIT\$)

Another group of BASIC string functions allows you to convert strings to numbers and numbers to strings. In particular, you can convert:

1. Character to ASCII code (ASCII)
2. ASCII code to character (CHR\$)
3. String representation of a number to a number (VAL%, VAL)

BASIC's relational operators allow you to concatenate and compare strings; string functions allow you to analyze the composition of a string. The following sections describe these string functions.

The functions LEFT\$, RIGHT\$, MID\$, and SEG\$ all return the null string if their string argument is null. No further range checking is done in this case.

6.2.1 Finding the Length of a String (LEN)

The LEN function returns an integer equal to the number of characters in the specified string (including trailing blanks). The format of the LEN function is:

```
LEN(string)
```

USING FUNCTIONS

For example:

```
READY
LISNH
00010 A$="ABCDEFGHIJKLMNOPQRSTUVWXYZ"
00020 PRINT LEN(A$)
00999 END
```

```
READY
RUNNH
26
```

```
READY
```

6.2.2 Trimming Trailing Blanks (TRM\$)

The TRM\$ function returns the specified string with all trailing blanks, tabs, and backspace characters removed. The format of the TRM\$ function is:

TRM\$(string)

Consider the following example in which two strings are concatenated and printed, both before and after trailing blanks have been trimmed:

```
READY
LISNH
00010 A$="ABCD "
00020 B$="EFG"
00030 PRINT "BEFORE TRIMMING:",A$+B$
00040 PRINT "AFTER TRIMMING:", TRM$(A$)+B$
00999 END
```

```
READY
RUNNH
BEFORE TRIMMING:          ABCD  EFG
AFTER TRIMMING:          ABCDEFG
READY
```

6.2.3 Finding the Position of a Segment (POS, INSTR)

Use the POS or INSTR function to find the position of a group of characters (called a segment) in a string. The formats of the functions are:

POS(string1, string2, expression)
INSTR(expression, string1, string2)

where:

string1 is the string being searched.

string2 is the segment.

expression is the character position at which BASIC starts the search.

USING FUNCTIONS

These functions search for and return the position of the first occurrence of string2 in string1, starting with the character position specified by expression. If the specified segment is found, the character position of the first character of the segment is returned. If the specified segment is not found, the function returns 0.

You can use these functions to map a string of characters to a corresponding integer which can then be used in calculations. This technique is called a table look-up: the table string is string1 and the string to be mapped is string2 in the POS function. Consider the following example which translates month names to numbers.

```
READY
LISNH
00010 REM PROGRAM TO TRANSLATE MONTH NAMES TO NUMBERS
00020 T$="JANFEBMARAPRMAYJUNJULAUGSPFEOCTNOVDEC"
00030 PRINT "TYPE THE FIRST 3 LETTERS OF A MONTH";
00040 LINFUT M$
00050 IF M$="" GO TO 999
00060 IF LEN(M$) <> 3 GO TO 200
00070 M=(POS(T$,M$,1)+2)/3
00080 IF M <> INT(M) GO TO 200
00090 PRINT M$;" IS MONTH NUMBER";M
00100 GO TO 30
00200 PRINT "INVALID ENTRY - TRY AGAIN"
00210 GO TO 100          ;AND GET ANOTHER STRING
00999 END
```

```
READY
RUNNH
TYPE THE FIRST 3 LETTERS OF A MONTH ? NOV
NOV IS MONTH NUMBER 11
TYPE THE FIRST 3 LETTERS OF A MONTH ? DEC
DEC IS MONTH NUMBER 12
TYPE THE FIRST 3 LETTERS OF A MONTH ? JAN
JAN IS MONTH NUMBER 1
TYPE THE FIRST 3 LETTERS OF A MONTH ? AUG
INVALID ENTRY - TRY AGAIN
TYPE THE FIRST 3 LETTERS OF A MONTH ?
READY
```

There are certain possible error conditions dependent on the values of the strings and the expression.

1. If string1 (the table string) is null, an error message is given.
2. If string1 is non-null and string2 (the segment) is null, 1 is returned.
3. If neither 1. nor 2. holds, and if the value of the expression is greater than the length of string1 or less than 1, an error is given.

6.2.4 Extracting a Segment from a String (SEG\$)

The SEG\$ function is used to extract a segment from a string. The original string remains unchanged. The format of the SEG\$ function is:

```
SEG$(string, expression1, expression2)
```

USING FUNCTIONS

where:

string	is the string from which the segment is copied.
expression1	specifies the starting character position of the segment.
expression2	specifies the last character position of the segment.

For example:

```
READY
LISNH
00010 PRINT SEG$("ABCDEF",3,5)
00999 END
```

```
READY
RUNNH
CDE
```

```
READY
```

If expression1 equals expression2, SEG\$ returns the character at expression1.

There are several error conditions based on the values of the expressions and the string:

1. If expression2 equals 0, a null string is returned.
2. If string is null and expression1 is less than 1, an error is given.
3. If expression1 is greater than the length of the string, an error is given.
4. If expression2 is less than 0, an error is given.
5. If expression1 plus expression2 is greater than the length of the string, an error is given.

By using the SEG\$ function and the string concatenation operator (+), you can replace a segment of a string. Consider the following example:

```
READY
LISNH
00010 A$="ABCDEFG"
00020 C$=SEG$(A$,1,2)+"XYZ"+SEG$(A$,6,7)
00030 PRINT C$
00999 END
```

```
READY
RUNNH
ABXYZFG
```

```
READY
```

Line 20 replaces the characters CDE in the string A\$ with XYZ. Examine line 20:

```
20 C$ = SEG$(A$,1,2)+"XYZ"+SEG$(A$,6,7)
```

USING FUNCTIONS

You can use similar string expressions to replace any given characters in a string.

A general formula to replace the characters in positions n through m of string $A\$$ with $B\$$ is:

$$C\$ = \text{SEG}\$(A\$,1,n-1)+B\$\text{+SEG}\$(A\$,m+1,LEN(A\$))$$

For example, to replace the sixth through ninth characters of the string "ABCDEFGHIJK" with "123456", enter the following program:

```
READY
LISNH
00010 A$="ABCDEFGHIJK"
00020 B$="123456"
00030 C$=SEG$(A$,1,5)+B$+SEG$(A$,10,LEN(A$))
00040 PRINT C$
00999 END
```

```
READY
RUNNH
ABCDE123456JK
```

```
READY
```

6.2.5 The MID\$ Function

The MID\$ function has the following format:

$$\text{MID}\left[\begin{array}{c} \$ \\ \end{array}\right](\text{string}, \text{expression1}\%, \text{expression2}\%)$$

where:

\$	is an optional, but recommended, dollar sign.
string	is a string constant or string variable.
expression1%	is a positive integer designating the starting position of the substring.
expression2%	is a positive integer designating the number of characters in the substring.

Starting with the character at expression1%, the MID\$ function returns a substring with a length of expression2%.

For example:

```
READY
LISNH
00010 ALPHA$="ABCDEFGHIJKLMNOPQRSTUVWXYZ"
00020 PRINT MID$(ALPHA$,15%,5%)
00030 PRINT
00040 PRINT MID$("ENCYCLOPEDIA",3%,6%)
00999 END
```

```
READY
RUNNH
OPQRS
```

```
CYCLOP
```

```
READY
```

USING FUNCTIONS

The following error conditions apply to the MID function:

1. If expression2 is zero, BASIC returns the null string.
2. If expression2 is less than zero, an error message is given.
3. If string is not null and expression1 is less than 1, an error message is given.
4. If expression1 is greater than the string length, an error message is given.
5. If expression1 plus expression2 is greater than the string length, an error message is given.

If expression2 is greater than zero, then

```
MID(string,expression1,expression2)
```

is equivalent to

```
SEG$(string,expression1,expression1+expression2-1)
```

6.2.6 The LEFT\$ and RIGHT\$ Functions

The LEFT\$ function has the following format:

```
LEFT[$](string, expression)
```

where:

\$	is an optional, but recommended, dollar sign.
string	represents the string that contains the substring.
expression	represents an integer constant denoting the character position where the copying should stop.

BASIC returns a substring of the string you specify, from the first character in the string to the character position you specify in the expression. For example:

```
READY
LISNH
00010 PRINT LEFT$("ABCDEFG",4)
00999 END
```

```
READY
RUNNH
ABCD
```

```
READY
```

The RIGHT\$ function has the following format:

```
RIGHT[$](string, expression)
```

USING FUNCTIONS

where:

\$	is an optional, but recommended, dollar sign.
string	represents the string that contains the substring.
expression	represents the character position where the copying begins.

BASIC returns a substring of the string you specify, starting with the character position in the expression up to the last character in the string. For example:

```
READY
LISNH
00010 PRINT RIGHT$("ABCDEFG",6%)
00999 END
```

```
READY
RUNNH
FG
```

```
READY
```

If expression is less than one or greater than the length of the string, an error message is given. Two particular cases that return a null string instead of an error message are:

```
LEFT$(string,0) and
RIGHT$(string, 1+LEN(string))
```

6.2.7 The STRING\$ and SPACE\$ Functions

The STRING\$ function has the following format:

```
STRING$(expression1%,expression2%)
```

where:

expression1%	is a positive integer constant representing the length of the string you want to create.
expression2%	is a positive integer constant representing the decimal ASCII value of the character you want in the string.

BASIC creates a string of length expression1% with characters whose ASCII value is expression2%. For example, to create a string consisting of ten uppercase A's, use the following:

```
READY
LISNH
00010 PRINT STRING$(10%,65%)
00999 END
```

```
READY
RUNNH
AAAAAAAAAA
```

```
READY
```


USING FUNCTIONS

The SPACE\$ function has the following format:

```
SPACE$(expression%)
```

where:

expression% is an integer constant representing the number of spaces you want to add to a string.

For example:

```
READY
LISNH
00010 A$="ABC"+SPACE$(5%)
00020 PRINT A$+"DEF"
00999 END
```

```
READY
RUNNH
ABC DEF
```

```
READY
```

For both the STRING\$ and SPACE\$ functions, if the number of characters is less than 0, an error message is given.

6.2.8 The EDIT\$ Function

The EDIT\$ function has the following format:

```
string var = EDIT$(string,expression%)
```

where:

string var contains the new string after alterations.

string is a string constant or string variable representing the original string.

expression% is one of the integers in the following table, or a sum of the integers.

Table 6-1
EDIT\$ Conversions

Expression%	Effect
2%	Discard all spaces and tabs.
4%	Discard excess characters: CR, LF, FF, ESC, RUBOUT, and NULL.
8%	Discard leading spaces and tabs.
16%	Reduce spaces and tabs to one space.
32%	Convert lowercase to uppercase.
64%	Convert to (and to).
128%	Discard trailing spaces and tabs.
256%	Do not alter characters inside quotes.

USING FUNCTIONS

The EDIT\$ function converts the source character string according to the decimal value of the integer represented by expression%.

For example:

```
READY
LISNH
00010 B$="DISCARD ALL SPACES AND TABS."
00020 A$=EDIT$(B$,2%)
00030 PRINT B$
00040 PRINT A$
00050 PRINT
00060 C$="REDUCE          SPACES AND          TABS          TO ONE SPACE."
00070 D$=EDIT$(C$,16%)
00080 PRINT C$
00090 PRINT D$
00999 END
```

```
READY
RUNNH
DISCARD ALL SPACES AND TABS.
DISCARDALLSPACESANDTABS.

REDUCE SPACES AND          TABS          TO ONE SPACE.
REDUCE SPACES AND TABS TO ONE SPACE.

READY
```

You can also specify the sum of two or more integers in the table for a multiple effect. For example:

```
READY
LISNH
00010 PRINT "TYPE THE INPUT STRING";
00020 LINPUT A$
00030 B$=EDIT$(A$,80%)
00040 PRINT "B$=";B$
00999 END

READY
RUNNH
TYPE THE INPUT STRING ? DATA IS OUTPUT WITH PRINT EXPRESSION
B$=DATA IS OUTPUT WITH PRINT EXPRESSION

READY
```

In line 30, the expression% 80% is a combination of 16% and 64%.

If the EDIT\$ conversion expression 256% has been selected and the corresponding string contains unpaired quote characters, an error message will be given.

USING FUNCTIONS

6.3 CONVERSION FUNCTIONS

BASIC provides several functions to do string-to-numeric and numeric-to-string conversions. The ASCII and CHR\$ functions convert a one-character string to the character's ASCII number and vice versa. These functions are often useful in analyzing the characters in a string. The RAD function converts integer values into RADIX-50 format. The XLATE function converts data in one storage format to another. The CHANGE statement converts decimal values to ASCII characters and vice versa.

The VAL%, VAL, NUM\$, and STR\$ functions convert a string representation of a number to the number and vice versa. You should use them when you want to input a numeric value as a string or to print a number without spaces around it.

6.3.1 Converting a Character to ASCII Code (ASCII)

The ASCII function returns the decimal ASCII value of the first character in the string specified.

The ASCII function has the following format:

```
ASCII(string)
```

where:

string is either a string constant, a string variable, or string expression.

For example, ASCII ("D") is equal to 68, the decimal ASCII value of D. You can also use a string variable as an argument:

```
READY
LISNH
00010 A$="DOG"
00020 PRINT ASCII(A$)
00999 END
```

```
READY
RUNNH
68
```

```
READY
```

This program prints the value of the first character in A\$.

The ASCII function returns an integer value.

6.3.2 Converting ASCII Code to a Character (CHR\$)

The CHR\$ function returns a one-character string having an ASCII value of the specified expression. Only one character is generated at a time. The format of the function is:

```
CHR$ (expression)
```

The expression is treated as modulo 128.

USING FUNCTIONS

For example:

```
READY
LISNH
00010 REM THIS PROGRAM WILL RETURN AN INPUT LETTER AND THE 2
00020 REM FOLLOWING IT ALPHABETICALLY
00030 PRINT "ENTER A LETTER A THRU Z, ENTER END WHEN FINISHED"
00040 PRINT "LETTER", "NEXT LETTER", "3RD LETTER"
00050 INPUT X$
00060 IF X$="END" GO TO 999
00070 IF X$<"A" GO TO 200
00080 IF X$>"Z" GO TO 200
00090 FOR F=ASCII(X$) TO ASCII(X$)+2 !RETURNS ASCII VALUE OF $X
00100 IF CHR$(F)>"Z" GO TO 150 !USES CHR$ TO PRODUCE NEXT
00110 PRINT CHR$(F),;
00120 NEXT F
00130 PRINT
00140 GO TO 50
00150 PRINT "END OF ALPHABET"
00160 GO TO 50
00200 PRINT "ENTRY IS NOT A LETTER A THRU Z"
00210 GO TO 30
00999 END
```

```
READY
RUNNH
ENTER A LETTER A THRU Z, ENTER END WHEN FINISHED
LETTER      NEXT LETTER    3RD LETTER
? E
E           F
? A
A           B             C
? Y
Y           Z             END OF ALPHABET
?
ENTRY IS NOT A LETTER A THRU Z
ENTER A LETTER A THRU Z, ENTER END WHEN FINISHED
LETTER      NEXT LETTER    3RD LETTER
? END

READY
```

6.3.3 Converting an Integer to RADIX-50 (RAD)

The RAD function has the following format:

```
RAD(expression%)
```

where:

expression% represents an integer constant that you supply.

The RAD function converts the integer you specify to its RADIX-50 equivalent. RADIX-50 is a character set similar to the ASCII code.

USING FUNCTIONS

6.3.4 Translating from one Storage Code to Another (XLATE)

The XLATE function converts a string from one storage code to another. For example, you may want to use data from a file written in EBCDIC format. Before you can use this data, you must first convert it to ASCII format. You do this by using the following format:

```
XLATE(string1, string2)
```

where:

string1 is a string expression called the source string.

string2 is a string expression called the table string.

XLATE takes characters sequentially from the source string and uses the value of the character (0 to 127) as an index into the table string (0 meaning the first character of the table string, 1 the second, etc.). The character selected from the table string is appended to the resultant string value unless the selected table string character has a value of 0, or the index value is one greater than the length of the table string.

For example, the following program requests a string, removes all characters except 0 through 9, and translates 8 and 9 to A and B, respectively.

```
READY
LISNH
00010 T%=STRING$(48%,0%)+ '01234567AB'
00020 LINPUT S$ \ IF S$ = '' GO TO 999
00030 PRINT XLATE(S$,T%)
00040 PRINT
00050 GO TO 20
00999 END
```

```
READY
RUNNH
? 0123456789
01234567AB

? LOTS OF DATA AND 87870987726561296 AND MORE DATA
A7A70BA77265612B6

?

READY
```

6.3.5 The CHANGE Statement

The CHANGE statement converts a string of alphanumeric characters into their ASCII decimal values and a list of decimal numbers into a string of alphanumeric characters (see Appendix E for the ASCII Table).

The CHANGE statement has two formats:

```
CHANGE list TO string variable
```

```
CHANGE {string variable } TO list
      {string expression}
```

USING FUNCTIONS

where:

`list` is a numeric or integer variable representing a 1- or 2-dimensional array of decimal values.

In the first format, the `CHANGE` statement converts a list of integers (real numbers are truncated) into a string of characters. The length of the string is determined by the value found in element 0 of the list. For example:

```
READY
LISNH
00010 FOR I=0 TO 5
00020 READ A(I)
00030 NEXT I
00040 DATA 5,65,66,67,68,69
00050 CHANGE A TO A$
00060 PRINT A$
00999 END
```

```
READY
RUNNH
ABCDE
```

```
READY
```

In this example, the `CHANGE` statement uses the first value in the list (5) to determine the length of the character string. It then converts the next five values into their ASCII representations (see Appendix E).

In the second format, the `CHANGE` statement converts a string of characters into a list of integers. The length of the string determines the value placed in element 0 of the list. For example:

```
READY
LISNH
00010 DIM A(50)
00020 READ A$
00030 CHANGE A$ TO A
00040 PRINT A(0)
00050 FOR I=1 TO A(0)
00060 PRINT A(I);
00070 PRINT
00080 NEXT I
00090 DATA ABCDEFG
00999 END
```

```
READY
RUNNH
```

```
7
65
66
67
68
69
70
71
```

```
READY
```

Notice that `A(0)` is equal to 7 because there are 7 characters in the string.

USING FUNCTIONS

6.3.6 Numbers and their String Representation (VAL%, VAL, NUM\$ and STR\$)

Three functions - VAL%, VAL, and STR\$ - convert numbers to their string representation and vice versa.

Consider these programs:

String Representations	Numbers
READY	READY
LISNH	LISNH
00010 PRINT "25"	00010 PRINT 25
00020 PRINT "25+1"	00020 PRINT 25+1
00999 END	00999 END
READY	READY
RUNNH	RUNNH
25	25
25+1	26
READY	READY

The program on the left prints the string representation of numbers, but the program on the right prints the numbers themselves. Note how "25+1" on the left is printed as it is, while the 25+1 on the right is evaluated as 26.

The VAL% function converts an integer string expression to an integer number. The VAL function converts a real number string expression to a real number.

NOTE

Since the VAL% function operates about ten times faster than the VAL function, it is suggested that VAL% be used in place of VAL wherever possible.

The formats for VAL% and VAL are as follows:

VAL%(string expression)

where:

string expression may contain the digits 0 through 9, optionally prefixed by the symbols "+" and "-", may be suffixed by the "%" character, and must be a string representation of an integer.

VAL(string expression)

where:

string expression may contain the digits 0 through 9, the letter E (for E format numbers) and the symbols "+", "-", and ".", and must be a string representation of a number.

USING FUNCTIONS

NOTE

For the VAL% and VAL functions, if the string expression does not constitute the proper type of number, or if it contains a number which is too large or too small to be represented by BASIC, an error message will be given.

The examples that follow are simple demonstrations of how VAL% and VAL are used to convert integer- and real-number strings (respectively) into their equivalent numeric values.

```
READY
LISNH
00010 !THIS PROGRAM DEMONSTRATES THE VAL% FUNCTION
00020 !
00030 PRINT
00040 PRINT 'ENTER AN INTEGER STRING';
00050 INPUT A$
00060 B%=VAL%(A%)
00070 PRINT
00080 PRINT 'THE VAL% FUNCTION CONVERTS STRING 'A$' TO INTEGER';B%
00999 END
```

```
READY
RUNNH
```

```
ENTER AN INTEGER STRING ? 34564
```

```
THE VAL% FUNCTION CONVERTS STRING 34564 TO INTEGER 34564
```

```
READY
```

```
READY
LISNH
00010 !THIS PROGRAM DEMONSTRATES THE VAL FUNCTION
00020 !
00030 PRINT
00040 PRINT 'ENTER A REAL NUMBER STRING';
00050 INPUT A$
00060 B=VAL(A%)
00070 PRINT
00080 PRINT 'THE VAL FUNCTION CONVERTS STRING 'A$' TO REAL NUMBER';B
00999 END
```

```
READY
RUNNH
```

```
ENTER A REAL NUMBER STRING ? 45.67
```

```
THE VAL FUNCTION CONVERTS STRING 45.67 TO REAL NUMBER 45.67
```

```
READY
```

The NUM\$ and STR\$ functions convert a number to its string representation. The formats for NUM\$ and STR\$ are as follows:

NUM\$(expression)

STR\$(expression)

USING FUNCTIONS

where:

expression is any valid integer floating-point number.

The NUM\$ function returns the value of expression as it would have been printed by a PRINT statement. The STR\$ function returns the same value, but without a leading or trailing space.

Consider the following example:

```
READY
LISNH
00010 PRINT "PROGRAM TO CALCULATE 5% INTEREST"
00020 PRINT "TYPE IN AMOUNT";
00030 INPUT M$
00040 IF POS(M$,"$",1) <> 1 GO TO 999
00050 A$=SEG$(M$,2,LEN(M$))
00060 M=VAL(A$)
00070 I=.05*M+.005 !CALCULATE 5% AND ROUND
00080 I$=STR$(I)
00090 I$=SEG$(I$,1,2+POS(I$,".",1))
00100 PRINT "5% INTEREST OF ";M$;" IS $";I$
00999 END
```

```
READY
RUNNH
PROGRAM TO CALCULATE 5% INTEREST
TYPE IN AMOUNT ? $100.00
5% INTEREST OF $100.00 IS $5.00
```

```
READY
```

6.4 DATE, TIME, AND DIRECTORY FUNCTIONS

The BASIC function library includes a number of functions that return the current or a specified time and/or date in both string and numeric formats, the job time, and the connected structure and directory.

6.4.1 Returns Current Clock Time (CLK\$)

The CLK\$ function takes no argument and returns the current time of day as an 8-character string in the format: hh:mm:ss. The format of the CLK\$ function is as follows:

```
CLK$
```

For example:

```
READY
LISNH
00010 PRINT "THE CURRENT TIME OF DAY IS ";CLK$
00999 END
```

```
READY
RUNNH
THE CURRENT TIME OF DAY IS 12:21:31
```

```
READY
```

USING FUNCTIONS

6.4.2 Returns Current Date in the Format: dd-mmm-yy (DAT\$)

The DAT\$ function takes no arguments and returns the current date in the format: dd-mmm-yy. The format of the DAT\$ function is as follows:

```
DAT$
```

For example:

```
READY
LISNH
00010 PRINT "THE DATE TODAY IS: ";DAT$
00999 END
```

```
READY
RUNNH
THE DATE TODAY IS: 24-May-79
```

```
READY
```

6.4.3 Returns Date in the Format: mm/dd/yy (DATE\$)

The DATE\$ function returns a date in the format: mm/dd/yy. The format of the DATE\$ function is as follows:

```
DATE$(n%)
```

where:

n% is any integer. The formula for n% is: the day of the year + (the number of years since 1970*1000). If you supply a 0, the DATE\$ function returns the current date.

If you only specify the day of the year, the year will be 1970, unless n%=0. If you do not specify a day of the year, the date will be the current month and day, with the indicated year.

For example:

```
READY
LISNH
00010 PRINT "THE DATE FOR THE 123RD DAY IN 1973 IS: ";DATE$(3123%)
00020 PRINT
00030 PRINT "TODAY'S DATE IS: ";DATE$(0%)
00040 END
```

```
READY
RUNNH
THE DATE FOR THE 123RD DAY IN 1973 IS: 5/03/73
```

```
TODAY'S DATE IS: 6/20/79
```

```
READY
```

USING FUNCTIONS

6.4.4 Returns Time in the Format: hh:mm (TIME\$)

The TIME\$ function returns the current time or a specified time n minutes before midnight. The format of the TIME\$ function is as follows:

TIME\$(n%)

where:

n% is an integer from 1 to 1440 specifying that many minutes before midnight. Specify 0 to print the current time.

For example:

```
READY
LISNH
00010 PRINT "THE TIME 143 MINUTES BEFORE MIDNIGHT WILL BE: ";TIME$(143%)
00020 PRINT
00030 PRINT "THE CURRENT TIME IS: ";TIME$(0%)
00040 END
```

```
READY
RUNNH
THE TIME 143 MINUTES BEFORE MIDNIGHT WILL BE: 21:37

THE CURRENT TIME IS: 11:13

READY
```

6.4.5 Returns Clock, CPU, or Job Connect Time (TIME)

The TIME function returns the clocktime (in seconds) since midnight, the amount of CPU time for the current job (in tenths of seconds), or the total connect time for the job (in minutes). The format for the TIME function is as follows:

TIME(n%)

where:

n% is any of the following values:

- 0% returns the clocktime (in seconds) since midnight as a floating-point number.
- 1% returns the CPU time used by the job (in tenths of seconds).
- 2% returns the total connect time (in minutes) for the current job.

USING FUNCTIONS

For example:

```
LISNH
00010 PRINT "THE NUMBER OF SECONDS SINCE MIDNIGHT IS: ";TIME(0)
00020 PRINT
00030 PRINT "THIS JOB'S CPU TIME IS ";TIME(1)/10;" SECONDS"
00040 PRINT
00050 TZ=TIME(2)
00060 HZ=TZ/60
00070 MZ=TZ-HZ*60
00080 PRINT "THIS JOB HAS BEEN LOGGED IN";HZ;"HOURS";MZ;"MINUTES"
00090 END
```

```
READY
RUNNH
THE NUMBER OF SECONDS SINCE MIDNIGHT IS: 48565

THIS JOB'S CPU TIME IS 37.6 SECONDS

THIS JOB HAS BEEN LOGGED IN 3 HOURS 20 MINUTES
```

6.4.6 Returns Connected Structure and Directory (USR\$)

The USR\$ function returns a string containing the currently connected structure and directory. The format of this function is as follows:

```
USR$
```

For example:

```
READY
LISNH
00010 PRINT "THE CURRENT CONNECTED DIRECTORY IS: ";USR$
00999 END
```

```
READY
RUNNH
THE CURRENT CONNECTED DIRECTORY IS: BASIC:<MAGRATH>
```

```
READY
```

6.5 TERMINAL-FORMAT FILE FUNCTIONS

BASIC contains a number of functions that are used specifically with terminal-format files. These functions enable you to find the margin width, the horizontal and vertical print position, and the current page count of any terminal-format file.

6.5.1 Returns Margin Width (MAR%)

The MAR% function takes the channel number of a terminal-format file as an argument and returns its margin width. The format for this function is as follows:

```
MAR%(channel number)
```

USING FUNCTIONS

where:

channel number is the channel number of a terminal-format file. This is the same number assigned to the file in the OPEN statement. If you specify 0, BASIC returns the margin width of your terminal.

For example:

```
READY
LISNH
00010 !THIS PROGRAM CHANGES THE RIGHT MARGIN SETTING ON THE TERMINAL
00020 !AND REPORTS THE NEW SETTING WITH THE MAR% FUNCTION
00030 PRINT 'THE DEFAULT TERMINAL WIDTH IS: 'MARZ(OZ)
00040 PRINT 'X' FOR IZ=1% TO 75%
00050 MARGIN 50
00060 PRINT \ PRINT
00070 PRINT 'THE NEW TERMINAL WIDTH IS: 'MARZ(OZ)
00080 PRINT 'X' FOR IZ=1% TO 75%
00090 PRINT
00999 END

READY
RUNNH
THE DEFAULT TERMINAL WIDTH IS: 72
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXX

THE NEW TERMINAL WIDTH IS: 50
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

READY
```

6.5.2 Returns Horizontal Print Position (POS%)

The POS% function returns the current horizontal print position of the specified terminal-format file. The format of this function is as follows:

POS%(channel number)

channel number is the channel number of a terminal-format file. This is the same number assigned to the file in the OPEN statement. If you specify 0, BASIC returns the current horizontal print position of the terminal.

USING FUNCTIONS

For example:

```
READY
LISNH
00010 PRINT 'X'; FOR IZ=0% TO 20%
00020 PZ=POS(0%)
00030 PRINT
00040 PRINT 'THE HORIZONTAL PRINT POSITION WAS'#PZ
00999 END

READY
RUNNH
XXXXXXXXXXXXXXXXXXXXXXX
THE HORIZONTAL PRINT POSITION WAS 21

READY
```

6.5.3 Returns Vertical Print Position (VPS%)

The VPS% function returns the current vertical print position of the specified terminal-format file. This function has the following format:

```
VPS%(channel number)
```

where:

channel number is the channel number of a terminal-format file. This is the same number assigned to the file in the OPEN statement. If you specify 0, BASIC returns the current vertical print position of the terminal.

For example:

```
READY
LISNH
00010 PRINT I% FOR IZ=0% TO 10%
00020 PRINT 'THE VERTICAL PRINT POSITION IS NOW'#VPSZ(0%)
00999 END

READY
RUNNH
0
1
2
3
4
5
6
7
8
9
10
THE VERTICAL PRINT POSITION IS NOW 11

READY
```

USING FUNCTIONS

6.5.4 Returns Current Page Count (PPS%)

The PPS% function returns the total page count of the specified terminal-format file.

```
PPS%(channel number)
```

where:

channel number is the channel number of a terminal-format file. This is the same number assigned to the file in the OPEN statement.

For example:

```
READY
LISNH
00010 PAGE 5%
00020 PRINT I% FOR I%=1% TO 11%
00030 PRINT 'THE CURRENT PAGE NUMBER IS NOW';PPS%(0%)
00999 END
```

```
READY
RUNNH
1
2
3
4
5
^L 6
7
8
9
10
^L 11
THE CURRENT PAGE NUMBER IS NOW 2

READY
```

In the above example ^L represents a formfeed.

6.6 SYSTEM FUNCTIONS

The BASIC system functions provide you with a way of controlling a number of TOPS-20 operating system capabilities from within a BASIC program.

6.6.1 Resume Program Output (RCTRL0)

The RCTRL0 function cancels the effect of CTRL/O on the specified channel. Typing CTRL/O suppresses program output on terminals. If the device is not a terminal, this function has no effect. The format of this function is as follows:

```
variable=RCTRL0(channel number)
```

variable is any BASIC variable.

USING FUNCTIONS

channel number is a numeric expression specifying an open terminal-format file which is on a terminal device.

For example:

```
READY
LISNH
00010 PRINT I%, FOR I%=1% TO 300%
00020 PRINT
00030 X=RCTRL0(0%)
00040 PRINT "TERMINAL OUTPUT RESUMED BY RCTRL0"
00050 END
```

```
READY
RUNNH
  1           2           3           4           5
  6           7           8           9          10
 11          12          13          14          15
 16          17          18          19          20
 21          22          23          24          25
 26          27          28          29          30
 31          32          33          34          35
 36          37          38          39          40...
TERMINAL OUTPUT RESUMED BY RCTRL0

READY
```

6.6.2 Disable and Enable Echoing (NOECHO and ECHO)

The NOECHO function disables echoing on the specified channel; the ECHO function reverses the effect of the NOECHO function. These functions have the following formats:

```
variable = NOECHO(channel number)
```

```
variable = ECHO(channel number)
```

where:

variable is any BASIC variable.

channel number is a numeric expression specifying the channel number of an open terminal-format file which is on a terminal device.

For example:

```
READY
LISNH
00010 PRINT "ENTER YOUR NAME";
00020 INPUT N$
00030 PRINT "ENTER THE PASSWORD";
00040 X=NOECHO(0%)
00050 INPUT P$
00060 X=ECHO(0%)
00070 PRINT
00080 PRINT "ENTER YOUR ACCOUNT";
00090 INPUT A$
00100 PRINT "THE DATA ENTERED WAS: ";N$,P$,A$
00999 END
```


USING FUNCTIONS

6.6.4 Exit from a Program (ABORT)

The ABORT function causes an exit from the running program. If you supply an argument of 0 to the function, it will exit to BASIC command level and will inhibit the printing of the READY prompt. If you supply an argument of 1, the function will erase the program from the work area and will also not print the READY prompt. The format of the ABORT function is as follows:

```
variable=ABORT(n)
```

where:

variable is any BASIC variable.

n is either 0 or 1. Use 0 to exit from the program to BASIC command level and suppress printing of the READY prompt. Use 1 to exit from the program, suppress the READY prompt, and erase the program from the work area.

For example:

```
READY
LISNH
00010 PRINT "NOTICE THAT NO 'READY' PROMPT IS TYPED WHEN"
00020 PRINT "THIS PROGRAM ENDS."
00030 PRINT "ALSO, THE PROGRAM WILL HAVE BEEN ERASED AS IF"
00040 PRINT "A 'SCRATCH' COMMAND HAD BEEN TYPED."
00050 X=ABORT(1%)
00999 END
```

```
READY
RUNNH
NOTICE THAT NO 'READY' PROMPT IS TYPED WHEN
THIS PROGRAM ENDS.
ALSO, THE PROGRAM WILL HAVE BEEN ERASED AS IF
A 'SCRATCH' COMMAND HAD BEEN TYPED.
LISNH
```

```
READY
```

6.7 USER-DEFINED FUNCTIONS - THE DEF STATEMENT

In some programs you may want to execute the same sequence of statements in several places. You use the DEF statement to define a sequence of operations as a user-defined function. You can then use this function as you would use the functions BASIC provides. There are three ways of defining functions:

1. single-line DEF statement
2. multi-line DEF statement
3. multi-line DEF* statement

USING FUNCTIONS

6.7.1 Single-Line DEF Statement

A single-line DEF statement consists of the keyword DEF followed by a function name consisting of the letters FN followed by 1 to 29 letters, digits, or periods optionally followed by a % or a \$. Therefore, the function name can have a total of 33 characters.

Valid User-Defined Function Names	Invalid User-Defined Function Names
FN	NF1
FNC%	FN A2
FNR.B\$	FNA%\$

The format of the single-line DEF statement is:

```
DEF FNa [(b1,b2,b3,...bn)]=expression
```

where:

a is 1 to 29 letters, digits, or periods followed by an optional percent sign (%) or dollar sign (\$) to represent an integer or string value. If the function name does not end in either a % or a \$, then it returns a floating-point number.

(b1,b2,b3,...bn) can be integer, floating-point, or string dummy variables.

expression may contain any of the dummy variables or any other variables in the program.

Be sure to use an expression that is the same data type, string or numeric, as indicated by the function name. If the expression is floating point and the function name is integer or vice versa, then the expression is converted to the type specified by the function name.

After the function has been defined, it can be called, or evaluated. The format for calling the function is:

```
FNa (expression1 [,expression2,...,expression5])
```

where the number of expressions must be the same as the number of dummy variables in the DEF statement.

When evaluating the function, BASIC substitutes the values for the dummy variables in the DEF statement, then evaluates the expression, and returns the result.

USING FUNCTIONS

Consider the following two programs:

Program #1

```

READY
LISNH
00010 DEF FNS(A) = A^A
00020 FOR I=1 TO 5
00030 PRINT I,FNS(I)
00040 NEXT I
00999 END
    
```

```

READY
RUNNH
    
```

1	1
2	4
3	27
4	256
5	3125

READY

Program #2

```

READY
LISNH
00010 DEF FNS(X) = X^X
00020 FOR I=1 TO 5
00030 PRINT I,FNS(I)
00040 NEXT I
00999 END
    
```

```

READY
RUNNH
    
```

1	1
2	4
3	27
4	256
5	3125

READY

These two programs produce the same output. The actual names of the arguments in the DEF statement have no significance; they are strictly dummy variables. But the data types of the variables are significant. If the DEF statement specifies a string variable, then the corresponding argument must be a string. If the DEF statement specifies a numeric variable, then the corresponding argument must be numeric. BASIC converts, as necessary, a numeric argument to the type (floating point or integer) specified by the variable in the DEF statement.

The defining expression can contain any constants, variables, BASIC-supplied functions, or any other user-defined functions except the function you are defining. For example:

```

10 DEF FNA(X) = X^2+3*X+4
20 DEF FNB(X) = FNA(X)/2 + FNA(X)
30 DEF FNC(X) = SQR(X+4)+1
    
```

You can include any variables in the defining expression. If the expression contains variables that are not in the dummy variable list, they are not dummy variables. That is, when the user-defined function is evaluated, the variables have the value currently assigned to them.

Consider the following example:

```

READY
LISNH
00010 DEF FNB(A,B) = A+X^2      !DEFINE FUNCTION...
00020 X=1                       !ASSIGN...
00030 PRINT FNB(14,87)         !EVALUATE...
00040 X=2                       !CHANGE...
00050 PRINT FNB(14,87)         !EVALUATE...
00999 END
    
```

```

READY
RUNNH
    
```

15
18

READY

USING FUNCTIONS

Note that in this example the second argument (the dummy variable B and the actual argument 87) is unused.

The expression does not have to contain any of the variables. For example:

```
READY
LISNH
00010 DEF FNA(X) = 4+2
00020 LET R=FNA(10)+1
00030 PRINT R
00999 END
```

```
READY
RUNNH
7
```

```
READY
```

Consider the following example:

```
READY
LISNH
00010 REM MODULUS ARITHMETIC PROGRAM
00020 REM FIND X MOD M
00030 DEF FNM(X,M)=X-M*INT(X/M) !DEFINE FNM MODULUS FUNCTION
00040 REM
00050 REM FIND A+B MOD M
00060 DEF FNA(A,B,M)=FNM(A+B,M) !USE MODULUS FUNCTION FNM
00070 REM
00080 REM FIN A*B MOD M
00090 DEF FNB(A,B,M)=FNM(A*B,M) !USE MODULUS FUNCTION FNM
00100 REM
00110 PRINT
00120 PRINT "ADDITION AND MULTIPLICATION TABLES MOD M"
00130 PRINT "GIVE ME AN M"; \ INPUT M
00140 PRINT \ PRINT "ADDITION TABLES MOD";M
00150 GO SUB 300
00160 FOR I=0 TO M-1
00170 PRINT I;" ";
00180 FOR J=0 TO M-1
00190 PRINT FNA(I,J,M);
00200 NEXT J \ PRINT \ NEXT I
00210 PRINT \ PRINT \ PRINT "MULTIPLICATION TABLES MOD";M
00220 GO SUB 300
00230 FOR I=0 TO M-1
00240 PRINT I;" ";
00250 FOR J=0 TO M-1
00260 PRINT FNB(I,J,M); !CALL FNB
00270 NEXT J \ PRINT \ NEXT I
00280 GO TO 999
00300 REM SUBROUTINE TO PRINT TABLE HEADINGS
00310 PRINT \ PRINT TAB(4);
00320 FOR I=0 TO M-1
00330 PRINT I; \ NEXT I \ PRINT
00340 FOR I=1 TO 3*M+4
00350 PRINT "-"; \ NEXT I \ PRINT
00360 RETURN
00999 END
```

USING FUNCTIONS

READY
RUNNH

ADDITION AND MULTIPLICATION TABLES MOD M
GIVE ME AN M ? 7

ADDITION TABLES MOD 7

	0	1	2	3	4	5	6
0	0	1	2	3	4	5	6
1	1	2	3	4	5	6	0
2	2	3	4	5	6	0	1
3	3	4	5	6	0	1	2
4	4	5	6	0	1	2	3
5	5	6	0	1	2	3	4
6	6	0	1	2	3	4	5

MULTIPLICATION TABLES MOD 7

	0	1	2	3	4	5	6
0	0	0	0	0	0	0	0
1	0	1	2	3	4	5	6
2	0	2	4	6	1	3	5
3	0	3	6	2	5	1	4
4	0	4	1	5	2	6	3
5	0	5	3	1	6	4	2
6	0	6	5	4	3	2	1

READY

6.7.2 Multi-Line DEF Statement

Some calculations are so complex that they require more than the one line used in the single-line DEF statement. BASIC's multi-line DEF statement allows you more flexibility in defining complicated function values.

The multi-line DEF statement has the following format:

```
DEF FNa [(b1,b2,b3,...)], [c1,c2,c3...]
```

where: a represents 1 to 29 letters, digits, or periods followed by an optional percent sign (%) or dollar sign (\$) to represent an integer or string function value.

 (b1,b2,b3,...) represents the dummy argument list.

 c1,c2,c3,... represents a list of variables local to the function definition. This list is optional.

Single- and multi-line DEF statements are similar in format. However, multi-line DEFs do not have the equal sign expression on the first line. Instead, the function name must appear in a defining position (such as an assignment statement) within the function definition. Otherwise, the function value will be set to 0 or null string.

USING FUNCTIONS

Multi-line functions can have from zero to any number of parameters. The variables specified in the function definition (or DEF statement) can be used only within the body of the function. Any variable referred to in the definition that is not a local variable refers to the variable of the same name outside the body of the DEF. This means that variables in the main program are global as opposed to variables specified in the DEF statement.

The FNEND statement signals the physical and logical end of the function definition. The FNEND statement has the following format:

```
FNEND
```

When BASIC executes the FNEND statement, it returns the function value to the calling statement.

You can also end a function definition with the FNEXIT statement. FNEXIT has the following format:

```
FNEXIT
```

The FNEXIT statement is equivalent to a GOTO n where n is the line number of the FNEND for the current multi-line DEF. The FNEXIT statement is valid only inside a multi-line DEF.

Most statements can be used within the function definition (between the DEF and FNEND statements). However, multi-line DEFs are local to the main program or subprogram in which they are contained. No transfers are allowed into or out of a multi-line DEF. If you attempt to transfer into the body of a multi-line DEF, BASIC will execute the next statement following the FNEND statement and issue a warning message.

DATA statements are global throughout the program. Therefore, even though they may reside within a function definition, the main program can still access them and vice versa. DIM statements within a function are local to the function definition if they apply only to local variables.

You can place a multi-line DEF anywhere in a program. The entire body of the multi-line DEF, as well as the single-line DEF, does not produce code in straight-line execution until it is called.

You call a function into action by using its name in a statement expression. With the name, you must include the actual argument list, one with the same number of arguments as in the DEF statement. The actual arguments can be constants, variables, array elements, or expressions. They must be the same data type as the dummy arguments they replace. (Whole arrays are not valid arguments.)

BASIC uses the actual arguments within the function to define the function value. Using dummy arguments in the multi-line DEF statement allows you to use the function definition many times with a different set of actual arguments.

USING FUNCTIONS

The following example illustrates the use of the multi-line DEF statement:

```
READY
LISNH
00010 DEF FNX%(A,B),C
00020 REM C IS INITIALIZED TO 0 AT FUNCTION ENTRY
00030 IF A>B THEN C=2.5
00040 FNX%=A+B+C
00050 FNEND
00060 PRINT FNX%(3,2.5)
00070 PRINT FNX%(1.1,3.1)
00999 END

READY
RUNNH
8
4

READY
```

BASIC ignores the function definition, lines 10 through 50, and begins execution at line 60. The PRINT statement calls the function with actual arguments to be substituted in the definition. A is larger than B; therefore, C is set equal to 3.4. At line 40, BASIC calculates the value to be 10.53. Because the function name is integer (%), the value returned to FNX% is 10.

6.7.3 Multi-Line DEF* Statement

Using standard multi-line DEFs you cannot transfer into and out of a function definition and retain global values for variables. There is another method of writing multi-line DEFs that allows you to transfer from a function and retain global variables. This method also uses the DEF statement; however, an asterisk (*) has been added to the keyword.

```
DEF* FNa [(b1,b2,b3,...)], [c1,c2,c3,...]
```

The asterisk tells BASIC that the BASIC-PLUS compatible form of the function definition is being used. This form allows you to include the GOTO, ONGOTO, GOSUB, and ONGOSUB statements within the body of the function in order to transfer outside the function definition. The variables you define as dummy variables during execution of the function are global while the function is still open (that is, before BASIC reaches the FNEND statement).

USING FUNCTIONS

The following example illustrates the DEF* method of multi-line DEFs:

```
READY
LISNH
00010 DEF* FNX(A)
00020 PRINT 'FUNCTION PARAMETER A=';A
00030 IF A<3 GO TO 50
00040 A=6 \ GO TO 200
00050 FNEED
00100 A=3
00110 PRINT 'MAIN ROUTINE INITIAL A=';A
00120 PRINT
00130 C=FNX(4)
00140 D=FNX(2)
00150 PRINT
00160 PRINT 'MAIN ROUTINE FINAL A=';A
00170 GO TO 999
00200 PRINT 'FUNCTION MODIFIED A=';A
00210 GO TO 50
00999 END
```

```
READY
STATUS
VERBOSE CHECK
VERBOSE HEADER
VERBOSE WARN
QUIET COMMAND
MODE DEF *
```

```
READY
RUNNH
MAIN ROUTINE INITIAL A= 3

FUNCTION PARAMETER A= 4
FUNCTION MODIFIED A= 6
FUNCTION PARAMETER A= 2

MAIN ROUTINE FINAL A= 3
```

```
READY
```

This example demonstrates the operation of a multi-line DEF*. Note that prior to executing the program, the STATUS command is used to ensure that MODE DEF* is turned on. The status of MODE (DEF* or NODEF*) is controlled by the MODE command (See Section 2.2.9.3.).

When the program is executed, the variable A is given the value 3 (line 100), and is printed at line 110.

Next, line 130 calls the FNX function and replaces the dummy argument in line 10 with the value 4. This value is printed at line 20. A is greater than 3 (condition tested at line 30); therefore, A is assigned a new value of 6 (line 40). BASIC then transfers out of the function and prints the current value of A within the function, 6. Execution continues at the next line following the function call.

USING FUNCTIONS

The second function call (line 140) sends BASIC back to the function definition. The new value of A, 2, is printed at line 20. This time, because A is less than 3, BASIC transfers to the FNEND statement (line 50). Once the function ends, the value of A is no longer global. BASIC prints the value of A outside the function, 3, and then stops.

You can define multi-line functions in either the standard (DEF) or non-standard (DEF*) way. However, all DEFs in the same program must be written the same way throughout. Functions defined either way must have argument lists agreeing in both data type and number.

CHAPTER 7

USING ARRAYS

BASIC provides a special set of statements for working with arrays. The statements each contain the keyword MAT. The MAT statements apply to both lists and matrices (that is, to one-dimensional and two-dimensional arrays), except where noted in the text. If you specify an array without subscripts (MAT A), the default is two dimensions.

For information on using arrays, refer to the following sections:

- 7.1 Dimensioning an Array
- 7.2 Initializing an Array
- 7.3 Matrix Operations
- 7.4 Array Input and Output

7.1 DIMENSIONING AN ARRAY

In a BASIC program, you dimension an array in one of two ways:

1. Explicitly, by using the DIM, MAP, or COMMON statement
2. Implicitly, by declaring a subscripted variable

Although every list has an element 0, and every matrix has a row 0 and a column 0, the MAT statements ignore and, in some cases, destroy the contents of these locations. Data should, therefore, begin in row or column 1 and never row or column 0 if MAT statements are to be used.

The MAT statements allow you to alter the number of elements in each row and column of an array as long as the total number of elements does not exceed the number originally defined. A one-dimensional array cannot be made a two-dimensional array or vice versa. Changing the size of an array in this way is called redimensioning an array.

7.2 INITIALIZING AN ARRAY

MAT statements allow you to assign values to individual array elements. The values can be set to all ones, all zeros, or zeros with ones along the main diagonal (upper left to lower right).

The MAT statement has the following format:

```
MAT name=value [(DIM1, [DIM2])]
```

USING ARRAYS

where:

name is an array already dimensioned either implicitly or explicitly.

(DIM1,DIM2) are new dimensions for the array. These dimensions are optional.

value is one of the following:

Value	Meaning
ZER	sets the value of all elements in the array to zero. All arrays, except for those in a virtual array, MAP, or COMMON area, have a value of zero when first created. ZER does not set row 0 and column 0.
CON	sets the value of all elements in the array to one. CON does not set row 0 or column 0.
IDN	sets the value of all elements in the array to zero except for those on the diagonal (upper left to lower right) which are set to one. This is called an identity matrix. The matrix must be square. IDN does not set row 0 or column 0.
NUL\$	sets the value of all elements in a string array to a null string. NUL\$ does not set row 0 or column 0.

The first three values apply to real and integer arrays; the fourth applies to string arrays.

If you do not specify new dimensions with the (DIM1,DIM2) option, the existing dimensions remain unchanged.

USING ARRAYS

Consider the following examples:

```
00010 DIM A(10,10), B(15), C(20,20)
00020 MAT A=ZER !SETS ALL ELEMENTS OF A TO ZERO
00030 MAT B=CON(10) !SETS ALL ELEMENTS OF B TO ONE AND REDIMENSIONS B
00040 MAT C=IDN(10,10) !CREATES AN IDENTITY MATRIX 10X10
00050 MAT PRINT A;
00060 MAT PRINT B;
00070 MAT PRINT C;
00080 END
```

```
READY
RUNNH
```

```
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
```

ARRAY A

```
1 1 1 1 1 1 1 1 1 1
```

ARRAY B

```
1 0 0 0 0 0 0 0 0 0
0 1 0 0 0 0 0 0 0 0
0 0 1 0 0 0 0 0 0 0
0 0 0 1 0 0 0 0 0 0
0 0 0 0 1 0 0 0 0 0
0 0 0 0 0 1 0 0 0 0
0 0 0 0 0 0 1 0 0 0
0 0 0 0 0 0 0 1 0 0
0 0 0 0 0 0 0 0 1 0
0 0 0 0 0 0 0 0 0 1
```

ARRAY C

USING ARRAYS

7.3 MATRIX OPERATIONS

With the MAT statement, you can perform the following operations with arrays:

1. Assignment
2. Addition
3. Subtraction
4. Multiplication
5. Transposition
6. Inverting and Finding the Determinant

Each MAT operation statement begins with the keyword MAT followed by an expression to be evaluated.

7.3.1 Matrix Assignment

You can assign the value of one array to another array as in the following example:

```
00010 MAT A=B
```

This statement sets each entry of array A equal to the corresponding entry of array B. A is redimensioned also automatically to the size of array B.

7.3.2 Matrix Addition and Subtraction

You can add and subtract arrays as shown in the following lines:

```
00010 MAT A=B+C  
00020 MAT D=B-C
```

The first statement assigns the sum of corresponding elements in arrays B and C to corresponding elements in array A. The second statement assigns the difference between arrays B and C to array D. B and C can be either lists or matrices; however, they both must have identical dimensions.

7.3.3 Matrix Multiplication

You can multiply two arrays as shown in the following line:

```
00010 MAT A=B*C
```

This statement causes array A to be set equal to the product of arrays B and C. A, B, and C must all be 2-dimensional arrays, and the number of columns in array B must be equal to the number of rows in array C. BASIC redimensions A to the number of rows in B and the number of columns in C.

USING ARRAYS

The following statements are invalid in BASIC:

```
00010 MAT A=A*A
00020 MAT A=A*B
00030 MAT A=B*A
```

Elements in array A are needed for the calculation of the expressions after they have already been destroyed. These invalid statements cause BASIC to print an error message.

However, this statement

```
00010 MAT C=A*A
```

is valid if A is a square matrix.

You can also perform scalar multiplication of a matrix:

```
00010 MAT A=(K)*B
```

where each entry in array B is multiplied by the value of K. K is any arithmetic expression and must be enclosed in parentheses. Array A is automatically redimensioned to array B if enough space is reserved.

7.3.4 Matrix Transposition

The TRN function transposes the dimensions of an array and renames it. A matrix with m rows and n columns will be renamed and redimensioned to n rows and m columns. The format of this function is as follows:

```
MAT array=TRN(array)
```

For example:

```
00010 DIM B(3,5)
00020 MAT READ B
00030 MAT A=TRN(B)
00040 DATA 1,2,3,4,5
00050 DATA 6,7,8,9,10
00060 DATA 11,12,13,14,15
00070 MAT PRINT B;
00080 MAT PRINT A;
```

READY

RUNNH

```
1 2 3 4 5
6 7 8 9 10
11 12 13 14 15
```

```
1 6 11
2 7 12
3 8 13
4 9 14
5 10 15
```

Note that the array name on the left of the equal sign must be different from the array name on the right of the equal sign: for example, MAT A=TRN(A) is invalid.

USING ARRAYS

7.3.5 Inverting and Finding the Determinant of a Matrix

If you want to find the determinant of a matrix, you must first find the inverse. Use the INV function for this purpose. The INV function is used as in the following example:

```
00010 MAT A=INV(B)
```

The INV function allows matrix A to be the inverse of matrix B. (B must be a square matrix.) BASIC redimensions A to be the same size as B.

NOTE

Although matrix inversion does not operate on the elements of row 0 and column 0 of a matrix, BASIC does store intermediate results in these elements of an inverse.

Therefore, the values of the elements in row 0 and column 0 of an inverse matrix may change.

The function DET is available after the inversion. You can then use DET as a variable set equal to the value of the determinant of B. Consequently, you can obtain the determinant of a matrix by inverting the matrix and then noting the value of DET. For example:

```
00010 MAT A = INV(X)\D1=DET
00020 MAT B = INV(A)\D2=DET
00030 IF D1 = D2 THEN PRINT "RELATIONSHIP TRUE"
00040 PRINT D1
```

NOTE

If you specify a list rather than a matrix, BASIC cannot complete the inversion. Therefore, DET is set equal to 0.

7.4 ARRAY INPUT AND OUTPUT

Elements in an array can be accessed with the statements MAT INPUT, MAT PRINT, and MAT READ.

USING ARRAYS

7.4.1 MAT INPUT Statement

The MAT INPUT statement reads the values you type at the terminal and enters the values for each element of a list or matrix.

The MAT INPUT statement has the following format:

```
MAT INPUT array(s)
```

where:

MAT INPUT must have a space between MAT and INPUT.

array can be one or several lists or matrices separated by commas.

BASIC reads data from the terminal the same way it does with the normal INPUT statement. The question mark signals that BASIC is ready to accept input.

Unlike the INPUT statement, however, the MAT INPUT statement allows you to enter a variable number of values into an array. You need not supply the same number of elements as are requested in the MAT INPUT statement; you can include fewer elements but not more than requested.

You can also continue typing data on more than one line by using the continuation character, the ampersand(&). In this case, you terminate the input with a line terminator.

The values you type are entered into successive array elements in row order starting with the first element. If you type a variable number of values, you can determine the number of rows and columns you filled by using the two variables NUM and NUM2.

If the array is a list, BASIC sets NUM equal to the number of elements you enter. If the array is a matrix, BASIC sets NUM to the number of rows you enter and NUM2 to the number of elements in the last row. By printing these variables, you can see the size of the array.

If you specify more than one array in the MAT INPUT statement, only the last array can have a variable number of elements. You can also redimension an array by specifying a new size in the MAT INPUT statement.

The following is an example of the MAT INPUT statement:

```
00010 DIM A(5)
00020 MAT INPUT A
READY

RUNNH
? 1,2,3,4,5
```

You cannot include a string constant within the MAT INPUT statement as you can in the INPUT statement. You can print the results of your input with the MAT PRINT statement.

USING ARRAYS

NOTE

The MAT INPUT # statement is used to enter list or matrix values from a terminal-format file.

7.4.2 MAT PRINT Statement

The MAT PRINT statement prints the specified array element(s) or entire array(s) at the terminal.

The MAT PRINT statement has the following format:

```
    MAT PRINT array(s)
```

where:

MAT PRINT	must have a space between the two words.
array(s)	without subscripts causes the printing of the entire array. Subscripted array name(s) cause the maximum size of the array (defined by the subscript) to be printed. (It does not redimension the array.)

If you follow the array with a semicolon, the data values print in a packed fashion. If you follow the array with a comma, the data values print across the line with one value per print zone. If neither character follows the array, each element prints on a separate line. All but the last array in a list must have a comma or a semicolon, separating it from the next array on the list.

Each row of a matrix starts printing on a new line. You can print one-dimensional arrays (lists) in either row or column format. For example:

```
00010 DIM A(5)
00020 MAT INPUT A
00030 MAT PRINT A
00040 MAT PRINT A,
00050 MAT PRINT A;
00060 END
```

```
READY
RUNNH
? 5
```

```
5
0
0
0
0
```

```
5 0 0 0 0
```

```
5 0 0 0 0
```

USING ARRAYS

Notice that only one value was typed in response to the MAT INPUT. The remaining elements retain a value of 0.

When you specify more than one array, BASIC begins printing each array starting on a new line. (BASIC never prints row 0 and column 0 when using the MAT PRINT statement.)

7.4.3 MAT READ Statement

The MAT READ statement reads the values into elements of a 1- or 2-dimensional array from DATA statements. The MAT READ statement has the following format:

```
MAT READ array(s)
```

where:

MAT READ must have a space between the two words.

array(s) without subscripts cause the entire array to be read. Arrays with subscripts cause the array to be redimensioned accordingly.

The maximum size of the redimensioned array cannot exceed the previous dimensions.

BASIC reads the values from DATA statements in the same manner as the READ statement. (Row 0 and column 0 are not read.) The DATA statement must contain enough data. You cannot input a number of elements that varies from the corresponding MAT READ statement.

Consider the following example:

```
00010 DIM B(2,2), C(2,2)
00020 MAT READ B
00030 MAT READ C
00040 MAT A=B+C
00050 MAT PRINT A;
00060 MAT PRINT B;
00070 MAT PRINT C;
00080 DATA 1,2,3,4,5,6,7,8
```

READY

RUNNH

```
6 8          Matrix A
```

```
10 12
```

```
1 2          Matrix B
```

```
3 4
```

```
5 6          Matrix C
```

```
7 8
```

CHAPTER 8

USING TERMINAL-FORMAT AND VIRTUAL-ARRAY FILES

There are three types of files in BASIC:

1. Terminal-Format files - files whose elements are stored in line; the elements are accessed sequentially.
2. Virtual-Array files - files whose elements are stored in arrays; the elements are stored randomly.
3. Record files - files whose elements are stored in records; the elements can be accessed sequentially, randomly, or by index.

Record files are described in Chapter 9. Terminal-format files and virtual-array files are described in this chapter in the following sections:

- 8.1 Terminal-Format Files
- 8.2 Virtual-Array Files

Section 8.3 describes topics common to all three file types: file renaming and deleting.

8.1 TERMINAL-FORMAT FILES

A terminal-format file is a collection of ASCII characters stored in lines of various lengths. The end of a line is determined by a line terminator, that is, a line feed. BASIC stores these ASCII characters, including spaces and line terminators, exactly as they would appear on the terminal; hence the name terminal-format file.

Terminal-format files are sequential-access files. Sequential-access files are those files that contain information that must be read or written one item after another from the beginning of the file. This means that you cannot retrieve an item from the file without first retrieving all the items preceding it.

BASIC has a file pointer that keeps track of where you are in the file. To add new items to an existing file without overwriting current information, you can either read the entire file or use a special access mode called APPEND. Both methods place the file pointer at the end of the file where you can then add data.

USING TERMINAL-FORMAT AND VIRTUAL-ARRAY FILES

8.1.1 Opening Terminal-Format Files

Before you can access a terminal-format file, you must open it. The OPEN statement allows you to open a new file, or an existing file, and associate the file with a file number.

The OPEN statement has the following format:

```
OPEN filename exp [FOR {INPUT } ] AS [FILE] [ # ] expression
                    [ ,ACCESS {READ
                               WRITE
                               MODIFY
                               SCRATCH
                               APPEND } ] [ ,ALLOW {NONE}
                                               {READ} ]
                    [ ,INVALID line no. ] [ ,LOCKED line no. ]
```

where:

filename exp	is a string expression representing a TOPS-20 file specification.
INPUT	specifies an existing file from which data can be taken.
OUTPUT	specifies creation of a new file to which data can be sent. If a file exists with the same file specification, the existing file is superseded.
expression	is the file number you want to associate with the files. It can be any real or integer expression.

INPUT and OUTPUT are both optional. If you omit this part of the statement from the OPEN statement, BASIC checks first for an existing file (INPUT). If no file exists with the name specified, BASIC creates a new file (OUTPUT). The ACCESS, ALLOW, INVALID, and LOCKED clauses are optional attributes that you can specify (in any order) when opening a file.

The ACCESS clause defines both the position of the file pointer and the operations you can perform:

READ	The file pointer is at the beginning of the file. You can only read the file. A READ operation can only be performed on an existing file.
WRITE	The file pointer is at the beginning of the file. You can only add data to the file.
MODIFY	This is the default. The file pointer is at the beginning of the file. You can read and write to the file.
SCRATCH	The file pointer is at the beginning of the file. You have complete access to the file; you can read, write, and truncate the file.

USING TERMINAL-FORMAT AND VIRTUAL-ARRAY FILES

APPEND The file pointer is at the end of the file. You can read and write to the file at this point. An APPEND operation is valid only for an existing file.

The ALLOW clause defines what you allow other users to do to the file while you are using it.

NONE No one can read or write data while you have the file open.

READ is the default. It allows others to read the file while you are using it.

You cannot specify an ALLOW clause if you use ACCESS SCRATCH.

The INVALID clause specifies the line number to which control passes if the operation is unsuccessful. This transfer takes effect only if the OPEN fails, that is, if you specified an illegal file specification. If you do not include INVALID or any active error handler, BASIC takes over the error handling.

The LOCKED clause also specifies the line number to which control passes if the operation is unsuccessful. This transfer takes effect only if the file is locked because another user specified ALLOW NONE. If you do not specify a LOCKED clause (or any active error handler), BASIC takes over the error handling.

Table 8-1 describes the results of specifying the keywords in ACCESS clause of the OPEN statement.

Table 8-1
ACCESS-Clause Keywords of the OPEN Statement

Access	Initial File Position	I/O Operation
READ	beginning	read only
WRITE	beginning	write only
MODIFY	beginning	read or write
SCRATCH	beginning	read, write, truncate
APPEND	end	read or write

Consider the following example:

```
00010 OPEN "DATA1" FOR INPUT AS FILE 1, ACCESS APPEND
00015 N=5
00020 OPEN "MONEY" FOR OUTPUT AS FILE N
00030 END
```

Line 10 opens an existing file at the end and associates it with file 1. Line 20 creates a new file specified by MONEY and associates it with file 5. If a file named MONEY already exists, BASIC supersedes it with the new request. When you open a file and associate a file number to it, you use that number when referencing the file, for example, DATA1 is 1, MONEY is 5.

USING TERMINAL-FORMAT AND VIRTUAL-ARRAY FILES

To save files for future use, you must close them.

8.1.2 Closing Terminal-Format Files

All programs that open files should close them before terminating execution. Most systems do not save files unless they are closed. An existing file with the same file specification as a new file will not be superseded until the new file is closed.

BASIC closes all files:

1. when executing a CHAIN statement
2. when executing an END statement
3. after executing the highest numbered line in the program

Note that BASIC does not automatically close files after executing a STOP statement.

A more specific way to close files is with the CLOSE # statement. The CLOSE # statement closes the files you specify and disassociates them from their file numbers. After you close a file, you cannot access it without reopening it.

Unlike the three methods listed above, the CLOSE # statement allows you to specify which files you want closed.

The CLOSE # statement has the following format:

```
CLOSE [ [# ] expression(s) ]
```

where:

expression(s) specifies one or more file numbers, separated by commas. The number sign, preceding each expression, is optional.

If no expressions are specified, BASIC closes all open files.

The following examples illustrate the CLOSE # statement:

```
00010 CLOSE #1 !CLOSES FILE ASSOCIATED WITH FILE 1
00020 B=4
00030 CLOSE 2,B,6+1 !CLOSES FILE NUMBERS 2, 4, 7
00040 CLOSE !CLOSES ALL FILES
```

8.1.3 Reading Data from a Terminal-Format File

The INPUT # statement reads data stored in a terminal-format file and assigns a value to each variable listed.

The INPUT # statement has the following format:

```
INPUT [ [# ] expression, [ prompt string { ; } ] ] variable(s)
```

USING TERMINAL-FORMAT AND VIRTUAL-ARRAY FILES

where:

# expression,	is the file number of the terminal-format file. If the value of the expression is zero, data is input from the terminal rather than a file. The comma is required. The number sign, preceding the file expression, is optional.
prompt string	is an optional quoted string constant which is printed only if input is being taken from the terminal. A semicolon placed after the prompt causes BASIC to accept input right after the prompt. A comma placed after the prompt causes BASIC to tab over to the first available print zone before accepting input.
variable(s)	is one or more variable names separated with commas.

The INPUT # statement acts very much as the INPUT statement described in Section 3.1.1. However, the INPUT # statement requests data from a terminal-format file rather than from the user's terminal. In order to INPUT # from a file, you must OPEN it for ACCESS READ, MODIFY, APPEND, or SCRATCH. If you OPEN the file with ACCESS APPEND, you must RESTORE # it before you can read it (See Section 8.1.5.).

Consider the following example:

```
READY
LISNH
00010 OPEN "DATA.B20" AS FILE 2
00020 INPUT #2,A$,B
00025 PRINT
00030 PRINT A$,B
00040 GOTO 20
00050 END
```

```
READY
RUNNH

SARAH          187.2

JOE            117.45

NORMA          200

MICHAEL        89

JANET          125

SARA           63.6
```

```
? 11 End of file found on INPUT at line 00020 of MAIN PROGRAM
```

```
READY
```

If this example had been written with an INPUT statement (rather than INPUT #), BASIC would have stopped and printed a question mark to request data from you. Instead, the INPUT # read the data into the program from a previously stored terminal-format file (DATA.B20).

USING TERMINAL-FORMAT AND VIRTUAL-ARRAY FILES

BASIC starts reading data from the beginning of the file. If the line of data in the file contains more data than there are variables in the INPUT # statement, BASIC prints the error message

```
% 436 Too much data present - ignored
```

and ignores the excess data. If, however, there is not enough data in the line, BASIC prints the error message

```
? 59 Insufficient data
```

and does not attempt to satisfy the INPUT # request for the remaining variable. An input list that is terminated by a comma indicates that subsequent INPUT # requests may read more data from the current data line. Additionally, the ampersand character (&) immediately followed by a line terminator indicates that the data line is continued, for all forms of the INPUT # statement. If you try to INPUT # from a new file or a file OPENed with either ACCESS WRITE or ACCESS APPEND (before a RESTORE # is done), BASIC prints an error message.

8.1.3.1 The INPUT LINE # and LINPUT # Statements - The INPUT LINE # and LINPUT # statements have the following format:

```
INPUT LINE [#]expression, [prompt string {;} ] variable(s)
```

```
LINPUT [#]expression, [prompt string {;} ] variable(s)
```

where:

# expression	is the file number of the file where the data resides. If the number is zero, BASIC inputs data from the terminal. The comma is required. The number sign, preceding the file expression, is optional.
prompt string	is a quoted string constant which is printed only if input is being taken from the terminal. A semicolon placed after the prompt causes BASIC to accept input right after the prompt. A comma placed after the prompt causes BASIC to tab over to the first available print zone before accepting input.
variable(s)	one or more string variables separated by commas.

The INPUT LINE # statement reads a string of characters from a terminal-format file into each respective string variable in the list. All characters on the input line including commas, quotation marks, and the line terminator are assigned to the string variable.

The LINPUT # statement also reads an entire line of data into each respective string variable in this list; however, it does not include the line terminator.

USING TERMINAL-FORMAT AND VIRTUAL-ARRAY FILES

The following example illustrates both statements:

```
00010 OPEN "TEST" AS FILE 1
00020 PRINT # 1, "DATA, FOR A PROGRAM."
00030 PRINT # 1, "SECOND LINE."
00035 RESTORE #1
00040 INPUT LINE # 1, A$
00050 PRINT A$
00060 LINPUT # 1, A$
00070 PRINT A$
00080 END
RUNNH
```

```
DATA, FOR A PROGRAM.
```

```
SECOND LINE.
```

```
READY
```

Line 10 opens a file named TEST and associates it with FILE 1. Lines 20 and 30 write data into the file. Line 35 restores the file pointer to the beginning of the file. The INPUT LINE # statement requests a line of data from the file. BASIC reads the entire line, including the line terminator, into the program. If an INPUT # statement had been used, BASIC would have read only "DATA" into the string variable A\$.

The LINPUT # statement on line 60 requests another line of data. This time BASIC reads all characters into the program except for the line terminator.

8.1.4 Writing to a Terminal-Format File

The PRINT # statement writes data into the specified terminal-format file. The PRINT # statement has the following format:

```
PRINT[#] expression, list
```

where:

expression is the file number of the terminal-format file. If the value of the expression is zero, BASIC prints the data on the terminal. The number sign, preceding each file expression, is optional.

list contains the items you want printed. The items can be any real, integer, or string expressions. Separate the items with commas or semicolons. The resulting output format is the same as the simple PRINT statement.

If there are no items in the list, BASIC sends a blank line to the file. To PRINT # to a file, you must OPEN with ACCESS WRITE, MODIFY, SCRATCH or APPEND.

The PRINT # expression USING statement prints formatted data to a file (See Chapter 10.).

USING TERMINAL-FORMAT AND VIRTUAL-ARRAY FILES

Consider the following example that writes to a terminal-format file from data stored in DATA statements:

```
00010 OPEN "NAMES" FOR OUTPUT AS FILE 1
00020 READ A#, A !READ DATA FROM PROGRAM
00030 IF A#="" THEN 90 !CHECK FOR LAST ITEM
00040 PRINT # 1, A#; ", "; A !PRINT TWO ITEMS
00050 GOTO 20
00060 DATA "SARAH", 187.2, "JOE", 117.45
00070 DATA "JANE", 200, "JIM", 89
00080 DATA "", 0
00090 CLOSE # 1
00100 END
RUNNH
```

READY

After you run this program, the file NAMES contains the following:

```
SARAH,    187.2
JOE,      117.45
JANE,     200
JIM,      89
```

Note that the commas between data items are not automatically generated by the PRINT statement. If you want to read individual data items back in with the INPUT statement, the commas must be there to separate the items.

NOTE

The MAT INPUT # statement reads array elements from a terminal-format file. It allows a variable number of data items to be read from one data record. The MAT READ # statement also reads array elements from a terminal-format file. With MAT READ #, however, all elements of the array must be filled by data items from one data record.

The MAT PRINT # statement outputs all elements in an array to a terminal-format file. (The MAT statements are described in Chapter 7.)

8.1.5 Restoring a Terminal-Format File

The RESTORE # statement resets the file pointer of the specified terminal-format file to the beginning of the file.

The RESTORE # statement has the following format:

```
RESTORE[#]expression
```

where:

expression is the file number of the terminal-format file. The number sign, preceding the file expression, is optional.

USING TERMINAL-FORMAT AND VIRTUAL-ARRAY FILES

After writing into a file, you can bring the file pointer back to the beginning with the RESTORE # statement.

```
00010 OPEN "NEW" AS FILE 1
00020 PRINT#1, 65;"",";80;"",";95
00030 RESTORE#1
00040 INPUT#1, A, B, C
00050 PRINT A, B, C
00060 CLOSE#1
00070 END
RUNNH

65 80 95
```

8.1.6 Truncating a Terminal-Format File

The SCRATCH # statement truncates the specified terminal-format file at the current position of the file.

The SCRATCH # statement has the following format:

```
SCRATCH[#]expression
```

where:

expression is the file number of the terminal-format file. The number sign, preceding the file expression, is optional.

You may truncate a file at any position. The file, however, must first be opened with ACCESS SCRATCH.

8.1.7 Checking for the End of a Terminal-Format File

To test for the end of a terminal-format file, you can use the IFEND #, IFMORE #, or a NODATA # statement.

8.1.7.1 IFEND # Statement - The IFEND statement has the following format:

```
IFEND[#]expression { THEN statement }
                    { THEN line number }
                    { GOTO line number }
```

where:

expression is the file number of a terminal-format file. The number sign, preceding the file expression, is optional.

statement is any BASIC statement.

line number is any line number in the program.

With the IFEND # statement you check for the end of the file. If the file pointer is at the end, you can transfer control to another line of the program or execute a statement.

USING TERMINAL-FORMAT AND VIRTUAL-ARRAY FILES

8.1.7.2 **IFMORE # Statement** - The IFMORE statement has the following format:

```
IFMORE[#]expression { THEN statement
                     { THEN line number
                     { GOTO line number }
```

where:

expression is a file number of a terminal-format file. The number sign, preceding the file expression, is optional.

statement is any valid BASIC statement.

line number is any line number in the program.

IFMORE tests whether the file pointer is at the end of the file specified. If not at the end, BASIC executes the statement or goes to the line number specified.

8.1.7.3 **NODATA # Statement** - The NODATA # statement has the following format:

```
NODATA[#]expression, ]line number
```

where:

expression is a terminal-format file number. The number sign, preceding the file expression, is optional.

line number is any valid line number in the program.

When you specify NODATA #, such as:

```
NODATA #6, 45
```

BASIC checks for the end of the file, that is, no data. If at the end of the file, BASIC transfers control to the specified line number. If no file number is specified, BASIC transfers control to the specified line only if the data pointer is at the end of all DATA-statement data for this program segment.

For example:

```
NODATA 100
```

8.1.8 Changing Margins in a Terminal-Format File

The MARGIN # statement allows you to modify the margin setting of a terminal-format file or the margin setting of your terminal. The MARGIN # statement has the following format:

```
MARGIN[#]expression, ]num exp
```

USING TERMINAL-FORMAT AND VIRTUAL-ARRAY FILES

where:

expression, is the file number of an open terminal-format file. If you do not specify this argument (or you specify file number 0), your terminal margin is changed. The number sign, preceding the file expression, is optional.

num exp is the numeric expression that determines the margin. (If it is a real number, it is truncated.)

The default margin for a terminal-format file is 72; the default margin width for a terminal is the current terminal width.

Consider these examples:

```
00010 MARGIN 5
00020 PRINT "#"; FOR I = 1 TO 10
00030 END
RUNNH
#####
#####
READY
```

This example changes the terminal width to 5. To change it back to the default, type:

```
00010 MARGIN 0
```

The following example changes the margin of the terminal-format file number 4 to 132 columns:

```
00010 MARGIN #4, 132
```

8.1.9 Setting Page Size in a Terminal-Format File

Normally, output to a terminal-format file and to a terminal are not divided into pages. The PAGE # statement allows you to set a page size of any positive number of lines.

The PAGE # statement has the following format:

```
PAGE [#]expression, ]num exp
```

where:

expression is the file number of an open terminal-format file. If the file number is 0 or is omitted, the PAGE # setting affects the user's terminal. The number sign, preceding the file expression, is optional.

num exp is any numeric expression. It is truncated before the page size is set. It represents the number of lines per page.

USING TERMINAL-FORMAT AND VIRTUAL-ARRAY FILES

The page size remains in effect until:

1. The page size is set again with the PAGE statement. A page size of 0 disables paging.
2. Execution ends.
3. The file is closed.

At the end of program execution, the terminal page size is reset to what it was before program entry.

When a PAGE # statement is executed, BASIC ends the current output line (if necessary), outputs a form feed, and starts counting lines beginning with the next line of output. As soon as a new page is necessary, a form feed is output.

8.2 VIRTUAL-ARRAY FILES

A virtual-array file, like a terminal-format file, is information stored on a system device (disk). Once you open a virtual-array file, the similarity with terminal-format files ends. There is no need for the INPUT #, INPUT LINE #, LINPUT # PRINT #, or RESTORE # statements with virtual-array files. You access elements in a virtual-array file exactly as you access elements in an array in memory. (See Sections 1.8 and 1.8.1.) In fact, you can use virtual arrays just as you would regular arrays.

Virtual-array files are random-access files. You can read or write any element in the file no matter where it is located. The last element in a virtual array can be accessed as quickly as the first. Contrast this with a terminal-format file where you must read the entire file to get to the last element.

When BASIC stores data in a virtual-array file, it does not convert them to ASCII characters but rather stores them in the internal binary representation. Consequently, there is no loss of precision caused by data conversion.

You define storage space for a virtual-array file just as you do for a regular array. The DIM # statement (Section 8.2.1) allows you to set parameters for the file. Unlike when you specify arrays in memory, you must specify the maximum character length of strings in a virtual-array file. Strings longer than the maximum are truncated. Strings shorter than the maximum are padded with trailing nulls.

8.2.1 Dimensioning a Virtual-Array File

To use a virtual-array file, you must first define its size with a DIM # statement. The DIM # statement has the following format:

$$\text{DIM} \left[\# \right] \text{expression, array(s)} \left[=\text{number} \right]$$

where:

expression, is the file number associated with the virtual-array file. The number sign, preceding the file expression, is optional.

USING TERMINAL-FORMAT AND VIRTUAL-ARRAY FILES

array(s) is one or more 1- or 2-dimensional arrays separated by commas. These arrays are within the virtual-array file.

=number is the maximum length of a string array if any are specified. The default is 16 characters.

For example:

```
00010 DIM #2, A(15,20), B(50), C$(18)=10
```

The DIM # statement establishes the number of subscripts allowed for each virtual array, and the maximum values for each. In addition, the DIM # statement allocates all space for the virtual arrays associated with a particular file number. Storage allocation always starts at the beginning of the file. Therefore:

```
00100 DIM #1, A(100), B(100)
```

and

```
00010 DIM #1, A(100)
```

```
00020 DIM #1, B(100)
```

do not perform the same function. Line 100 allocates 202 elements (101 elements in each array) on file number 1. Lines 10 and 20 allocate only 101 elements on file number 2, because the DIM # statement in line 20 causes storage allocation to start at the beginning of the file. Thus, file 2 contains 101 elements that have two array names. For example, A(97) and B(97) would reference the same element.

When you specify a virtual array of strings, you should indicate the maximum length of each string. If no maximum is specified, the default is 16.

To correctly access the data in an existing virtual-array file, ensure that the DIM # statement specifies the same data type and subscript as in the program which created the file. The variable name associated with the file can be different from the original as long as the data type is the same.

8.2.2 Opening and Closing Virtual-Array Files

To open a virtual-array file, you use the OPEN statement with a few variations. The virtual-array OPEN statement has the following format:

```
OPEN filename exp [FOR {INPUT } ] AS [FILE][#] expression
, [ORGANIZATION] VIRTUAL [ ,ACCESS {READ }
{MODIFY }
{WRITE } ]
[ ,ALLOW {NONE } ] [ ,INVALID line no. ]
{READ }
[ ,LOCKED line no. ]
```


USING TERMINAL-FORMAT AND VIRTUAL-ARRAY FILES

where:

filename exp	is a string expression which is a TOPS-20 file specification.
FOR INPUT	specifies an existing file from which data can be taken.
FOR OUTPUT	specifies the creation of a new file to which data can be sent.
expression	is the file number. It can be any real or integer expression.
ORGANIZATION VIRTUAL	specifies a virtual array file. The keyword ORGANIZATION is optional.
ACCESS READ	allows read only.
ACCESS WRITE	allows write only.
ACCESS MODIFY	allows read and write operations. This is the default.
ALLOW NONE	allows no simultaneous access.
ALLOW READ	allows others to read while you have the file. This is the default.
INVALID line no.	specifies a line in the program to which control passes if the OPEN statement fails.
LOCKED line no.	specifies a line in the program to which control passes if the OPEN statement fails because the file is locked.

The ORGANIZATION clause defines the file to be a virtual-array file as opposed to a terminal-format file. (The ORGANIZATION clause is not allowed in a terminal-format OPEN statement.) The rest of the clauses may be specified in any order.

Consider these two program lines:

```
00010 DIM #2, F(100,10)
00020 OPEN "VARAY" FOR OUTPUT AS FILE 2, VIRTUAL
```

These lines open a virtual-array file as file 2 and allocate 1,111 elements of storage space, that is, 100 x 10 plus the 0 elements.

As an example of a use of virtual-array files, consider the problem of an information retrieval system for a small organization. Assume there are 1000 employees each needing a 255-character record containing the name, home address, home phone, work station and phone extension. If this information is maintained in a terminal-format file, it would take a long time to locate the information for any employee and it would be impossible to update. Alternatively, these records can be maintained in a virtual-array file. In this case some index is needed to associate a particular employee with a record.

USING TERMINAL-FORMAT AND VIRTUAL-ARRAY FILES

In the following example, an index file containing badge numbers is used to find the record in the master file. The employee's badge number is in the same position in the index file as the record is in the master file. It is much faster to search through the index file because the data elements are much shorter, and less time is spent reading data from the file. This example program prints the employee's name based on the badge number.

```
00010 DIM #1, B%(1000)           !1000 ELEMENTS IN BADGE
                                !NUMBER FILE.
00020 DIM #2, B$(1000)=255       !1000 ELEMENTS IN MASTER
                                !FILE.
00030 OPEN "BADGE" AS FILE 1, VIRTUAL !OPEN BADGE FILE.
00040 OPEN "MASTER" AS FILE 2, VIRTUAL !OPEN MASTER FILE.
00050 PRINT "WHAT IS THE BADGE NUMBER"; !REQUEST A BADGE NUMBER.
00060 INPUT N
00070 FOR I%=1 TO 1000           !SEARCH FOR A MATCH IN
                                !THE BADGE NUMBER
00080 IF B%(I%)=N THEN 200       !FILE
00090 NEXT I%
00100 PRINT "NO SUCH EMPLOYEE"   !CAN NOT FIND MATCH,
00110 GO TO 32700                !PRINT MESSAGE AND
00200 PRINT "NAME IS"; SEG$(B$(I),10,30) !TERMINATE. AFTER
32700 CLOSE #1,#2               !A MATCH IS FOUND, GET
32767 END                        !RECORD OF EMPLOYEE,
                                !B%(I%), AND EXTRACT THE
                                !NAME FROM THE RECORD
                                !(THE NAME IS STORED
                                !FROM THE 10TH TO THE
                                !30TH CHARACTERS, THEN
                                !TERMINATE.
```

To close a virtual-array file, use the CLOSE statement described in Section 9.5.2.

8.3 FILE RENAMING AND DELETING

The following sections describe the process of renaming and deleting terminal-format, virtual-array, and record files from storage.

8.3.1 The NAME-AS Statement

The NAME-AS statement has the following format:

```
NAME string 1 AS string 2
```

where:

string 1 is a string expression which is the file specification of the file to be renamed.

string 2 is a string expression which is the new file specification.

For example:

```
00010 NAME "MONEY" AS "ACCNTS"
```

This statement changes the file named MONEY to ACCNTS.

USING TERMINAL-FORMAT AND VIRTUAL-ARRAY FILES

The NAME-AS statement does not alter the contents of the file. It renames the first file specified to that of the second file without changing the file number. If you use the NAME-AS statement on an open file, the new name does not take effect until the file is closed.

8.3.2 The KILL Statement

The KILL statement has the following format:

```
KILL string expression
```

where:

```
string expression is the file specification of the file you  
want deleted from storage.
```

After you delete a file, you cannot open it or access it in any way. For example:

```
00010 KILL "DATA"
```

deletes the file DATA from storage.

NOTE

If you want to undelete a file that has been deleted by the KILL statement, use the TOPS-20 UNDELETE command. (See the TOPS-20 Commands Reference Manual.)

CHAPTER 9

USING RECORD FILES

In addition to terminal-format and virtual-array files, BASIC provides another type of file for storing information, the record file. A BASIC record file is a collection of related data stored in the form of records. You determine the size and content of the records and the structure and access properties of the file.

Programs can write records into a file and subsequently retrieve them. Each record is treated as a separate unit. All input and output is performed on a record-by-record basis via a buffer between the file and the program.

In order to write BASIC programs that deal with record files, refer to the following sections:

- 9.1 File Organization
- 9.2 Access Methods
- 9.3 Record Formats
- 9.4 Record Mapping
- 9.5 File Operations
- 9.6 Record Operations
- 9.7 Dynamic Mapping of an I/O Buffer
- 9.8 Examples

Section 8.3 describes renaming and deleting files.

9.1 FILE ORGANIZATION

The manner in which BASIC stores and retrieves records in a file is determined by the structure, or organization, of the file. When you create the file, you specify its organization. The organization in turn determines the operations and access methods that you can use on the file. The three organizations you can specify for a record file are:

1. Sequential
2. Relative
3. Indexed

The organization you specify when the file is created is permanently assigned to the file. When an existing file is opened for processing, you must again specify the organization. An organization that does not match the file's initial organization results in an error.

USING RECORD FILES

9.1.1 Sequential Organization

A record file with sequential organization contains records that are stored in series. The order in which the records occur in the file is always the order in which they are written to the file. To read a particular record in the file, for example the fifteenth record, a program must open the file and successfully read the first 14 records before accessing the desired record.

Consequently, records can be added only to the end of a sequential file because the location of each record is fixed in relation to the record preceding and succeeding it. Sequential files are allowed on disk.

9.1.2 Relative Organization

A record file with relative organization contains records that are stored in numbered locations. BASIC structures the file into a series of record positions with each position capable of containing a single record. The number associated with a position represents its location relative to the beginning of the file. Thus, record number 1 occupies the first record position, record number 2 occupies the second record position, and so forth.

Access to a record can be made sequentially or randomly by record number. Relative files are allowed only on disk.

9.1.3 Indexed Organization

A record file with indexed organization contains records that are stored according to a field, called a key field, within each record. BASIC sets part of the file aside as an index, or key, in order to locate these records. Records are retrieved by a particular key depending on the contents of the key field in each record.

Indexed files require that at least one key, called the primary key, be associated with every record. In addition to a primary key, you can optionally define up to 254 alternate keys for a record. Alternate keys represent secondary data fields and are defined in the same manner as a primary key. Your program can also use these alternate keys to identify and retrieve records.

When you create an indexed file, you must specify which field in the record is to be used as the primary key. Access to a record can be made sequentially or randomly by reference to the primary key. Indexed files are allowed only on disk.

USING RECORD FILES

9.2 ACCESS METHODS

The methods that you use to store or retrieve records in a file are determined by the file's organization. The organization of a file is fixed at the time you create it, but, depending on the access allowed, an access method can change each time the file is opened. In some cases, you can vary the access from record to record during program execution. BASIC allows you to specify one of two access methods:

1. Sequential
2. Random

Sequential access indicates that records are accessed in serial order. Random access indicates that records are accessed by record number or key.

Table 9-1 shows the relationship between file organization and access methods.

Table 9-1
Record-File Access Methods

File Organization	Access Methods	
	Sequential	Random
Sequential	yes	no
Relative	yes	yes
Indexed	yes	yes

The following sections describe each type of record access.

9.2.1 Sequential Access

All file organizations allow you to access records sequentially. Sequential record access is employed when you issue a series of requests for the next record. The record operations are performed in terms of a predecessor-successor record relationship. For each successfully accessed record (except the last) there is a succeeding record somewhere in the file.

9.2.1.1 Sequential Access to Sequential Files - Sequentially organized files allow only sequential access. In these files, the predecessor-successor relationship is physical (that is, each record, except the last, is physically adjacent to the next record). Sequential access to a sequential file means that records are accessed in the order of their insertion into the file. A record can be processed only after each preceding record has been successfully accessed. Similarly, once a record is processed, the program must be repositioned to the beginning of the file before preceding records can be accessed. A RESTORE operation, or re-opening the file, positions the record pointer at the beginning of the file.

USING RECORD FILES

For sequential files, a PUT operation requires that the current record pointer be positioned at the end of the file. A FIND operation moves the pointer to the next sequential record position. Therefore, a series of FIND operations can be used to locate the end of the file (an unsuccessful FIND usually indicates end-of-file). A GET causes the program to locate the next record and perform a read operation. A succeeding GET or FIND operation moves the pointer to the next record. An UPDATE operation must be preceded by a FIND or a GET, which moves the pointer to the record to be modified.

9.2.1.2 Sequential Access to Relative Files - Relative file organization allows sequential access as established by the contents of record positions. Sequential access to a relative file means that records are accessed in ascending order according to record numbers. Relative files allow empty record positions that can be caused by a record deletion or by a program that purposely leaves the position empty. BASIC's record I/O maintains the predecessor-successor relationship through its ability to recognize empty or occupied record positions.

A sequential PUT operation to a relative file causes a record to be placed in a location whose position number is one higher than the previous operation. If that position is occupied, the operation fails.

9.2.1.3 Sequential Access to Indexed Files - Indexed file organization also supports sequential access. In indexed files, the predecessor-successor relationship exists among the entries in the index. Sequential access to an indexed file means that records are accessed by their key field according to the ASCII collating sequence. The search is made through a specified index in serial fashion. The records are retrieved in the same order that key values appear in the index.

PUT operations on indexed files write the record and place its key value in the appropriate index. On GET operations, the first record in the collating sequence (according to the appropriate index) is made available to the program. The next GET updates the pointer to the record whose key appears next in that index, and accesses the record. FIND operations perform in the same manner but without reading the record. UPDATE and DELETE operations require a successful GET or FIND.

9.2.2 Random Access

Random access allows the BASIC program rather than the file organization to control the order of record access. The predecessor-successor relationship has no effect on random access. The program identifies each record of interest in each operation. This procedure allows you to access records in any order at any point in the file.

Random access is not permitted on sequential files because of the strict physical relationship maintained among records. Relative and indexed files do allow random access.

USING RECORD FILES

The capability to shift from random to sequential access (or vice versa) is only allowed on relative and indexed files. Sequential files do not support random access.

Relative and indexed file organizations impose their own restrictions on the sequence of operations. For example, a GET operation always shifts the current file position to the target record. If you follow a series of sequential GET operations with a random PUT, the current file position remains at the location of the last GET. A sequential GET after the random PUT will resume at the point of the previous GET operation.

9.2.2.1 Random Access to Relative Files - Programs employ random access to relative files through the specification of a particular record number. BASIC record I/O interprets the number as representing a record position in the file. If the operation is a GET or a FIND and no record exists in the specified location, you will receive an error. If the operation is a PUT and a record already exists in that location, you will also receive an error.

Note that random access DELETE and UPDATE operations do not allow the specification of a record number. Also note that you must do a successful GET or FIND before performing these operations.

9.2.2.2 Random Access to Indexed Files - Programs initiate random access on indexed files by means of a key specification. You specify a key-of-reference number and key value in a manner determined by the desired operation. The specified key value indicates the contents of a record data field, and the key-of-reference number identifies the index used to locate that record.

On GET and FIND operations, you must specify the content of the desired key field. BASIC's Record Management Services (RMS) automatically searches the appropriate index, finds the desired key value (if present), locates the record in the file, and passes the record to the program.

A PUT operation does not allow an explicit key specification. Instead, the BASIC program provides RMS with a new data record. RMS retrieves the primary-key definition in the file, extracts the new record key value, and finally places the record into the file according to the value of the primary key.

Indexed files allow you to specify one of three types of key values for record retrieval. These are: exact key, approximate key, and generic key. You specify an exact key by indicating that the desired record's key field must be equal to the specified key. You specify an approximate key in your program by indicating that the desired record's key field can be equal to or greater than, or simply greater than the specified key. You specify a generic key in the program by indicating an initial portion of a key field. These three methods are described in Section 9.6.3.

USING RECORD FILES

9.3 RECORD FORMATS

In addition to specifying record organization and access for record files when you create a file, you must also specify the record format. Record format determines the general structure of each record in the file. The format is specified when the file is created and is permanently assigned to each record placed into that file.

BASIC allows you to specify one of three formats:

Fixed	The file contains records of equal and fixed length.
Variable	The file may contain records of different lengths.
Stream	The file contains a series of contiguous ASCII characters. A record is defined as a set of characters delimited by a line terminator.

The file organization determines which of the formats you can select. Table 9-2 shows the relationship between file organization and record formats.

Table 9-2
Record Formats

File Organization	Fixed-Length Records	Variable-Length Records	Stream-Format Records
Sequential	yes	yes	yes
Relative	yes	yes	no
Indexed	yes	yes	no

You must specify record format when you create a file. In a BASIC program, the record format is specified as an element of the file organization.

Stream format is supported only for sequential files on disk devices. A stream specification creates a file containing ASCII stream data. If you attempt to create a stream file on a non-disk device, an error is returned.

Variable format is the default for all three file organizations. The maximum record length is indicated by a count field or reference to a MAP.

USING RECORD FILES

9.3.1 Fixed-Length Records

Fixed-length describes an attribute of a file in which records are of equal and nonvarying length. Under fixed-length format, each record in a file occupies an identical amount of space.

In the BASIC program, you specify the length of records when the file is created. The length can be explicitly stated in the RECORDSIZE clause or implicitly defined by a map reference in the MAP clause. When a program requests a record from the file, the desired record is passed to the program within the length restrictions defined for that file. The record length you specify in your program is dependent on the data stored in the records. See Section 9.4 for a description of how to determine the length of a record.

Fixed-length format is optional for sequential, relative, and indexed files. However, relative files store records in fixed-length positions, regardless of the format specification. Relative file records are stored in positions that are each equal to the maximum record size specified when the file is created. This condition is true whether the format is fixed or variable.

9.3.2 Variable-Length Records

Variable-length format describes an attribute of a file in which the length of each record can differ. Variable-length format is the default for sequential, relative, and indexed file organizations.

When you use variable-length format, you must specify the length of the longest record in the file with the RECORDSIZE clause or the MAP clause. The length is dependent on the data stored in the record. See Section 9.4 for a description of how to determine record size.

Because record operations require that the record size be known, a count field is prefixed to each record as it is written to the file. The count field identifies the individual record size in characters, but this is transparent to the BASIC program.

Relative files are an exception in that variable-record format is allowed but record-position length is fixed. The length of each record position is defined by the size of the largest record. However, the size of each variable-format record may vary up to the size of this position. Thus, a record operation causes a transfer of only as many characters as the record contains, not the entire record position.

9.3.3 Stream-Format Records

Stream format describes an attribute of a file that contains a contiguous group of ASCII characters. A record in such a file is a set of characters delimited by one of the following:

1. Line feed (LF)
2. Form feed (FF)
3. Vertical tab (VT)

USING RECORD FILES

Stream format is allowed only for sequential files. To create a sequential file with stream format, you specify the STREAM attribute in the SEQUENTIAL clause, as follows:

```
OPEN file name exp FOR OUTPUT AS FILE # expression
, SEQUENTIAL STREAM, RECORDSIZE 80
```

Stream-format records are usually of variable lengths; however, no count field precedes the record. During output operations (PUT), RMS moves as many characters from the program buffer to the file as were specified in the COUNT or MAP specifier in the PUT statement. No checking is made of the value of the data in the record.

For input operations, RMS scans the record for the first occurrence of a LF, FF, or VT. If the scan is terminated by a LF, FF, or VT, RMS passes the entire string (including the terminator) to the program. Each successive input operation causes the scan to resume at the character following the last LF, FF, or VT encountered.

Do not PUT a record whose last character is not LF, FF, or VT. If you do a subsequent GET of that record, it will access all characters in the file up to the next LF, FF, or VT.

9.4 RECORD MAPPING

To access records in a file, you must establish a buffer for input and output. You can establish and name the buffer and describe the characteristics of the records in a particular file with the MAP statement. The MAP statement specifies that certain variables are contained in the buffer. The MAP clause in an OPEN statement, on the other hand, references a MAP statement and associates it with a particular file.

The MAP statement has the following format:

```
MAP (name) [ {ALIGNED } ] element(s)
            [ {UNALIGNED} ]
```

where:

(name) is the name you give to the buffer. Parentheses are optional.

ALIGNED is optional. These keywords refer to the way data is placed in the buffer. ALIGNED is the permanent setting; UNALIGNED is not currently supported and will be ignored.

element(s) is a list of elements, separated by commas, defining the characteristics of the record. Each element represents a field in the record.

A valid element in a MAP statement can be any numeric, integer, or string variable, an entire array, or a FILL. (FILL, FILL%, FILL\$, FILL\$=n, FILL(n), FILL\$(n)=m).

FILL acts as a space holder allowing you to mask parts of a record or to hold space for future use. FILL\$=m is a string of m characters and FILL\$(n)=m is n strings of m characters.

USING RECORD FILES

The length of a string variable is specified by the syntax:

A\$=n

where:

A\$ is the string variable.

n is the number of characters in the string. n must be a constant.

For example:

```
00025 MAP (BUFF1) NAME$=25, SS%, FILL, AGE%
```

This statement sets up a buffer area named BUFF1 and describes four data fields:

1. A string field containing up to 25 characters (NAME\$=25)
2. An integer field (SS%)
3. A floating-point place holder (FILL)
4. Another integer field (AGE%)

When specifying a string field, you should define the number of characters in the field. The default is 16 characters. Strings in a string field are fixed length. BASIC stores them left-justified and padded with blanks. For example:

```
00010 MAP (TEST) B$ = 7
00020 OPEN "FILE" AS FILE 1, SEQUENTIAL, ACCESS APPEND, MAP TEST
00030 B$='ABC'
00040 CZ=LEN(B$)
00045 PRINT CZ
00050 PUT #1
00060 CLOSE #1
00070 END

READY
RUNNH
7
```

Although the value assigned to B\$ is three characters long, the length of the string field contains those three characters plus four trailing blanks. Its length is seven.

NOTE

Variables that are parameters to subprograms are all passed by reference. Therefore, if an actual parameter is a variable that is MAPed, execution of a GET inside the subprogram will change the value of the dummy parameter.

The format of the data within a record is:

1. String data is stored in 7-bit ASCII, five characters per word.

USING RECORD FILES

2. Elements of string arrays and successive string elements in a given MAP are packed together.
3. Integer and real values are always word-aligned. Each occupies one word.
4. There may be unused character positions between a string value and the numeric value that follows it. The contents of these character positions is unspecified.

For example:

```
MAP FILE1MAP A$=2, B$=2, C
```

specifies that A\$ is stored in the first two characters of the first word of the record. The next two characters in that word contain B\$. Because numeric values are always word-aligned, character position 5 in that word is not used. C occupies all five characters in the second word of the record.

The RECORDSIZE clause of an OPEN statement should always be specified as a character count. If a record contains real or integer data, this count must include:

1. The number of characters contained in string items in the record
2. Five characters per real or integer item in the file
3. Any unused characters between a string item and the numeric item that follows it

Because the MAP statement defines the data content of the record, it also defines the position and length of indexed-file keys. Both primary and alternate key clauses in an indexed file OPEN statement refer to elements in a MAP statement.

A MAP statement must appear in your program before the OPEN statement that references it and before any references to variables it defines.

Because the MAP statement defines the amount of data in the record, it can also be used in the OPEN statement to define record size. In addition, a map reference and a RECORDSIZE specification can both appear in the same OPEN statement. This enables you to specify a smaller record (buffer) for a particular operation when the record-length format is variable. When both a map reference and a RECORDSIZE specification are used, the RECORDSIZE takes precedence.

NOTE

Because a RECORDSIZE specification overrides a MAP, it is possible to define a record size that overwrites mapped areas. You should never specify a RECORDSIZE that is larger than previously defined MAP statements for the same file.

USING RECORD FILES

A file may contain several different types of records, each composed of different data fields. A BASIC program can manipulate such a file if you specify more than one MAP statement to describe the corresponding buffer. Each such MAP statement, in this case, has the same MAPname.

If a particular variable is contained in records of more than one format, that variable can be declared in each corresponding MAP statement. For example:

```
LISNH
00010 MAP INVENTORY.RECORD PART.NUM#=6,TOTAL.STOCK,STOCK.IN.A,STOCK.IN.B
00020 MAP INVENTORY.RECORD PART.NUM#=6,TOTAL.STOCK
00030 OPEN "INVENTORY" AS $1, SEQUENTIAL,MAP INVENTORY.RECORD
```

The file "INVENTORY" contains some records that have a 6-character part number, a count of the total number in stock, and a breakdown of those available at locations A and B. Other records only contain a 6-character part number and a count of the total number in stock.

Note that if a given variable appears in more than one MAP statement, that variable must appear at the same location in the record in both declarations. Also, if it is a string variable, the string lengths must be the same.

PUT and UPDATE operations allow a MAP line reference in the statement. This allows you to specify the length of a record when there are multiple MAP declarations.

The following rules apply to MAP statements in a BASIC program:

1. All MAP statements must appear before the OPEN statement in a program and before their variables are referenced.
2. You can have multiple MAP statements with the same name. The largest buffer (longest element list) must be specified first. The first MAP statement sets up storage allocation.
3. The same variable can appear on the element lists of different MAP statements with the same MAP name. In this case, the variable must occur in the same position in each MAP statement. For example:

```
00100 MAP (BUFF1) A,B,C
00200 MAP (BUFF1) A,Q,C
```

4. If you specify an array in a MAP statement, you must dimension it in that statement. If the same array occurs in two different MAP statements for the same buffer, the dimensions must be the same.
5. The length of a string field should be defined; otherwise, the default is 16 characters.
6. MAP statements are local to the MAIN program or subprogram in which they are defined.

The MAP statement is referenced in the OPEN statement when you create a new file or access an existing file.

USING RECORD FILES

9.5 FILE OPERATIONS

When dealing with record files, you are either working with the file as a whole or working with an individual record in the file. The following sections describe how to:

1. Create a new file - OPEN statement
2. Access an existing file - OPEN statement
3. Close a file - CLOSE statement
4. Return the file pointer to the beginning - RESTORE statement
5. Truncate an entire file - SCRATCH statement

Section 9.6 describes working with individual records in a file.

9.5.1 Creating and Accessing a File

The OPEN statement enables you to create a new file or access an existing file. With this statement you can define, explicitly, all the important aspects of each data transfer operation including the structure of the file and its file-sharing capabilities. You can also provide for transfer of control if the OPEN statement fails.

The syntax of the OPEN statement includes keywords that describe attributes of the file. These keywords are followed, in general, by a name, numeric expression, or line number, and are separated by commas.

The following is the general syntax of the OPEN statement for a record file:

```
OPEN filename exp [FOR {INPUT } ] AS [FILE] [#] expression
                    {OUTPUT}

,[ORGANIZATION] {SEQUENTIAL} [ {FIXED
                    {RELATIVE}   {VARIABLE}
                    {INDEXED}    {STREAM} ]

[ ,ACCESS {READ
          {WRITE
          {MODIFY
          {SCRATCH
          {APPEND } ] [ ,ALLOW {NONE
                              {READ
                              {WRITE
                              {MODIFY } ]

          {,MAP mapname
          {,RECORDSIZE num exp}

          [ ,INVALID line no. ]

          [ ,LOCKED line no. ]

          [ {,DOUBLEBUF
          {,BUFFER [#] num exp} ]

[ {,SPAN } ] [ ,BLOCKSIZE num exp ]
 {,NOSPAN}
```

USING RECORD FILES

```
[,BUCKETSIZE num exp] [,CLUSTERSIZE num exp]
[,MODE num exp]           [,DENSITY num exp]
[,PRIMARY [KEY] name]     [{DUPLICATES }
                           {NODUPLICATES}]

[,ALTERNATE [KEY] name] [{DUPLICATES }
                          {NODUPLICATES}] [{CHANGES }
                                             {NOCHANGES}]
```

where:

filename exp is a string expression representing a TOPS-20 file specification.

FOR INPUT requires that the specified file exist. If the file does not exist, an error results. This error causes the OPEN statement to transfer control to the line specified by the INVALID clause if present.

FOR OUTPUT creates a new file with the name you specify.

If you leave the FOR clause out entirely, BASIC searches for an existing file of the specified name. If the search fails, BASIC creates a new file.

AS [FILE] [#] expression

associates the file with a file number. File number 0 (user's terminal) is invalid.

,[ORGANIZATION] SEQUENTIAL

arranges the records in the file by order of input, that is, in serial order.

,[ORGANIZATION] RELATIVE

arranges records by numbered position in the file.

,[ORGANIZATION] INDEXED

arranges records so that they can be accessed by reference to a keyed index.

[FIXED]

specifies that the records are fixed length.

[VARIABLE]

specifies variable-length records in the file. Note that if records are variable length, the buffer is padded with 0's (nulls) after a GET of a record that is smaller than the buffer. This format is the default for all three organizations.

[STREAM]

specifies ASCII stream records (Sequential files only).

USING RECORD FILES

```
[ ,ACCESS ( READ  
           )  
          ( WRITE  
           )  
          ( MODIFY  
           )  
          ( SCRATCH  
           )  
          ( APPEND  
           ) ]
```

specifies the operations that the current user can perform on the file.

READ

allows read only.

WRITE

allows write only.

MODIFY

allows read, write, delete, and update operations. This is the default for sequential, relative and indexed files.

SCRATCH

allows full access; read, write, delete, update and truncate. (Note that a file cannot be accessed by multiple users if it is open with ACCESS SCRATCH.)

APPEND

allows read, write, delete, and update operations with the file pointer initially set to the end of file.

```
[ ,ALLOW ( NONE  
          )  
         ( READ  
          )  
         ( WRITE  
          )  
         ( MODIFY  
          ) ]
```

defines what you allow other users to do to the file while you are using it.

NONE

specifies a protected file.

READ

allows read only. This is the default for sequential, relative, and indexed files.

WRITE

allows write only.

MODIFY

allows read and write access.

,MAP mapname

references a MAP statement. The map buffer you reference defines the buffer used to store the file's data temporarily. The MAP clause can also be used to define the record size.

USING RECORD FILES

NOTE

If you use the RECORDSIZE clause in the same OPEN statement as a MAP clause and define two different record sizes, the size specified in the RECORDSIZE clause overrides the size specified in the MAP clause.

,RECORDSIZE num exp

defines the maximum length of records (in characters) in the file. RECORDSIZE must be specified when no MAP clause is specified.

[,INVALID line no.]

specifies the line number to which control transfers if the OPEN statement fails.

[,LOCKED line no.]

also specifies the line number to which control transfers if the OPEN statement fails because the file is locked (protected).

[,SPAN, NOSPAN]

signifies whether or not records are allowed to cross block boundaries. The default is SPAN.

[,BUCKETSIZE num exp]

specifies the number of TOPS-20 file pages to be associated with each indexed file bucket. The default is 1.

[,BUFFER [#] num exp
,DOUBLEBUF
,BLOCKSIZE num exp
,MODE num exp
,DENSITY num exp
,CLUSTERSIZE num exp]

These attributes have no effect for BASIC-PLUS-2 running under the TOPS-20 operating system; they are for compatibility with other versions of BASIC. If used in a TOPS-20 BASIC program, BUFFER, DOUBLEBUF, and CLUSTERSIZE are ignored. Use of the others will generate errors.

[,PRIMARY [KEY] name]

is required for an indexed file. It defines the name of the primary index key. The size and location of this key is specified in the MAP statement. The name is one of the elements in the list. The key must be a string and cannot be an array element. DUPLICATES are allowed but CHANGES are not.

[,ALTERNATE [KEY] name]

allows you to optionally define the names of one to 254 alternate index keys. Alternate keys must also be strings.

USING RECORD FILES

[{NODUPLICATES}
{DUPLICATES}]

NODUPLICATES is the default. If DUPLICATES is specified for a given key of reference, the file can contain more than one record with the same value for that key.

[{CHANGES}
{NOCHANGES}]

NOCHANGES is the default. If CHANGES is specified for a given key of reference, the value of the field for that key in a given record can be changed. Alternate keys may have changes but the primary key may not. Note that the combination CHANGES and NODUPLICATES is invalid.

The ORGANIZATION clause must be the first attribute specified. If it does not appear first, an error is generated. The other attributes may be specified in any order.

The following sections describe the OPEN statement as it applies to each file organization.

9.5.1.1 Opening a Sequential File - The following syntax is used when opening an existing file or creating a new sequential file:

```
OPEN filename exp [FOR {INPUT }  
                  {OUTPUT}] AS [FILE] [#] expression  
  
[, [ORGANIZATION] SEQUENTIAL [ {FIXED }  
                              {VARIABLE} ]  
  
                              STREAM  
  
[ ,ACCESS { READ }  
          { WRITE }  
          { MODIFY }  
          { SCRATCH }  
          { APPEND } ] [ ,ALLOW { NONE }  
                          { READ }  
                          { WRITE }  
                          { MODIFY } ]  
  
    { ,MAP mapname }  
    { ,RECORDSIZE num exp } [ ,INVALID line no. ] [ ,LOCKED line no. ]  
  
[ { ,SPAN }  
  { ,NOSPAN } ]
```

The following example opens a sequential file:

```
00020 OPEN "CASE" AS FILE #5 &  
      ,ORGANIZATION SEQUENTIAL STREAM &  
      ,ACCESS APPEND, ALLOW NONE, MAP MAP1 &  
      ,INVALID 120, LOCKED 90
```

This statement opens an existing file named CASE, positions the file pointer at the end of the file, and associates the file with file number 5.

The SCRATCH statement allows you to truncate the file at any point. This statement is only valid for a file OPENed with ACCESS SCRATCH. See Section 9.5.4 for a description of the SCRATCH statement.

USING RECORD FILES

9.5.1.2 **Opening a Relative File** - The following syntax opens a relative file:

```

OPEN filename exp [FOR {INPUT } ] AS [FILE] [#] expression
, [ORGANIZATION] RELATIVE [ {FIXED } ]
                        [ {VARIABLE} ]
[ ,ACCESS {READ } ] [ ,ALLOW {NONE } ]
  {WRITE }   {READ }
  {MODIFY}   {WRITE}
                        [ {MODIFY} ]
{ ,MAP (mapname) }
{ ,RECORDSIZE num exp }
[ { ,SPAN } ] [ ,INVALID line no. ] [ ,LOCKED line no. ]
  { ,NOSPAN }

```

The following example opens a relative file:

```

00110 OPEN "FOO" FOR OUTPUT AS FILE #1 &
      ,ORGANIZATION RELATIVE FIXED, ACCESS MODIFY, &
      ALLOW READ, MAP TEST, LOCKED 230

```

This statement creates a new file named FOO and associates it with file number 1. Each record in the file has a fixed length. The user of the file has read and write access capabilities, while other people can only read records.

9.5.1.3 **Opening an Indexed File** - The following syntax opens an indexed file:

```

OPEN filename exp [FOR {INPUT } ] AS [FILE] [#] expression
, [ORGANIZATION] INDEXED [ {FIXED } ]
                        [ {VARIABLE} ]
[ ,ACCESS {READ } ] [ ,ALLOW {NONE } ]
  {WRITE }   {READ }
  {MODIFY}   {WRITE}
                        [ {MODIFY} ]
{ ,MAP mapname } [ ,BUCKETSIZE num exp ] [ ,LOCKED line no. ]
{ ,RECORDSIZE num exp }
, PRIMARY [KEY] name [ {DUPLICATES } ] [ ,INVALID line no. ]
                  [ {NODUPLICATES} ]
[ ,ALTERNATE [KEY] name ] [ {DUPLICATES } ] [ {CHANGES } ]
                          [ {NODUPLICATES} ] [ {NOCHANGES} ]

```

The PRIMARY KEY is mandatory for an indexed file. The following example illustrates the opening of an indexed file:

```

00050 OPEN "ACCOUNT" FOR INPUT AS FILE #4 &
      ,ORGANIZATION INDEXED VARIABLE &
      ,ACCESS MODIFY, ALLOW NONE &
      ,PRIMARY KEY B$, ALTERNATE KEY WAGES$ &
      ,MAP BUFF1

```

USING RECORD FILES

This statement opens an existing file named ACCOUNT and associates the file with file number 4. The records are variable length. The primary index key is in the data field B\$, and there is one alternate key named WAGES\$.

9.5.2 Closing a File

Files should be closed when no longer needed. The CLOSE statement, closes all types of files.

The CLOSE statement has the following format:

```
CLOSE[#]file number(s)
```

where:

file number(s) represent one or several files separated by commas. The number sign, preceding each file number, is optional.

For example:

```
00065 CLOSE #3
```

If you do not specify any file number, BASIC closes all files.

9.5.3 Restoring a File

The RESTORE # statement allows you to bring the file pointer back to the beginning of a record file without disturbing the data.

The RESTORE # statement has the following format:

```
RESTORE[#]file number [,KEY # num exp]
```

where:

file number is the file whose pointer you want to restore to the beginning. The number sign, preceding the file number, is optional.

,KEY # num exp is for indexed files only. This allows you to establish a new key of reference.

For example:

```
00025 RESTORE #6, KEY #0
```

This example brings the file pointer to the key designated by 0. This is the primary key.

9.5.4 Truncating a File

The SCRATCH statement erases the contents of a sequential file from the current file pointer position to the end of the file. The SCRATCH statement has the following format:

```
SCRATCH[#]file number(s)
```

USING RECORD FILES

where:

file number(s) is one or more open sequential files. Separate each file number with a comma. The number sign, preceding each file number, is optional.

To use the SCRATCH statement, the file must be OPENed, with ACCESS SCRATCH.

9.6 RECORD OPERATIONS

There are several operations that you can perform on individual records in a file, depending on its organization. Record-file operations allow you to add, remove, examine, and modify the records within a file. When writing into a file, a program builds records and passes them for storage in the file. When reading a file, a program requests records from the file. With BASIC, you can:

1. Read a record - GET statement
2. Write a record - PUT statement
3. Locate a record - FIND statement
4. Replace a record - UPDATE statement
5. Remove a record - DELETE statement

The GET statement reads a record from the file into a buffer.

The PUT statement writes a new record from the buffer to the file.

The FIND statement locates the specific record in the file and points to it.

The UPDATE statement replaces an existing record with a new one. You must do a FIND or a GET before you can UPDATE.

The DELETE statement erases an existing record from the file. You must do a FIND or a GET before a DELETE operation.

The following sections describe your options in relation to each file organization.

9.6.1 Sequential Record Operations

The following are the operations you can perform on a sequentially organized file:

GET file exp	[,LOCKED line no.] [,INVALID line no.]	
PUT file exp	[{,MAP line no.} {,COUNT exp}]	[,INVALID Line no.] [,LOCKED Line no.]
UPDATE file exp	[{,MAP line no.} {,COUNT exp}]	[,INVALID line no.]
FIND file exp	[,LOCKED line no.]	[,INVALID line no.]

USING RECORD FILES

In a sequential file, a GET operation is performed on succeeding records starting at the beginning of the file. Each successive GET statement retrieves the next record in the file and places it in the buffer identified by the MAP statement. If you retrieve a record that is smaller than the buffer, BASIC fills the buffer with nulls.

A PUT statement in a sequential file writes the record from the buffer to the end of the file without truncating records. You can write only at the end of a sequential file without truncating records. When you are writing to a file, you must specify the record size with a MAP or COUNT clause. The MAP line number in the PUT statement specifies the size of the RECORD to PUT. It cannot be used to specify a different buffer from the one previously specified in the OPEN statement. If the MAP or COUNT clause is not specified, the record size is defined by the MAP or RECORDSIZE clause in the OPEN statement.

In order to replace an existing record with the UPDATE statement, you must first do a successful GET or FIND. You may also reference a MAP line number, or give a record size in characters with COUNT clause. If you do this, however, the new record size must be the same as the one being replaced.

Because you can only access sequential files sequentially, a FIND operation locates the next record in sequence.

You cannot DELETE records in a sequential file.

9.6.2 Relative Record Operations

The following operations can be performed with a relative file:

```
GET file exp    [,RECORD num exp ]  
                [,LOCKED line no.] [,INVALID line no.]  
  
PUT file exp    [,RECORD num exp ]  
                [ {,MAP line no. }  
                  {,COUNT num exp } ]  
                [,LOCKED line no.] [,INVALID line no.]  
  
UPDATE file exp [ {,MAP line no. } ]  
                [ {,COUNT num exp } ]  
                [,LOCKED line no.] [,INVALID line no.]  
  
DELETE file exp [,INVALID line no.]  
  
FIND file exp   [,RECORD num exp ]  
                [,LOCKED line no.] [,INVALID line no.]
```

With relative files, you are allowed random access as well as sequential access. Therefore, you can specify which record you want to GET, FIND, and PUT. If you leave off the record number in the statement, BASIC will read, write, or locate the next record in sequence. Each record position in a relative file need not contain a record. Such record positions are considered to be empty and may appear anywhere in a relative file. A sequential GET or FIND, with no record number specified, will locate the next occupied record position and will bypass empty positions.

New records can be inserted only into empty positions of a file. A PUT operation can be performed only on an empty position or at the end of a file.

USING RECORD FILES

Some record operations change the value of the record pointer and some do not. In a relative file, a sequential GET and a sequential PUT each modify the value of the record pointer.

For example:

```
00100 GET #7, RECORD 2    !Random Retrieves Record 2
00200 GET #7              !Sequential Retrieves Record 3
00300 GET #7              !Sequential Retrieves Record 4
```

A random GET operation also modifies the value of the record pointer; however, a random PUT does not. Consider the following example:

```
00300 GET #1, RECORD 15  !Random Retrieves Record 15
00400 PUT #1, RECORD 20  !Random Writes Record 20
00500 PUT #1              !Sequential Write Records 16
```

Note that line 500 PUTs record 16 (not 21) because the random PUT in line 400 did not change the value of the record pointer.

A FIND operation is only relevant if the next operation is a GET, DELETE, or UPDATE. A PUT after a FIND invalidates the FIND; therefore, a subsequent GET retrieves the next record rather than the record located by the FIND.

In the following example line 600 PUTs record 11:

```
00400 GET #1, RECORD 10  !Retrieves Record 10
00500 FIND #1, RECORD 20 !Locates Record 20
00600 PUT #1              !Writes Record 11
```

However, in this example line 800 UPDATES record 20:

```
00700 FIND #1, RECORD 20 !Locates Record 20
00800 UPDATE #1          !Replaces Record 20
```

9.6.3 Indexed Record Operations

The following operations deal with indexed files only:

```
GET file exp    [ ,KEY #num exp {EQ} string exp ]
                [ ,LOCKED line no.] [ ,INVALID line no.]

PUT file exp    [ { ,MAP line no. } ]
                [ { ,COUNT num exp } ] [ ,INVALID line no.]
                [ ,LOCKED line no. ]

UPDATE file exp [ { ,MAP line no. } ]
                [ { ,COUNT num exp } ] [ ,INVALID line no.]

DELETE file exp [ ,INVALID line no.]

FIND file exp   [ ,KEY #num exp {EQ} string exp ]
                [ ,LOCKED line no.] [ ,INVALID line no.]
```


USING RECORD FILES

In random access to an indexed file, you supply a key of reference previously specified in a MAP statement and defined in an OPEN statement as a PRIMARY or ALTERNATE key. To locate a specific record, specify one of those key matches:

1. Exact key match
2. Approximate key match
3. Generic key match

With the exact key match, BASIC looks for the record that matches the value you assign to the key.

For example:

```
00010 MAP (FILE1) SURNAME$=20,GIVENNAME$=10,SSN$=9,ADDRESS$=40&
,ZIPCODE$=5
00020 MAP (FILE1) NAME$=30,ID$=9,ADDR$=45
00030 OPEN "ACCOUNT" FOR INPUT AS #5,INDEXED VARIABLE,MAP FILE1,&
PRIMARY NAME$,ALTERNATE SSN$,ALTERNATE ZIPCODE$
00040 GET #5, KEY #1 EQ "013445695"
00050 GET #5, KEY #0 EQ "MURPHY"
```

The map at line 10 defines the record as follows:

SURNAME\$	GIVENNAME\$	SSN\$	ADDRESS\$	ZIPCODE\$
-----------	-------------	-------	-----------	-----------

The map at line 20 defines the record as follows:

NAME\$	ID\$	ADDR\$
--------	------	--------

The OPEN statement at line 50 defines the keys within the records as follows:

KEY#	Starting Position	Length
0 (PRIMARY)	0	30
1	30	9
2	79	5

When executing line 40, BASIC refers to the keys specified in the OPEN statement. KEY #1 is the first ALTERNATE key SSN\$. This field is defined in the MAP at line 10. BASIC searches for an exact match of a record with "013446595" starting at position 30.

When executing line 50, BASIC again refers to the keys in the OPEN statement. KEY #0 is the PRIMARY key NAME\$. This key is part of the record described in the MAP on line 20. BASIC searches for an exact match of a record with a field "MURPHY" in starting position 0.

The second type of search, the approximate key match, allows you to request the record closest to the value you specify. The proximity is determined by the ASCII collating sequence. The approximate key search allows your program to select either of the following relationships:

1. Equal to or greater than (GE)
2. Greater than (GT)

USING RECORD FILES

If the key requested does not exist, BASIC returns the record that contains the next higher key value. This allows you to retrieve records without knowing the exact key.

For example:

```
00040 GET #5,KEY #0 GE "JONES"  
00050 GET #5,KEY #0 GT "ABRAMSON"
```

Line 40 defines the key as PRIMARY and searches for a data field that is greater than or equal to (GE) the value "JONES", for example, "KNIGHT".

Line 50 also uses the PRIMARY key but searches for a data field greater than "ABRAMSON", for example, "ADAMS". "ABRAMSON" is not an acceptable match in this case.

The third type of search, the generic key match, allows you to effect a match by specifying a key value with fewer characters than were specified for the corresponding field in the record. The match occurs if the first characters in the field are identical to the key value.

If you do not specify a KEY, you effect a sequential GET according to the previous KEY specified. BASIC will then retrieve the next record in the index according to the ASCII collating sequence.

When you PUT to an indexed file, you merely specify:

```
PUT file exp
```

BASIC places the record in the proper index.

In addition to read, write, and find operations, your program can delete any record in an indexed file and update any record. However, during an update operation, be sure that the contents of the modified record do not change the primary key value. You can change alternate key values if the CHANGES clause is specified for that alternate key in the OPEN statement.

9.6.4 Record Locking

If you plan to allow file sharing (simultaneous access) by specifying ALLOW READ, ALLOW WRITE, or ALLOW MODIFY, you should be aware of the correlation between an I/O operation and the locked status of a record. Records are locked according to the attributes of the file specified in the OPEN statement and according to the particular I/O operation you perform on the record. The ACCESS clause of the OPEN statement determines the effect the I/O operation has on the locked status of a record.

If you OPEN with ACCESS READ,

GET	locks the record only for the duration of the operation. The record is unlocked when the GET is completed.
FIND	locks the record until a GET is completed or until another record is accessed with a FIND or GET operation.

USING RECORD FILES

If you OPEN with ACCESS WRITE, MODIFY, or APPEND,

GET locks the record until another record is accessed by a GET, FIND, or PUT operation; or until the current record is UPDATED or DELETED.

FIND locks the record until another record is accessed by a GET, FIND, or PUT operation; or until the current record is UPDATED or DELETED.

Locks are somewhat more complicated for indexed files opened with ACCESS, WRITE, or MODIFY. If you OPEN an indexed file with ACCESS, WRITE, or MODIFY, an additional lock and unlock is automatically done to freeze the index structure for the duration of each operation. The data is locked as mentioned above. The lock, however, is applied to the entire data bucket in which the desired record is located. This in effect causes all the records in that data bucket to be locked until the lock is released.

9.7 DYNAMIC MAPPING OF AN I/O BUFFER

The MAP statement (described in Section 9.4) defines the format of a record when that format can be specified at compile time. When you cannot specify the record format at compile time, the MOVE statement can be used to dynamically access the data in a record. For example, you can use MOVE to access a record in which the lengths of strings or arrays in the record are specified by fields at the beginning of the record. The MOVE statement associates the data in a record with the variables you specify in an I/O list. The format of the MOVE statement is:

```
MOVE [ {ALIGNED } ] {FROM} file exp, I/O list
      {UNALIGNED} {TO }
```

where:

ALIGNED ALIGNED is the permanent setting for BASIC-PLUS-2
UNALIGNED running under the TOPS-20 operating system;
UNALIGNED is ignored.

FROM moves the data from the buffer associated with the
file number and places the data in the elements in
the I/O list.

TO moves the data from the elements in the I/O list and
places it in the buffer associated with the file
number.

file exp is the file number associated with the file OPENed
previously.

, the comma is mandatory between file exp and I/O list.

USING RECORD FILES

I/O list is a list of valid elements:

1. numeric, integer, string variables
2. arrays
3. array elements
4. fill specifiers

The length of a string may be defined in the I/O list, for example, A\$=n, where n is a valid numeric expression. The default length for MOVE TO is LEN(A\$); the default for MOVE FROM is 16.

An array specified in a MOVE statement must have the following format:

```
A( ) list
A(,) matrix
```

Caution, row zero and column zero are destroyed by the MOVE statement. You should, therefore, never use these locations in an array.

You specify an array element by name, that is:

```
A(25)
```

The following are examples of MOVE statements:

```
00060 MOVE FROM #5, A$, B, FILLZ, C( )
00085 MOVE TO #5, A$, B, FILLZ, C( )
```

Successive MOVE statements to or from the same file each start at the beginning of the buffer. The size of the buffer is not affected by MOVE statements. If a MOVE only partially fills a buffer, the rest of the buffer is unchanged.

To retrieve a record from a file, first read the record with a GET statement. This places the record in a buffer. This buffer can be one assigned by the BASIC system buffer or a buffer you have set up with a MAP statement.

Then a MOVE FROM places the data from the buffer into the elements in the I/O list. Once the data is associated with the elements, you can reference them in the program.

A MOVE TO moves the data from the elements in the I/O list to a buffer. To move the data into a file, perform a write operation using the PUT statement.

Consider the following program:

```
00010 OPEN "MOVE.DAT" AS FILE #1, ORGANIZATION SEQUENTIAL,&
ACCESS MODIFY,ALLOW NONE
00020 GET #1
00030 MOVE FROM #1, I, A$=I
00040 A$=A$ + ","
00050 I = I+1
00060 MOVE TO #1, I, A$=I
00070 UPDATE #1
00080 CLOSE #1
00090 END
```

USING RECORD FILES

This program opens an existing file named MOVE.DAT, reads the first record into the buffer, and associates the data with the variables in the MOVE FROM statement.

The MOVE TO places the record into the buffer, and the UPDATE statement writes the record back into file #1. The file is then closed and the program ends.

9.8 EXAMPLES

The following examples consist of BASIC-PLUS-2 programs using sequential, relative, and indexed files. Each program is listed and executed.

Example 1

```
LISNH
00010 MAP (MAP1) NAME#=30, IDNUM%, JOBCLASS#=8%
00020 OPEN "SEQ" FOR OUTPUT AS FILE #1, ORGANIZATION SEQUENTIAL FIXED, &
ACCESS MODIFY, MAP MAP1
00030 INPUT 'NAME' ; NAME%
00040 IF NAME% = 'END' THEN 70
00050 INPUT 'ID NUMBER' ; IDNUM%
00055 INPUT 'JOB CLASS' ; JOBCLASS%
00060 PUT #1
00065 GOTO 30
00070 CLOSE #1
00100 END
```

```
READY
RUNNH
NAME      ? SARAH
ID NUMBER ? 36577
JOB CLASS ? 4
NAME      ? TONY
ID NUMBER ? 76545
JOB CLASS ? 4
NAME      ? LORRAINE
ID NUMBER ? 34766
JOB CLASS ? 4
NAME      ? JAY
ID NUMBER ? 54877
JOB CLASS ? 4
NAME      ? DANNY
ID NUMBER ? 21344
JOB CLASS ? 2
NAME      ? MATTHEW
ID NUMBER ? 67544
JOB CLASS ? 2
NAME      ? ROSE
ID NUMBER ? 65488
JOB CLASS ? 4
NAME      ? IZZIE
ID NUMBER ? 38766
JOB CLASS ? 4
NAME      ? END
```

```
READY
```

USING RECORD FILES

Example 2

```
OLD OPEN
READY
LISNH
00005 MAP AREA A$,B$,C$
00010 OPEN "TONY" FOR OUTPUT AS FILE #2,RELATIVE VARIABLE,MAP AREA
00020 A$="HI","\B$="HOW ARE YOU?","\C$="I'M FINE."
00025 PUT #2
00030 CLOSE #2
00035 OPEN "TONY" FOR INPUT AS FILE #2,RELATIVE VARIABLE,MAP AREA
00040 GET #2
00050 PRINT B$
00060 CLOSE #2
00070 END
```

```
READY
RUNNH
HOW ARE YOU?,
```

```
READY
```

Example 3

```
OLD PUT
```

```
READY
LISNH
00010 MAP BUFF A$,B$
00020 OPEN "DUMB" AS 1, SEQUENTIAL, ACCESS APPEND,MAP BUFF
00030 A$="TONY"\B$="ECG"
00040 PUT #1
00050 CLOSE
00060 OPEN "DUMB" AS 1, SEQUENTIAL, MAP BUFF
00070 GET #1
00080 PRINT A$;B$
00090 CLOSE
00100 END
```

```
READY
RUNNH
TONY          ECG
```

```
READY
```

USING RECORD FILES

Example 4

```
LISNH
00010 MAP FILES LASTNAM$,SALARY,AGE
00020 OPEN "EMPLOYEE" FOR OUTPUT AS #1, INDEXED VARIABLE,&
MAP FILES, PRIMARY LASTNAM$
00030 INPUT 'LAST NAME, SALARY,AGE';LASTNAM$,SALARY,AGE
00035 IF LASTNAM$='END' AND SALARY=0 AND AGE=0 GOTO 50
00040 PUT #1
00045 GOTO 30
00050 CLOSE #1
00100 END
```

READY

RUNNH

```
LAST NAME, SALARY,AGE ? ANDERSON,10000,22
LAST NAME, SALARY,AGE ? BROWN, 11000,23
LAST NAME, SALARY,AGE ? CROSBY,12000,24
LAST NAME, SALARY,AGE ? DAVIDSON,13000,25
LAST NAME, SALARY,AGE ? ELLEN,14000,26
LAST NAME, SALARY,AGE ? FOSTER,15000,27
LAST NAME, SALARY,AGE ? GEORGE,16000,28
LAST NAME, SALARY,AGE ? HALPER,17000,29
LAST NAME, SALARY,AGE ? ILSON,18000,30
LAST NAME, SALARY,AGE ? JENSEN,19000,31
LAST NAME, SALARY,AGE ? KELLER,20000,32
LAST NAME, SALARY,AGE ? LEVINE,21000,33
LAST NAME, SALARY,AGE ? MASELLA,22000,34
LAST NAME, SALARY,AGE ? NATHAN,23000,35
LAST NAME, SALARY,AGE ? END,0,0
```

READY

Example 5

```
LISNH
00010 MAP FILES LASTNAM$,SALARY,AGE
00020 OPEN "EMPLOYEE" FOR INPUT AS #1, INDEXED VARIABLE,&
MAP FILES, PRIMARY LASTNAM$
00030 GET #1,KEY #0 GT "KELLER"
00040 PRINT LASTNAM$
00045 PRINT AGE
00048 GET #1,KEY #0 GT "JOLSON"
00049 PRINT LASTNAM$
00050 CLOSE #1
00060 END
```

READY

RUNNH

LEVINE

33

KELLER

READY

CHAPTER 10

FORMATTED OUTPUT - THE PRINT USING STATEMENT

When the format as well as the content of output is important, use the PRINT USING statement rather than the PRINT statement. The PRINT USING statement allows you to control the appearance and location of both string and numeric data on the output line thus enabling you to create formatted lists, tables, reports, and forms.

The following examples print a series of numbers. One program uses the PRINT statement and the other uses the PRINT USING statement.

PRINT	PRINT USING
00010 PRINT 1	00010 PRINT USING "#####.##",1
00020 PRINT 100	00020 PRINT USING "#####.##",100
00030 PRINT 1000000	00030 PRINT USING "#####.##",1000000
00040 PRINT 100.3	00040 PRINT USING "#####.##",100.3
00050 PRINT .0123456	00050 PRINT USING "#####.##",.0123456
READY	READY
RUNNH	RUNNH
1	1.00
100	100.00
1E+6	1000000.00
100.3	100.30
0.0123456	0.01
READY	READY

PRINT left-justifies numbers and outputs certain numbers in E (exponential) format. These characteristics make it difficult to compare numbers. In contrast, PRINT USING allows you to format numbers so that the decimal points are aligned, making it easier to compare the column of numbers.

This chapter describes formatting output in the following sections:

- 10.1 The PRINT USING Statement
- 10.2 Formatting Numbers with PRINT USING
- 10.3 Formatting Strings with PRINT USING
- 10.4 Summary of the Format Characters
- 10.5 The IMAGE Statement
- 10.6 Error Conditions

FORMATTED OUTPUT - THE PRINT USING STATEMENT

10.1 THE PRINT USING STATEMENT

The format of the PRINT USING statement is:

```
PRINT USING (string  
            {string variable}  
            (line number) ) ,list
```

where:

string is a coded-format image of the line to be printed, a string variable to which a format image has been assigned, or the line number of an IMAGE statement (See Section 10.5). It is called the format string. If it is a string constant, it must be enclosed in double quotation marks, not single quotation marks.

list contains the items to be printed. Note that a comma or semicolon separating items in the list has no effect on the output. However, a comma or semicolon placed after the last item in the list suppresses the carriage return/line feed normally generated after the last list item.

In the following example

```
00010 PRINT USING "HI 'LLLLL YOU WEIGH ###.# LBS.,"PAUL",145  
00020 END
```

the format string is

```
"HI 'LLLLL YOU WEIGH ###.# LBS."
```

The list contains two data items: the string constant "PAUL", and the integer 145.

In the format string, there are two fields corresponding to the two data items. The first field is 'LLLLL which corresponds to the first data item, "PAUL", and the second field is ###.# and it corresponds to the second data item, 145. When BASIC prints the line, it prints each data item in the position and format specified by the field. The rest of the format string, namely "HI YOU WEIGH LBS." is the printed message. The output of this example is:

```
RUNNH  
HI PAUL YOU WEIGH 145.0 LBS.
```

10.2 FORMATTING NUMBERS WITH PRINT USING

When you want to format numeric data for output, use the PRINT USING statement to control:

- Number of digits
- Location of decimal point

FORMATTED OUTPUT - THE PRINT USING STATEMENT

- Inclusion of symbols (trailing minus sign, asterisks, dollar sign, commas)
- Exponential format

10.2.1 Specifying the Number of Digits

With PRINT USING, you can specify the number of places reserved for digits in a field by specifying a corresponding amount of number signs (#).

For example:

```
00010 PRINT USING "###",123
00020 PRINT USING "#####",12345

READY
RUNNH
123
12345
```

If there are not enough digits to fill the field specified, BASIC prints spaces before the first digit. For example:

```
00010 PRINT USING "#####",1
00020 PRINT USING "#####",10,
00030 PRINT USING "#####",1709
00040 PRINT USING "#####",12345

READY
RUNNH
  1
 10 1709
12345
```

Note that spaces are printed before the number so that the entire field is filled. The number sign indicates where BASIC prints a space. Placing a comma or semicolon after the list item (line 20) causes the next item to be printed on the same line.

BASIC rounds numbers printed with PRINT USING. For example:

```
00010 PRINT USING "###",126.7
00020 PRINT USING "#",5.9
00030 PRINT USING "#",5.4

READY
RUNNH
127
6
5
```

FORMATTED OUTPUT - THE PRINT USING STATEMENT

10.2.2 Specifying the Decimal Point Location

You can reserve any number of digits on both sides of the decimal point by placing a decimal point in the number sign field. BASIC always prints the digits to the right of the decimal point, even if they are zeros. Consider the following example:

```
00010 PRINT USING "##.###",5.72
00020 PRINT USING "##.###",39.3758
00030 PRINT USING "##.###",26
```

```
READY
RUNNH
 5.720
39.376
26.000
```

Note that BASIC prints spaces to the left of the decimal point as necessary, but prints zeros to the right of the decimal point. Also note that 39.3758 is rounded to 39.376.

If there is more than one number sign to the left of the decimal point, at least one digit is printed to the left of the decimal point. If only one number sign is to the left of the decimal point and the number is negative and less than 1, BASIC prints the minus sign to the left of the decimal point instead of a zero.

For example:

```
00010 PRINT USING "###.#",.99
00020 PRINT USING "###.#",.1
00030 PRINT USING "#.#",-.1
```

```
READY
RUNNH
 1.0
 0.1
-.1
```

10.2.3 Printing a Number that is Larger than the Field

If you have not supplied enough number signs for a number, BASIC prints a percent sign (%) followed by the number and ignores the format specified by the field. After BASIC prints the number, it completes the rest of the PRINT USING statement in the usual manner.

Consider the following example:

```
00010 PRINT USING "###.##",256.786
00020 PRINT USING "##.##",256.786
```

```
READY
RUNNH
256.79
% 256.786
```

FORMATTED OUTPUT - THE PRINT USING STATEMENT

The number at line 10 is printed correctly (with rounding). In line 20, there are only 2 number signs to the left of the decimal point; therefore, the number will not fit. The number is printed in the usual default PRINT-statement format, with a space before and after it, but the number is preceded by a percent sign.

Be sure to enter one number sign for every number that will print on the left side of the decimal point. Add one more number sign in case the number is negative. (For another method of reserving a place for the minus sign, see Section 10.2.4.1.) For example:

Field	There are enough places for	But not enough places for
###.##	100.569	%-100.569
.####	.1258	%-.12579

A number can become larger than its field, when rounding increases the number of places needed, as when .99 is rounded off to 1.00. For example:

```
00010 PRINT USING ".###",.999
00020 PRINT USING ".##",.999
00030 PRINT USING "#.##",.999

READY
RUNNH
.999
% 0.999
1.00
```

10.2.4 Printing Numbers With Special Symbols

The PRINT USING statement enables you to include the following symbols in numeric output:

- Trailing minus sign
- Leading asterisks
- Floating dollar signs
- Commas

10.2.4.1 Printing Numbers with a Trailing Minus Sign - To print the minus sign for negative numbers after the number instead of before it, specify a trailing minus sign in a field. You must use the trailing minus sign to print a number in an asterisk-fill or floating-dollar-sign field.

If a field contains a trailing minus sign, BASIC prints a negative number as the number followed by a minus sign, and prints a positive number as the number followed by a space.

FORMATTED OUTPUT - THE PRINT USING STATEMENT

For example:

Standard Fields	Fields with Trailing Minus Signs
00010 PRINT USING "###.##",-10.54	00010 PRINT USING "##.##-",-10.54
00020 PRINT USING "###.##",10.54	00020 PRINT USING "##.##- ",10.54
READY	READY
RUNNH	RUNNH
-10.54	10.54-
10.54	10.54

10.2.4.2 Printing Numbers in Asterisk-Fill Fields - To print a number with asterisks (*) filling up any blank spaces before the first digit, start the field with two asterisks. For example:

```
00005 PRINT USING "####.##",1.2
00010 PRINT USING "####.##",27.95
00020 PRINT USING "####.##",107
00030 PRINT USING "####.##",1007.5

READY
RUNNH
**1.20
**27.95
*107.00
1007.50
```

Note that the asterisks reserve two places as well as cause asterisk fill.

To print a negative number in an asterisk-fill field, specify a trailing minus sign in the field. For example:

```
00010 PRINT USING "####.##-",27.95
00020 PRINT USING "####.##-",107
00030 PRINT USING "####.##-",1007.5

READY
RUNNH
**27.95
*107.00-
1007.50-
```

If you attempt to print a negative number in an asterisk-fill field without a trailing minus sign, BASIC prints a fatal error message.

FORMATTED OUTPUT - THE PRINT USING STATEMENT

10.2.4.3 Printing Numbers with Floating Dollar Signs - To print a number with a dollar sign (\$) before the first digit, start the field with two dollar signs. The two dollar signs reserve output places for a single dollar sign and a digit, for example: \$\$## allows for \$1.95. If a negative number is desired, end the field with a trailing minus sign. For example:

```
00010 PRINT USING "####.## ", 77.44
00020 PRINT USING "####.##", 304.55
00030 PRINT USING "####.##", 2211.42
00040 PRINT USING "####.##-", -125.6
00050 PRINT USING "####.##", 127.82
```

```
READY
RUNNH
  $77.44
 $304.55
% 2211.42
 $125.60-
 $127.82
```

Note that the dollar sign is always printed. Contrast this with the asterisk fill-field where asterisks are printed only if there are leading spaces.

If you attempt to print a negative number in a dollar-sign field without a trailing minus sign, BASIC prints a fatal error message.

10.2.4.4 Printing Numbers with Commas - To insert commas in a number, place a comma in the format field, anywhere to the left of the decimal point (if present). Regardless of where the comma appears in the format field, BASIC prints a comma every third digit to the left of the decimal point. If there is no digit to be printed to the left of the comma, BASIC does not print the comma. Commas in the format field to the right of the decimal point are treated as literals. For example:

```
00010 PRINT USING "##,###", 10000
00020 PRINT USING "##,###", 759
00030 PRINT USING "###,###.##", 25694.3
00040 PRINT USING "**#,###", 7259
00050 PRINT USING "####,#.##", 25239
00060 END
```

```
READY
RUNNH
10,000
  759
 $25,694.30
 **7,259
25,239.00
```

FORMATTED OUTPUT - THE PRINT USING STATEMENT

10.2.5 Printing Numbers in E (Exponential) Format

To print a number in E (exponential) format, place four circumflexes ($\wedge\wedge\wedge\wedge$, also called up-arrows) at the end of the field. The circumflexes reserve space for the capital letter E followed by a space, or minus sign (which indicates a positive or negative exponent, respectively) and then a one- or two-digit exponent. In exponential format the digits to the left of the decimal point are not filled with spaces. Instead the first non-zero digit is shifted to the leftmost place and the exponent is adjusted to compensate.

For example:

```
00010 PRINT USING "###.## $\wedge\wedge\wedge\wedge$ ", 5
00020 PRINT USING "###.## $\wedge\wedge\wedge\wedge$ ", 1000
```

```
READY
RUNNH
500.00E-02
100.00E+01
```

If you use fewer than four circumflexes, the number is not printed in E format but the circumflexes are printed as a literal. If you use more than four circumflexes, the number prints in E format but the extra circumflexes are also printed. For example:

```
00010 PRINT USING "###.## $\wedge\wedge\wedge$ ",5
00020 PRINT USING "###.## $\wedge\wedge\wedge\wedge\wedge\wedge$ ",5
```

```
READY
RUNNH
5.00 $\wedge\wedge\wedge$ 
500.00E-02 $\wedge\wedge\wedge\wedge\wedge\wedge$ 
```

Exponential format cannot be used with asterisk-fill, floating-dollar sign, or trailing-minus formats.

10.2.6 Fields that Exceed BASIC's Accuracy

If a field contains more places than there are digits of accuracy, BASIC prints zeros in all the places following the last significant digit.

10.3 FORMATTING STRINGS WITH PRINT USING

By using the PRINT USING statement, you can specify whether strings are printed in a left-justified, right-justified, centered, or extended format. String fields start with a single quotation mark ('). The single quotation mark is optionally followed by a contiguous series of uppercase L's, R's, C's, or E's representing left-justified, right-justified, centered, and extended string fields, respectively.

If a string is larger than the string field, BASIC prints as much of the string as fits and ignores the rest; the only exception is extended fields, in which case BASIC prints the entire string.

FORMATTED OUTPUT - THE PRINT USING STATEMENT

10.3.1 One-Character String Fields

A string field consisting of only a single quotation mark or a single exclamation point is a 1-character string field. BASIC prints the first character of the corresponding string and ignores all following characters. For example:

```
00010 PRINT USING "'", "ABCDE"  
  
READY  
RUNNH  
A
```

10.3.2 Printing Strings in Left-Justified Format

If you specify a left-justified field, BASIC prints the string starting at the leftmost position. If there are any unused places, BASIC prints spaces after the string. If there are more characters than places, BASIC truncates the string and does not print the excess characters.

A left-justified field is composed of a single quotation mark followed by a series of capital L's. An alternative way to specify a left-justified field is to use a backslash followed by any number of spaces, and a second backslash. Each space, in this format, increases the field width one character from the minimum of two for the backslash pair.

For example:

```
00010 PRINT USING "'LLLLLL", "ABCD"  
00020 PRINT USING "'LLLL", "ABC"  
00030 PRINT USING "'LLLL", "12345678"  
00040 40 PRINT USING "\ \ ", "JKLMN"  
  
READY  
RUNNH  
ABCD  
ABC  
12345  
JKLM
```

10.3.3 Printing Strings in Right-Justified Format

If you specify a right-justified field, BASIC prints the string so that the last character of the string is in the rightmost place of the field. If there are any unused places before the string, BASIC prints spaces to fill the field.

FORMATTED OUTPUT - THE PRINT USING STATEMENT

A right-justified field is composed of a single quotation mark followed by a series of capital R's. For example:

```
00010 PRINT USING "RRRRRR","ABCD"  
00020 PRINT USING "RRRRRR","A"  
00030 PRINT USING "RRRRRR","XYZ"
```

```
READY  
RUNNH  
  ABCD  
   A  
  XYZ
```

If there are more characters than places, BASIC left-justifies the string and does not print the excess characters.

10.3.4 Printing Strings in Centered Fields

If you specify a centered field, BASIC prints the string so that the center of the string is in the center of the field. If the string cannot be exactly centered, such as a 2-character string in a 5-character field, BASIC prints the string one character off center to the left.

A centered field is composed of a single quotation mark followed by a series of capital C's. For example:

```
00010 PRINT USING "CCCCCCC","A"  
00020 PRINT USING "CCCCCCC","AB"  
00030 PRINT USING "CCCCCCC","ABC"  
00040 PRINT USING "CCCCCCC","ABCD"  
00050 PRINT USING "CCCCCCC","ABCDE"
```

```
READY  
RUNNH  
  A  
 AB  
 ABC  
 ABCD  
 ABCDE
```

If there are more characters than there are places in the field, BASIC left-justifies the string and does not print the excess characters.

10.3.5 Printing Strings in Extended Fields

The extended field is the only field that automatically prints the entire string. If you specify an extended field, BASIC left-justifies the string as it does for a left-justified field. But, if the string has more characters than there are places in the field, BASIC extends the field and prints the entire string. This extension may cause other items to be misaligned.

An extended field is composed of a single quotation mark followed by a series of capital E's.

FORMATTED OUTPUT - THE PRINT USING STATEMENT

Consider the following example that uses extended, left-justified, right-justified, and centered fields:

```

00010 PRINT USING 'LLLLLLLLLL',"THIS TEXT"
00020 PRINT USING 'LLLLLLLLLLLLLLLL',"SHOULD PRINT '
00030 PRINT USING 'LLLLLLLLLLLLLLLL',"AT LEFT MARGIN"
00040 PRINT USING 'RRRR',"1,2,3,4"
00050 PRINT USING 'RRRR',"1,2,3"
00060 PRINT USING 'RRRR',"1,2"
00070 PRINT USING 'RRRR',"1"
00080 PRINT USING 'cccccccc',"A"
00090 PRINT USING 'CCCCCCCC',"ABC"
00100 PRINT USING 'CCCCCCCC',"ABCDE"
00110 PRINT USING 'CCCCCCCC',"ABCDEFG"
00120 PRINT USING 'CCCCCCCC',"ABCDEFGHI"
00130 PRINT USING 'LLLLLLLLLLLLLLLL',"YOU ONLY SEE HALF OF THE LINE"
00140 PRINT USING 'E',"YOU CAN SEE ALL OF THE LINE WHEN EXTENDED"
00150 END

```

```

READY
RUNNH
THIS TEXT
SHOULD PRINT
AT LEFT MARGIN
1,2,3
1,2,3
 1,2
   1
    A
   ABC
  ABCD
 ABCDEF
ABCDEFHI
YOU ONLY SEE HALF
YOU CAN SEE ALL OF THE LINE WHEN EXTENDED

READY

```

10.4 SUMMARY OF FORMAT CHARACTERS

Table 10-1
Format Characters For Numeric Fields

Character	Effect on Format
# number sign	Reserves place for one digit.
. decimal point (period)	Determines location of decimal point.
, comma	Causes a comma to be printed between every third digit starting from the decimal point and proceeding from right to left.

FORMATTED OUTPUT - THE PRINT USING STATEMENT

Table 10-1 (Cont.)
Format Characters For Numeric Fields

** two asterisks	Cause leading asterisks instead of spaces to be printed before the first digit. The field formed is called an asterisk-fill field. They also reserve places for two digits.
\$\$ two dollar signs	Cause a dollar sign to be printed before the first digit. The field formed is called a dollar-sign field. They reserve places for one dollar sign and one digit.
^^^^ four circumflexes (up-arrows)*	Cause the number to be printed in E (exponential) format. They also reserve four places for the E notation.
- minus sign	Causes a trailing minus sign to be printed when number is negative. Printing a negative number in an asterisk-fill or a dollar-sign field requires that the field also have a trailing minus sign.

*Note that neither \$\$ nor ** can be combined with ^^^^^.

Table 10-2
Sample Formats for Numeric Fields

Valid Fields	Sample Output	Description
\$\$###.##	\$1234.50	Dollar sign field
#####	**12	Asterisk fill field
###,#	1,242	Comma in field
##.##^^^^	20.72E-02	E (exponential) format field
Invalid Fields	Reason	
**##.##^^^^	** can not be combined with ^^^^^	
##.##,#	Comma is to the right of the decimal point	
\$\$*#####.##	\$\$ can not be combined with **	

String fields are composed of a single quotation mark optionally followed by a series of contiguous capital L's, R's, C's, or E's. The effect these characters have on the format is described in Table 10-3.

FORMATTED OUTPUT - THE PRINT USING STATEMENT

Table 10-3
Format Characters for String Fields

Character	Effect on Format
' (single quotation mark) or ! (exclamation point)	Starts string field and reserves place for one character.
L (uppercase)	Causes string to be left-justified and reserves place for one character.
\\ (two backslashes)	Starts string field, causes string to be left-justified, and reserves place for two characters.
R (uppercase)	Causes string to be right-justified and reserves place for one character.
C (uppercase)	Causes string to be centered in field and reserves place for one character.
E (uppercase)	Causes string to be left-justified, expands field, as necessary, to print the entire string and reserves place for one character.

10.5 THE IMAGE STATEMENT

The IMAGE statement enables you to store PRINT USING output format(s) as unquoted literals. Any PRINT USING statement can then use the output format(s) by referencing the line number of the IMAGE statement.

The IMAGE statement has the following formats:

```
:unquoted string  
IMAGE unquoted string
```

where:

```
: IMAGE are interchangeable.
```

```
unquoted string is the format for printing characters listed  
in the PRINT USING statement. (See Section  
10.3.)
```

For example:

```
00010: ##.## ##.##  
00020 PRINT USING 10, 12.345,-12.5  
  
RUNNH  
12.35% -12.5
```

All characters following the colon or the word IMAGE except the line terminator are considered part of the output image.

FORMATTED OUTPUT - THE PRINT USING STATEMENT

The IMAGE statement must be the only statement on a line. No comment fields are allowed. You can, however, continue a format for printing characters to another line with an ampersand (&). BASIC continues the line until it finds a line terminator without an ampersand preceding it.

IMAGE is a non-executable statement; therefore, BASIC ignores the IMAGE statement until a PRINT USING statement appears in the program.

You can also use an IMAGE statement with both numeric and string formats. The following example demonstrates the use of a string format in an IMAGE statement:

```
00010: ++ 'CCCC ++ 'LLLL
00020 INPUT A$
00030 IF A$ = "STOP" GOTO 50
00040 PRINT USING 10, A$, A$
00050 END
```

```
READY
RUNNH
? ABCDE
++ ABCDE ++ ABCDE
```

10.6 PRINT USING STATEMENT ERROR CONDITIONS

There are two types of PRINT USING error conditions: fatal and warning. When a fatal error occurs, BASIC stops executing the program and prints a fatal error message. When a warning condition occurs, BASIC continues to execute the program, although the resulting output may not be in the format intended.

10.6.1 Fatal Error Conditions

A fatal error message is produced if:

1. The format string is not a valid string expression.
2. There are no valid fields in the format string.
3. A string is printed in a numeric field.
4. A number is printed in a string field.
5. A negative number is printed in a floating-dollar-sign or asterisk-fill field that does not specify a trailing minus.
6. An attempt is made to print a special formatting characters, that is, backspace, carriage return, line feed, or formfeed.

10.6.2 Warning Conditions

Warning error conditions are:

1. A number does not fit in the field. If a number is larger than the field allows, BASIC prints a percent sign (%) followed by the number in the standard PRINT format.

FORMATTED OUTPUT - THE PRINT USING STATEMENT

2. A string does not fit in the field. If a string is larger than any field other than an extended field, BASIC truncates the string and does not print the excess characters.
3. A field contains an illegal combination of characters. If a field contains an illegal combination of characters, the first illegal character and all characters to its right are not recognized as part of the field. They may form another valid field or they may be considered text. If the illegal characters form a new valid field, this field may cause a fatal error condition.

Consider the following examples of illegal combinations of characters in numeric fields.

```
00010 PRINT USING "$**##.##",5.41, 16.30
```

```
READY
RUNNH
$5**16.30
```

\$\$ are combined with **. \$\$ is a complete field and **#. # forms a second valid field. \$5 is printed by \$\$ and **16.30 is printed by **##.#.

```
00010 PRINT USING "$**##.## (LLL)",5.41, "ABC"
```

```
READY
RUNNH
```

```
? 234 Numeric IMAGE specified for string at line 00010 of MAIN PROGRAM
```

The same illegal combination appears here, but the next data item is a string. BASIC produces the fatal error message after trying to print the string "ABC in the numeric field **#.##.

```
00010 PRINT USING "##.#^",5.43E09
```

```
READY
RUNNH
% 5.43E 09^
```

Field has only three (not four) circumflexes. The number does not fit in the field ##.#; a % and the number are printed followed by the ^.

```
00010 PRINT USING "'LLEEE","VWXYZ"
```

```
READY
RUNNH
VWXEEE
```

Two letters can not be combined in one field. EEE is printed as it is.

Attempting to print characters as text produces errors when the characters form a valid field. For example:

```
00010 PRINT USING "THERE ARE ### # ## NAILS",123,4,16,6
```

FORMATTED OUTPUT - THE PRINT USING STATEMENT

is an attempt to print

```
THERE ARE 123 # 4 NAILS
THERE ARE 16 # 6 NAILS
```

but instead produces

```
THERE ARE 123 4 16 NAILS
THERE ARE 6
```

To correctly print characters that form a valid field, use a string field and place the characters as a string constant in the list. For example:

```
00010 A$="THE BALANCE OF ACCOUNT '###' is $####.##"
00020 PRINT USING A$,"#",5634,107.56
```

```
READY
RUNNH
```

```
THE BALANCE OF ACCOUNT #5634 is $107.56
```

This is also the only way to print a single or double quotation mark character with the PRINT USING statement.

APPENDIX A
DIAGNOSTIC MESSAGES

BASIC-PLUS-2 diagnostic messages are divided into three categories:

1. Command Error Messages
2. Compilation Error Messages
3. Execution Messages

The text of most messages is self-explanatory. The following symbols indicate the gravity of the error:

- ? indicates a fatal error. Program compilation or execution stops.
- % indicates a warning condition. Program compilation or execution continues.

The trappable execution error messages, used with the ONERROR GOTO statement (See Section 4.8.1.), are listed in Section A.3.

A.1 COMMAND ERROR MESSAGES

The following is an alphabetical list of the error messages printed at BASIC command level. Where necessary, additional text is added to clarify the meaning of the error message.

- ? Bad command syntax
- ? BAD INPUT LINE NUMBER
- ? Cannot CONTINUE, use RUN or DEBUG.
- ? Cannot resequence source program
- ? Cannot START, use RUN or DEBUG..
- ? Cannot use RESEQUENCE command to move source lines
- ? Do command illegal from within a command file
- ? Due to compile errors the BUILD is aborted.
- ? Due to errors, the CHAIN is aborted.
- ? Due to errors, the line will not be executed
- ? Due to errors, the RUN is aborted.

DIAGNOSTIC MESSAGES

? Free storage exhausted

Message indicates a size limitation; make the current program smaller.

? Illegal character before line number

? Input line to OLD does not have line number

? Internal consistency errors. Please use SCRATCH ALL to fix them.

? Line number less than 1

? Line number greater than 99999

? Line table is full

There are too many lines in the program; make the program smaller.

? Line too long

? Line-range not permissible as increment to resequence

? Line-range not permissible as new line number to resequence

? NO PROGRAM TO SAVE-COMMAND IGNORED

? No program, compile aborted

? Not in this version

? Proceed at your own risk ...

There is an internal error: SAVE the program, exit to system level, and run BASIC again.

? Resequencing aborted - program may be in an inconsistent state

? Statement illegal in immediate mode

? Too many line numbers to resequence

? What

General message; indicates that BASIC did not understand what you typed.

A.2 COMPILATION ERROR MESSAGES

The following sections list the fatal compilation error messages (Section A.2.1) and the compilation warning messages (Section A.2.2), which BASIC may print when you are compiling a program. Under each message is a brief explanation further clarifying the meaning of the message.

DIAGNOSTIC MESSAGES

A.2.1 Compilation Error Messages (Fatal)

? ALLOW clause not supported for ACCESS SCRATCH

You cannot have simultaneous access to a file when you OPEN with ACCESS SCRATCH. Take out the ALLOW clause from the OPEN statement or change the ACCESS clause.

? Argument types don't match

The arguments differ in quantity or type from those defined for the function or subprogram. Check the function definition or SUB statement, and change the arguments accordingly.

? Array previously dimensioned at line n

You have already dimensioned this array at the line specified by n.

? Array too big

The array exceeds the size allowed in memory. Redimension the array to conform to size restrictions.

? Array(s) too big in program segment

The program segment is either MAIN PROGRAM, a subprogram name, or a function name.

? Attribute is illegal for files open for output

The attribute cannot be specified when creating a file for output.

? Attribute is illegal for organization files

The attribute specified cannot be used with one of the three organizations: sequential, relative, indexed.

? Cannot assign numeric result to string array

Change the numeric value to a string.

? CHANGES are not allowed for primary keys

? COMMON area declared in line n longer than specified in line n

The first declaration of a COMMON area in a program must be the largest.

? COMMON or MAP area previously declared in line n

Only one declaration for a COMMON area can appear in a program segment and cannot be the same name as a MAP buffer.

? DATA must be the first statement in line

The DATA statement must be the only statement in the line.

? DEF and DEF* both in this main program or SUB

You can have DEF's or DEF*'s in a program or subprogram but not both in the same program or subprogram.

DIAGNOSTIC MESSAGES

? Extra comma found

Take out the extra comma in the line.

? Found n when expecting n

This is a general message signalling that the compiler finds something in a line which is different from what it expects. The first n is what it finds; the second n is what it expects.

? FOR clause must precede channel number

The FOR clause must precede the channel number in the OPEN statement.

? FOR does not match NEXT

The FOR statement does not have the same variable as the NEXT statement. Change one statement or the other to match the loop.

? FOR at line n does not match NEXT

The FOR statement at the line specified by n does not match the NEXT in the program. Change one statement or the other.

? FOR must have upper bound or conditional

The FOR statement is missing an upper bound or conditional expression.

? FOR without NEXT on line n

The FOR statement on line n does not have a corresponding NEXT statement.

? Function previously defined at line n

You used the same name in two function definitions. The first function definition occurs on line n.

? IF contains keyword without NEXT

The IF statement contains the FOR, WHILE, or UNTIL statement without the NEXT statement. Add the NEXT statement.

? Illegal ACCESS

This ACCESS is not allowed for this particular file.

? Illegal array specifier

The array name is not formatted correctly.

? Illegal decimal point found

A decimal point was found in an integer or in the middle of keyword. Take out the decimal point.

? Illegal element in READ list

You specified an invalid variable reference in the READ list.

DIAGNOSTIC MESSAGES

? Illegal element in PRINT list

You specified an invalid character in the PRINT list.

? Illegal exponent for numeric constant

The exponent is too large or too small for the numeric constant.

? Illegal expression found when looking for line number or restricted statement

BASIC was looking for a line number or statement but found an expression. Change the line.

? Illegal FOR clause

This FOR clause is not valid in the OPEN statement.

? Illegal INPUT list element

A character in the INPUT statement list is not valid.

? Illegal integer modifier

Illegal usage of % terminator.

? Illegal line number

The number you specified is not in the valid range of 1 to 99999.

? Illegal MAP buffer name

The name you specified for the record buffer is not valid.

? Illegal mode mixing

An attempt was made to mix string and numeric operations.

? Illegal name for user defined function

The name used for the user-defined function does not begin with FN, or has too many characters in it.

? Illegal numeric constant

The numeric constant is not within the range allowed.

? Illegal OPEN attribute

One of the attributes in the OPEN statement is invalid.

? Illegal operand in expression

The operand you specified is incorrect. Check the line and replace it.

? Illegal organization

The file you are trying to OPEN has a different organization than the one you specified.

? Illegal SUB redefinition

You cannot redefine a subprogram.

DIAGNOSTIC MESSAGES

? Illegal to create array name from immediate mode

You cannot dimension or create an array from immediate mode.

? Illegal use of array specifier

You cannot specify an array in this context.

? Illegal variable name

The variable name is a reserved word (See Appendix D.), or an invalid character was specified within the name.

? IMAGE must be the first statement in line

The IMAGE statement must be the only statement on the line.

? Integer constant required

You specified something other than an integer constant.

? Integer expression required

An integer expression is needed rather than a string or numeric expression.

? Integer variable required

An integer variable is needed rather than a string or numeric variable.

? Line n is not a MAP

You referenced a line number in a record operation that is not a MAP statement.

? Line n is not data

The line specified by n is not a DATA statement.

? Line n is not image

The line specified in the PRINT USING statement is not an IMAGE statement.

? Line n not compiled

The line specified by n is not compiled.

? Line n not found

The line specified by n is not in the program.

? Literal Too long

The literal exceeded the maximum number of characters.

? MAP area longer than specified at line n

The buffer is larger than specified at line n.

? MAP must be specified

A MAP clause must be specified in this OPEN statement.

DIAGNOSTIC MESSAGES

? MAP or RECORDSIZE must be specified

You must include a MAP or RECORDSIZE specification in the OPEN statement.

? Maximum string length exceeded

The string length must be 2**18-1 or less.

? Missing "," in READ list

One of the commas is missing between variables in the READ list.

? Missing "," OR ";"

Add the missing punctuation point.

? Missing comma

Add the missing comma to the syntax.

? Missing file specifier

You forgot the file number associated with the OPEN file.

? Missing operand in expression

A necessary part of the expression is missing.

? Multi-statement COMMON area declaration out of order

Multiple COMMON statements which declare the same COMMON area must appear together with no intervening statements.

? name has been previously referenced or declared

A variable (name) declared in COMMON must appear in a COMMON statement prior to any reference.

? name is not a MAP buffer

The name you specified to identify the map is not the map buffer.

? name WAS MAPPED DIFFERENTLY AT LINE n

Name is the buffer name.

? name WAS REFERENCED BEFORE IT WAS MAPPED

A MAP statement must appear in a program before any data in the record is referenced.

? Next with variable(s) not allowed with statement at LINE n

Statement is either WHILE or UNTIL.

? NEXT without FOR, WHILE, or UNTIL

A NEXT statement was found without a corresponding FOR, WHILE, or UNTIL statement.

? Not implemented in this version

You specified something that is not currently in this version of the software.

DIAGNOSTIC MESSAGES

? Statement NOT IN THIS VERSION

The statement printed is not valid in this version of BASIC.

? Null array list specified

You forgot the list after the ARRAY name.

? Null INPUT list

You forgot the list of variables with the INPUT statement.

? Null READ list

You forgot the list of variables with the READ statement.

? Numeric variable required

You must specify a numeric variable rather than an integer or string variable.

? OPEN attribute was previously specified

This attribute has already been specified in the OPEN statement.

? Operand(s) must be numeric array(s)

The operand in this statement must be a numeric array.

? Operand(s) must be string array(s)

The operand in this statement must be a string array.

? Organization must precede attributes

The ORGANIZATION clause must precede the attributes in the OPEN statement.

? Primary key must be specified

A primary key clause must appear in the OPEN statement for an indexed file.

? Program too big

The program exceeds limits set by system.

? Reference to undefined function name at line n

You referenced a function that has not been defined.

? Reference to undefined function name from immediate mode

The function referenced has not been defined in the current context of immediate mode.

? Reference to undefined SUB name from immediate mode

The subprogram referenced has not been defined in the current context of immediate mode.

? Single line functions cannot be recursive

You cannot define a single line function to call itself.

DIAGNOSTIC MESSAGES

? Statement illegal in immediate mode

The following statements are invalid in immediate mode: DATA, DEF, DEF*, DIM, END, FNEND, IMAGE, MAP, SUB, SUBEND.

? Statement is illegal when program is not compiled

You have issued an IMMEDIATE MODE statement in the wrong context.

? Statement NOT IN THIS VERSION

The statement printed is not recognized in BASIC-PLUS-2.

? Statement not legal inside IF

The statement specified is not a restricted statement.

? Statement not allowed in SUB

A statement that affects transfer outside the body of a subprogram is not allowed.

? Statement not recognized

You misspelled the statement. Type it again.

? Statement outside of program

You referenced a statement that is not in the program.

? Statement WITHOUT NEXT AT LINE n

The statement (FOR, WHILE, or UNTIL) does not have a corresponding NEXT statement.

? String expression not allowed

Use a numeric or integer expression.

? String expression required

Use a string expression instead of a numeric or integer expression.

? String variable required

Use a string variable instead of an integer or numeric variable.

? SUB must be the first statement in line

The statement must be the first or only statement in a multi-statement line.

? SUB name undefined, last referenced at line n

The subprogram referenced at line n has not been defined.

? SUBEND without matching SUB statement

There is a SUBEND statement in the program but not SUB statement.

DIAGNOSTIC MESSAGES

? SUBEXIT not allowed in DEF

The SUBEXIT statement is not allowed inside a multi-line function definition.

? SUBEXIT not inside SUB

A SUBEXIT statement was found outside the subprogram. This is not valid.

? Subscript must be numeric or integer

A subscript cannot be a string.

? This statement illegal outside a multi-line DEF

This statement is not allowed outside a multi-line function definition.

? This statement is illegal inside a multi-line DEF

This statement is not allowed inside a multi-line function definition.

? Too many decimal points found in numeric constant

Only one decimal point is allowed in a numeric constant.

? Too many digits in numeric constant

You have exceeded the limit for digits in a numeric constant.

? Too many immediate mode statements - use SCRATCH RESET

You have exceeded the limit of immediate-mode statements allowed. Use SCRATCH RESET to recover.

? Too many immediate mode variables or temporaries - use SCRATCH IMMEDIATE or do a CONTINUE

You have assigned values to too many immediate-mode variables. Use SCR IMM or CONT to recover.

? Too many literals

You specified too many literals.

? Too many subscripts

The array is 1-dimensional and you specified two dimensions.

? Transfer into function name

A transfer into a multi-line DEF is invalid.

? Transfer into name at line n - MODE DEF * command was not given

You transferred into a DEF* at line n without changing the compile mode.

? Transfer into name - MODE DEF * command was not given

You transferred into DEF * name without changing the compile mode.

DIAGNOSTIC MESSAGES

? Transfer out of function name

A transfer out of a multi-line DEF is invalid.

? Transfer out of name at line n

Name is a subprogram or function name at the specified line.

? Unmatched parentheses

An open parenthesis was found without the closing parenthesis or vice versa.

? Variable name too long

The variable name has more than 35 characters.

? Variable name declared twice in name

Name is either MAIN PROGRAM, a subprogram, or a function.

? Wrong number of arguments

The number of arguments specified for a subprogram or function is incorrect.

? Wrong number of dimensions

Array has two dimensions when expecting one or vice versa.

A.2.2 Compilation Error Messages (Warning)

%"Blank" COMMON area will not be preserved across a CHAIN

The "blank" COMMON area must be the first COMMON area declared in the program if it is to be preserved across a CHAIN statement. This warning can otherwise be ignored.

%Illegal character in input stream-ignored

BASIC ignored an invalid character found in the input stream.

%Missing END statement inserted

When no END statement is in the program, BASIC inserts one internally.

%Missing FNEND inserted

BASIC inserted the FNEND statement internally to end the function DEF.

%Missing SUBEND statement inserted

BASIC inserted the SUBEND statement internally to end the SUB.

%Number too big

The number used was larger than +235-1 for integers or 1.7x1038 for numeric.

DIAGNOSTIC MESSAGES

%Number too small

The number used was smaller than -235 for integers or $.14 \times 10^{-38}$ for numeric.

%Reference to undefined line n not changed

While executing a RESEQUENCE command, BASIC found an undefined line number which will remain unchanged after the RESEQUENCE.

%Reference to uninitialized immediate mode variable variable name

You must assign a value to an immediate-mode variable before you can reference it.

%Unterminated literal-treated as a comment

The closing quotation mark is missing. BASIC treats it as a comment rather than a literal.

A.3 EXECUTION MESSAGES

The messages that BASIC prints during program execution are divided into two sections:

- A.3.1 Trappable Error and Warning Messages
- A.3.2 Nontrappable Error and Warning Messages

A.3.1 Trappable Error and Warning Messages

This section lists the trappable error and warning messages. The error messages are preceded by a question mark. The warning messages are preceded by a percent sign.

- ? 2 Filename cannot be longer than 39 characters
- ? 2 Illegal filename
- ? 2 Illegal file type attempted
- ? 2 Invalid character in file name
- ? 2 Illegal generation number attempted
- ? 3 File already open
- ? 4 No room for user on device
- ? 5 File not found
- ? 5 No such filename found
- ? 5 No such file type found
- ? 5 File has been expunged
- ? 5 File not found because output only device
- ? 6 Not supported device

DIAGNOSTIC MESSAGES

- ? 7 I/O channel already open
- ? 7 File not closed
- ? 8 Device not available
- ? 9 Channel n not assigned
- ? 9 File is no longer open
- ? 11 EOF found
- ? 11 End of file found on INPUT
- ? 13 Data error on file
- ? 14 Device hung or write locked
- ? 15 Keyboard WAIT exhausted
- ? 16 Name already exists
- ? 17 Too many files open on unit
- ? 28 ^C
- ? 30 Device not file structured
- ? 31 Illegal byte count for I/O
- ? 43 Virtual array not on disk
- ? 44 Matrix or Array too big
- ? 45 Virtual Array is not open
- ? 46 Illegal I/O channel
- ? 47 Input line too long
- % 48 Floating overflow or underflow
- ? 50 Numeric encountered during string input
- ? 51 Integer error
- ? 51 Invalid integer
- ? 52 Invalid floating point number
- ? 52 Invalid integer
- ? 53 Attempt to take LOG of negative argument
- ? 54 Attempt to take SQR of a negative argument
- ? 55 Subscript out of bounds
- % 56 Cannot invert Matrix
- ? 57 End of data found
- ? 58 ON statement out of range

DIAGNOSTIC MESSAGES

- ? 59 Insufficient data
- ? 60 FOR loop index overflow
- % 61 Integer overflow or divide by zero
- ? 104 RESUME when no error
- ? 124 Array dimensions must match
- ? 124 Array must be two dimensional
- ? 124 Array redimensions too large
- ? 124 Illegal DIMENSION specifier
- ? 128 Not ANSI "D" record format
- ? 130 Illegal KEY value change
- ? 131 UPDATE or DELETE not preceded by FIND or GET
- ? 132 Record does not exist
- ? 133 Illegal usage for device
- ? 134 Duplicate KEY
- ? 135 Illegal ACCESS for file
- ? 135 Illegal ACCESS for file name
- ? 136 File must be OPEN for MODIFY or READ
- ? 136 File must be OPEN for MODIFY or SCRATCH
- ? 136 File not OPENed with WRITE or MODIFY ACCESS
- ? 137 Illegal combination of KEY attributes
- ? 138 File n is LOCKED
- ? 139 Illegal file options
- ? 140 INDEX not initialized
- ? 141 Must be RECORD file for MOVE
- ? 141 Operation name cannot be performed on file
- ? 141 Only terminal format or SEQUENTIAL files may be SCRATCH
- ? 142 Improper record encountered on file n
- ? 143 n is an illegal RECORD number
- ? 144 Incorrect KEY of reference
- ? 145 Incorrect KEY size
- ? 146 Invalid tape label
- ? 147 Maximum record number exceeded

DIAGNOSTIC MESSAGES

- ? 148 Invalid maximum record size
- ? 149 Not at end of file n
- ? 150 No primary KEY
- ? 151 Incorrect value of POS field
- ? 152 Invalid record access
- ? 153 Record already exists
- ? 154 Record n is LOCKED
- ? 155 RECORD NOT FOUND
- ? 155 Record n not found
- ? 156 Invalid record size
- ? 157 Record too big
- ? 158 KEYS out of sequence
- ? 159 Incorrect value of SIZ field
- ? 160 Mismatched attributes
- ? 161 MOVE statement goes past end of buffer
- ? 162 Cannot OPEN file
- ? 163 No file name given
- ? 164 Terminal format file required
- ? 164 Terminal format fie required
- ? 165 Cannot position to end of file n
- ? 166 Negative FILL or string length not allowed
- ? 229 Unterminated literal in EDIT\$ string
- ? 230 No fields in IMAGE string
- ? 231 Illegal string IMAGE
- ? 232 Null IMAGE
- ? 233 Illegal numeric IMAGE
- ? 234 Numeric IMAGE specified for a string
- ? 235 String IMAGE specified for a numeric
- ? 237 First argument to SEG\$ greater than second
- ? 238 Arrays must be same size and dimension
- * 239 Array should be square
- ? 239 Matrix must be square

DIAGNOSTIC MESSAGES

- ? 240 Cannot change array dimensionality
- ? 241 Overflow
- % 241 Overflow
- ? 242 Underflow
- % 242 Underflow
- ? 247 RETURN with no GOSUB statement encountered
- ? 249 Argument to function out of range
- ? 249 Argument to TIME function out of bounds
- ? 250 Not implemented for virtual arrays
- ? 250 Argument to TIME function meaningless on a DECsystem-20
- ? 400 Invalid record
- ? 401 File must be OPENed for exclusive ACCESS
- ? 402 File not OPENed with ACCESS WRITE or MODIFY
- ? 403 Illegal to RESTORE virtual array
- ? 404 Sequential file may not be accessed by RELATIVE record number
- ? 407 READ/RESTORE when no DATA statements exist
- ? 410 Illegal character in output
- ? 411 Invalid character in input stream
- ? 413 Directory access privileges required
- ? 414 Logical name loop detected
- ? 416 Disk quota exceeded
- ? 417 Non-fatal OTS error at line n
- ? 421 ^C
- ? 422 Files must be on same device
- ? 423 Illegal to rename file to itself
- ? 424 TAN of PI over 2
- ? 427 Multiplication will not produce correct results
- ? 428 Found end of line when expecting backslash
- ? 432 File not OPENed with ORGANIZATION VIRTUAL
- ? 434 Cannot rename files to or from this device
- % 436 Too much data present - ignored
- ? 438 Directory Full

DIAGNOSTIC MESSAGES

- ? 439 Illegal to open channel 0
- ? 441 unterminated literal in edit\$ string
- ? 442 Cannot CLOSE file
- % 444 BUCKETSIZe greater than 1-- 1 being used
- ? 446 Maximum string length exceeded
- ? 447 Negative number raised to non-integer power
- ? 448 No such directory exists

A.3.2 Nontrappable Error and Warning Messages

This section lists the nontrappable error and warning messages. The error messages are preceded by a question mark. The warning messages are preceded by a percent sign.

- ? 1 Bad directory for device
- ? 10 Protection violation
- ? 12 Fatal system I/O failure
- ? 35 Free storage memory management violation
- ? 49 Argument too large in expression
- ? 62 No object time system
- ? 88 Incorrect argument type passed to n
- ? 88 Invalid argument type found
- ? 88 Wrong type of arguments to function
- ? 89 Wrong number of arguments to function
- ? 104 Illegal RESUME
- ? 123 STOP
- ? 126 No more memory available
- ? 236 Time limit exceeded
- ? 243 Chain to non-existent line number
- ? 250 Unimplemented built-in function referenced
- ? 250 Not implemented on this machine in this version
- ? 405 RETURN before RESUME during ONERROR processing
- ? 406 Incorrect number of arguments passed to n
- ? 408 Inferior fork has terminated
- ? 418 No more room for stack

DIAGNOSTIC MESSAGES

- ? 419 Array too big or no more room for stack
- ? 420 ^C
- ? 425 Illegal memory write
- ? 426 System resources exhausted
- ? 429 Unknown .PDL overflow
- ? 431 Invalid punctuation in PRINT/INPUT statement
- ? 433 Illegal memory read
- ? 435 Must use RESUME to leave ONERROR processing
- ? 437 error n occurred while processing error n
- ? 443 Compiler required for CHAIN to source file
- ? 445 Illegal to transpose matrix to itself
- % "BLANK" COMMON area not preserved across CHAIN due to size inconsistency
- % EXE file not compatible with this BASOTS, program must be rebuilt
- % Old BASIC not compatible with this BASOTS, must use current BASIC
- % Incompatible version of BASOTS being used, must use current version

APPENDIX B

USING THE TOPS-20 OPERATING SYSTEM

This appendix provides you with the essential information necessary for communicating with TOPS-20 operating system. Refer to the TOPS-20 User's Guide for additional information on other commands and programs.

This appendix is divided into the following sections:

- B.1 Contacting the TOPS-20 Operating System
- B.2 If the System Stops
- B.3 Logging Off the System
- B.4 File Specifications

B.1 CONTACTING THE TOPS-20 OPERATING SYSTEM

To signal the TOPS-20 operating system to start processing your commands, type a CTRL/C (^C). If your terminal is connected to the TOPS-20 operating system by a telephone line, typing CTRL/C is not necessary because the system automatically gives a CTRL/C when it answers the telephone.

After you type CTRL/C, the system responds with an identification message and then prints an @ on the next line.

```
2102 Development Sys., TOPS-20 Monitor 4(2505)
@
```

The @ is a prompt character telling you the system is ready for you to type any valid system command. Once the system prints the @ character, you can give the log in command. However, before you can log in to the system, you need to obtain three items from the computer administration staff. These items are:

1. User name
2. Password
3. Account Descriptor

These three items uniquely identify you to the system so that you can receive storage space and be charged appropriately for the use of the computer.

USING THE TOPS-20 OPERATING SYSTEM

B.1.1 What is a Job?

In timesharing, the term job refers to the entire time you interact with the computer (from LOGIN to LOGOUT). You create a job when you successfully complete LOGIN. If in trying to create a job (LOGIN) you give an invalid user name, password, or account descriptor, the system prints a message and prevents you from continuing. After you successfully create a job, you can access any system resource to which you are entitled. (You can also submit a job through batch. See Appendix C, Running BASIC-PLUS-2 Through BATCH.)

USER NAME

Each user is identified to the system and to other users by a user name. A user name consists of up to 39 alphanumeric characters (including hyphens and periods) and is usually your surname. A user name identifies your job, your logged-in directory, and any messages that you send and receive.

PASSWORD

Because your user name is known to many TOPS-20 operating system users, it does not provide the security necessary in a large data processing system. Each user, therefore, selects a password (consisting of up to 39 alphanumeric characters, including hyphens) that is a secret between the system manager and the user. You must supply your password when you log in.

Whenever you type your password, it is not printed on the terminal. This prevents other people from learning it and logging into the system without proper authorization.

ACCOUNT DESCRIPTOR

Your account descriptor is charged with all bills that you accumulate by using the system. An account descriptor consists of up to 39 alphanumeric characters. Some users, however, are restricted to using numeric accounts.

Once you log in to the system, all charges are made to the account you gave in the LOGIN command. These charges include bills for time on the central processor unit (CPU) and for file storage.

B.1.2 Logging In (LOGIN)

To log in to the system, first type CTRL/C. After the @ appears, do the following:

1. Type the letters LOG and press the ESCape key (labeled **ESC**). The computer responds by printing IN (USER).

@LOGIN (USER)

2. Type your user name and press the **ESC** key. The computer responds by printing (PASSWORD). The example is for the user FORREST.

@LOGIN (USER) FORREST (PASSWORD)

USING THE TOPS-20 OPERATING SYSTEM

3. Now type your password and press the `(ESC)` key. (Since your password is a secret, it does not appear on your terminal.) After you press the `(ESC)` key, the computer responds with (ACCOUNT #).

```
@LOGIN (USER) FORREST (PASSWORD) (ACCOUNT #)
```

4. Finally, type your account number and press the RETURN key.

```
@LOGIN (USER) FORREST (PASSWORD) (ACCOUNT #) 341 (RET)
```

If your user name, password, and account descriptor are valid, the system types a message similar to the following:

```
Job 41 on TTY222 27-Mar-79 11:04:43  
@
```

The following example shows the entire LOGIN process. Since the `(RET)` and `(ESC)` keys do not print a character on the terminal, a dollar sign (\$) indicates where you press the `(ESC)` key, and `(RET)` indicates where you press the RETURN key.

```
2102 Development Sys., TOPS-20 Monitor 4(2505)  
@LOGIN$(USER) FORREST$(PASSWORD) $(ACCOUNT #) 341 (RET)  
Job 41 on TTY222 27-Mar-79 11:04:43  
@
```

At this point you are ready to access BASIC.

B.2 IF THE SYSTEM STOPS

The system may unexpectedly stop due to an error. In such a situation, your terminal does not respond to what you type. If recovery is possible, the system prints:

```
DECSYSTEM-20 CONTINUED
```

You can then continue the tasks you were doing before the system stopped.

In situations where a fatal error occurs, your terminal stops printing and the system prints the message:

```
%DECSYSTEM-20 NOT RUNNING
```

Shortly, the system restarts, printing:

```
System restarting, wait . . .
```

then

```
SYSTEM IN OPERATION
```

At this point, type a CTRL/C and log in to the system. Depending on the operation you were performing, some of the files you were using may be incomplete or missing. Contact the operator or system manager if you want to restore a file that had been saved on a system backup tape. Many systems make backup tapes at the end of each day, so make sure you contact the operator before needlessly re-entering a lost file.

USING THE TOPS-20 OPERATING SYSTEM

B.3 LOGGING OFF THE SYSTEM

If you want to leave the computer while still at system command level, type:

```
@LOGO
```

The LOGOUT command expunges deleted files in your connected directory, checks its disk storage allocation, ends your job, and prints a message. For example:

```
@LOGO
Killed Job 41, User FORREST, Account 341, TTY 222,
  at 27-Mar-79 11:05:16, Used 0:0:0 in 0:0:32
```

You can also log off the system while in BASIC command level. See Section 2.1.4.

B.4 FILE SPECIFICATIONS

TOPS-20 keeps all system programs, user programs, and data stored in files. When you create a BASIC program or data file, you must give a unique identifier, known as a file specification, to distinguish your file from everything else on the system.

The most complete form of file specification is:

```
dev:<dir>name.type.gen;attribute
```

where:

dev:	is the device name of the storage device, 1 to 6 alphabetic characters. Table B-1 lists the TOPS-20 device names.
<dir>	is a directory name, 1 to 39 alphanumeric characters.
name	is the name of the file or program, 1 to 39 alphanumeric characters.
.typ	is the file type, 0 to 39 alphanumeric characters.
.gen	is the generation number, a positive integer from 1 to 131,071.
;attribute	is a modifier for the file and specifies a distinctive characteristic for the file.

The items in a file specification are organized from the most general to the most specific. Proceeding from left to right in the above definition: the device name specifies the particular device; the directory specifies an area on that device; the filename specifies a particular file in that directory; the file type specifies the contents of the file (for example, a program, data, text); and the generation number specifies how many times the file has been changed.

USING THE TOPS-20 OPERATING SYSTEM

DEV:

A device name designates the system storage device that contains, or will contain, the file. (A logical name can replace a device name; refer to the TOPS-20 User's Guide.)

A device name consists of 1 to 6 alphabetic characters that indicate the type of device, a number that specifies a particular device (when more than one of a particular device are available), and a colon that terminates the device name.

The system devices available for use with BASIC are listed in Table B-1.

Table B-1
System Device Names

Device	Device Name
Public Structure	PS:
Disk	DSK:
User's Terminal	TTY:
A Particular Terminal	TTYn:
Any Line Printer	LPT:
A Particular Line Printer	LPTn:

The n indicates a particular device when more than one are available. In all cases, the colon must terminate the device name. For example:

TTY20: is the terminal connected to line 20.

LPT: is a line printer.

If you omit a device name from a file specification, the system uses your currently connected device as the default. At system command level, you can also store BASIC programs on magnetic tape. For information on this system process, refer to the TOPS-20 User's Guide.

DIR

The area of disk storage allocated for your use is called your logged-in directory. It is referenced by a directory name that is your user name enclosed in angle brackets (<>). Thus, if you are user LEVINE, you would have a directory named <LEVINE>.

A directory name consists of up to 39 alphanumeric characters including hyphen, dollar sign, and underline. (The characters percent and asterisk can be used to specify a group of directories, but these characters cannot actually be part of a directory name.)

Directory names are always enclosed in angle brackets, for example:

<MASELLA>
<CRONIN>
<LIBRARY>

USING THE TOPS-20 OPERATING SYSTEM

NAME

Each file has a file name consisting of up to 39 alphanumeric characters including hyphen, dollar sign, and underline. (The characters percent and asterisk can be used to specify a group of file names, but cannot actually be used in a file name.) For example:

```
TEST
LINDRW
SPRWAC
A-B$C.
```

You should be aware, however, that not all software components support the full length of 39 characters. Refer to the TOPS-20 User's Guide.

.TYP

To indicate the contents of a file or to give the same file name to more than one file, specify a file type consisting of a period followed by up to 39 alphanumeric characters, including hyphen, dollar sign, and underline.

The file types in BASIC are:

```
.B20    designating a source file
.CMD    designating a command file
.EXE    designating a compiled file
```

.GEN

A generation number reflects approximately the number of times you have changed your file. A generation number can be any positive integer up to and including 131,071 ($2^{17}-1$ or the largest number that can be represented in 17 binary bits).

Whenever you type a file specification (NEW, OLD, SAVE), you can include a generation number. At times you may have more than one generation of a file. The system always assumes that the most recent file is the one with the highest generation number.

;ATTRIBUTES

File attributes specify distinctive characteristics for a file specification. More than one attribute may appear in a file specification. The three attributes you may specify are: ;A for account, ;P for protection, and ;T for temporary.

The account descriptor takes the form:

```
;Adescriptor
```

The descriptor is an account name of up to 39 alphanumeric characters designating a valid system account. All charges for file storage will be billed to this account. If you do not specify an account, the system bills the account you specified in the LOGIN command or you last SET ACCOUNT command.

USING THE TOPS-20 OPERATING SYSTEM

The file protection code takes the form:

```
;Pprotection
```

After typing ;P, specify a valid TOPS-20 protection code. Refer to Section B.4.1 for a description of protection codes.

A temporary file specification contains the file descriptor ;T and a generation number of 10000 plus the number of the job that created the file. Temporary files are automatically deleted when you log off the system.

For additional information on file attributes, see the TOPS-20 User's Guide.

B.4.1 Protection Codes

The TOPS-20 file system is flexible and convenient. It allows you to protect your files and yet grant access to other users who need them. The access to each file and directory is determined by a protection code number.

Each directory protection number and file protection number consists of six digits divided into three fields of two digits each. The first 2-digit field specifies the owner's access, the second 2-digit field specifies the group members' access, and the last 2-digit field specifies all other users' access.

Any user's access to a file is subject first to the directory protection, then the protection on the individual file. Generally, the default for both files and directories is 777700. This code allows only users in your group to access your files. Protection codes need be specified only when the protection assigned by default is not sufficient. For more information on sharing directories and groups, see the TOPS-20 User's Guide.

APPENDIX C

RUNNING BASIC-PLUS-2 THROUGH BATCH

Many data processing jobs require long running times and make few demands of the user. Ideally these jobs should be run in the user's absence when the computer is not busy with other tasks. This ideal is readily met by the batch system on the TOPS-20 operating system.

Batch is a group of programs that allows you to submit a job to the TOPS-20 batch system on a leave-it basis. You may build and submit your job by entering data directly to an interactive computer system through a terminal or by submitting a card deck to the operator. (For more information on using card decks, refer to the Batch Reference Manual or Getting Started With Batch.)

Although batch allows you to submit your job to the computer through a timesharing terminal, output is never returned at a timesharing terminal. Instead, it is sent to a peripheral device (normally the line printer) at the computer site and returned to you in the manner designated by the installation manager.

You must make up a control file to use batch. A control file is a list of commands for the system, system programs, BASIC-PLUS-2 programs, or for batch itself that tells what steps to follow to process your job.

After it has been submitted, the job waits in a queue with other jobs until batch schedules it to run under guidelines established by the installation manager. Some factors that affect how long your job waits in the queue are its size, the amount of memory it needs, and the amount of execution time required. When the job is started, batch reads the control file to determine what actions are necessary to complete the job.

As each step in the control file is performed, batch records it in a log file. The system response is also written in the log file. Any response from your job that would be written on the terminal during timesharing is written in the log file by batch.

C.1 CREATING A CONTROL FILE

When you enter a job to batch from a timesharing terminal, you must create a control file that batch can use to run your job. The control file contains all the commands that you would use to run your job if you were running under timesharing.

RUNNING BASIC-PLUS-2 THROUGH BATCH

To create a control file and submit it to batch, do the following:

1. LOGIN to the system as a timesharing user.
2. Write a control file on disk using EDIT.
3. Submit the job to batch using the system command SUBMIT.

You can then wait for your output to be returned at the designated place.

Since you can put system and batch commands as well as BASIC-PLUS-2 commands in the control file, you have to tell batch what kind of command it is reading. To include a system command or batch command, you must put an at sign (@) in the first column and follow it immediately with the command. For example:

```
@BASIC
```

This is the system command that establishes contact with the BASIC-PLUS-2 compiler.

A BASIC-PLUS-2 command need not be preceded by a prompt character. Start the BASIC-PLUS-2 command in the first column of the terminal line. For example:

```
RUN
```

C.2 SUBMITTING A JOB TO BATCH

After you have created the control file and saved it on disk, you must enter it into the batch queue so that it can be run. All programs and data that are to be processed when the job is run must be made up in advance or be generated during the running of the job.

You enter your job in the batch queue with the SUBMIT system command. The SUBMIT command has the following format:

```
SUBMIT jobname=control file.typ/switches,log file.typ/switches
```

where:

jobname	is the name you give to the job. It can be omitted. The default is the control filename.
control file.typ	is the name and type of the control file. You must specify the name of the control file. The default type is .CTL.
log file.typ	is the name and type batch gives to the log file. If the name of the log file is omitted, batch uses the control file name. The default type is .LOG.
/switches	are switches telling batch how to process the job. Most switches can appear anywhere in the command string; however, a few must be placed after the files to which they pertain.

RUNNING BASIC-PLUS-2 THROUGH BATCH

Three kinds of switches are available for use with the SUBMIT command: queue operation, general, and file control. For information on these switches and other batch facilities, refer to the Batch Reference Manual.

C.3 A BATCH EXAMPLE

The following is a BASIC-PLUS-2 program saved in a file on disk in a file named MYBAS.B20:

```
LISTNH
00010 INPUT D
00020 IF D = 2 THEN 110
00030 PRINT "X VALUE","SINE","RESOLUTION"
00040 FOR X=0 TO 3 STEP D
00050 IF SIN(X) = M THEN 80
00060 LET X0=X
00070 LET M = SIN(X)
00080 NEXT X
00090 PRINT X0,M,D
00100 GOTO 10
00110 END
```

READY

This program requests data from the user. You include the data in the control file. The final data item must be 2 to conclude the program. The control file follows:

```
@TYPE BASIC.CTL
@BASIC
OLD MYBAS.B20
RUN
.1
.01
.001
2
MON
@
```

The command to submit the job to batch is as follows:

```
@SUBMIT BASIC.CTL
INP:BASIC=/Seg:6489/Time:0:05:00
@
```

The output from the program will be printed as part of the log file listing.

NOTE

If you turn off CTRL/C trapping (which happens to every job run through batch), you receive a warning message to that effect in the log file:

```
*CAN'T TRAP CONTROL C, USE CONTROL A
INSTEAD
```

RUNNING BASIC-PLUS-2 THROUGH BATCH

This message is repeated every time you use the RUN, DEBUG, START, or SCRATCH ALL command in a BASIC program submitted through batch.

The following log file was produced from this job:

@TYPE BASIC.LOG

```

09:40:56 BAJOB  BATCON version 103(3002) running BASIC sequence 6489 in stream 1
09:40:56 BAFIL  Input from PS:<FORREST>BASIC.CTL.1
09:40:56 BAFIL  Output to PS:<FORREST>BASIC.LOG
09:40:56 BASUM  Job Parameters
                    Time:00:05:00  Unique:YES  Restart:NO  Output:LOG
09:40:56 MONTR  SYSTEM 2116 THE BIG ORANGE WELCOMES YOU, TOPS-20 Monitor 3A(2013)
09:40:57 MONTR  System shutdown scheduled for 26-Jun-79 18:00:00,
09:40:57 MONTR  Up again at 27-Jun-79 00:00:00
09:40:57 MONTR  @SET TIME-LIMIT 300
09:40:57 MONTR  @LOGIN FORREST 1
09:41:01 MONTR  Job 23 on TTY223 25-Jun-79 09:41:01
09:41:01 MONTR  @
09:41:01 MONTR  CONNECTED TO PS:<FORREST>
09:41:01 MONTR  @BASIC
09:41:03 USER  * CAN'T TRAP CONTROL C, USE CONTROL A INSTEAD
09:41:04 USER  * CAN'T TRAP CONTROL C, USE CONTROL A INSTEAD
09:41:05 USER
09:41:05 USER  READY
09:41:05 USER  OLD MYBAS.B20
09:41:06 USER
09:41:06 USER  READY
09:41:06 USER  RUN
09:41:06 USER
09:41:06 USER  MYBAS.B20
09:41:06 USER  Monday, June 25, 1979 09:41:06
09:41:06 USER
09:41:07 USER  * CAN'T TRAP CONTROL C, USE CONTROL A INSTEAD
09:41:08 USER  ? .1
09:41:08 USER  X VALUE      SINE      RESOLUTION
09:41:08 USER  3            0.1411201  0.1
09:41:08 USER  ? .01
09:41:08 USER  X VALUE      SINE      RESOLUTION
09:41:08 USER  2.999999    0.1411205  0.01
09:41:08 USER  ? .001
09:41:08 USER  X VALUE      SINE      RESOLUTION
09:41:09 USER  2.99999    0.1411302  0.001
09:41:09 USER  ? 2
09:41:09 USER
09:41:09 USER  RUNTIME: 0.987 SECS          ELAPSED TIME: 0:00:02
09:41:09 USER
09:41:09 USER  READY
09:41:09 USER  MON
09:41:09 MONTR  @@
09:41:09 MONTR  @^C
09:41:10 MONTR  @LOGOUT
09:41:16 MONTR  Killed Job 23, User FORREST, Account 1, TTY 223,
09:41:16 MONTR  at 25-Jun-79 09:41:16, Used 0:0:1 in 0:0:15
09:41:17 LPDAT  LPTLSJ LPTSPL version 103(2310) running on FLPT1, 25-Jun-79 09:41:18
09:41:18 LPDAT  LPTSJS Startins Job BASIC, Seq #6489, request created at 25-Jun-79 09:41:16

```

APPENDIX D

BASIC RESERVED WORDS

Certain words in the BASIC-PLUS-2 language are reserved and, therefore, are not valid variable names. However, variations such as IF\$, AND%, DIM\$, MAT% are valid and proper variable names. Note that a space must follow all keywords. You may not recognize all the words in this list, since they are for both RSTS/E and the TOPS-20 operating system.

ABORT ACCESS% ALLOW APPEND ATN	ABS ALIGNED ALTERNATE AS ATN2	ACCESS ALL AND ASCII
BACK BIN\$ BLOCKSIZE BUCKETSIZE BUFSIZ	BEL BINARY BROADCAST BUFFER BY	BIN BIT BS BUFFERSIZE
CALL CHANGE CLK CLUSTERSIZE COMP% COS CR CVT%	CCPOS CHANGES CLK\$ COM CON COT CTRLC	CHAIN CHR CLOSE COMMON CONTIGUOUS COUNT CVT\$
DAT DATE DEL DENSITY DIM DUPLICATES	DAT\$ DEF DELETE DET DIMENSION	DATA DEF* DELIMIT DIF\$ DOUBLEBUF
ECHO EQ ERN\$ ESC	ELSE EQV ERR EXP	END ERL ERROR EXTEND
FF FILESIZE FILL% FIX FNEND FORCEIN	FIELD FILL FIND FIXED FNEXIT FROM	FILE FILL\$ FIX FNAME\$ FOR

BASIC RESERVED WORDS

GE	GET	GO
GOSUB	GOTO	GT
HANGUP	HT	
IDN	IF	IFEND
IFMORE	IMAGE	IMP
INDEXED	INIMAGE	INPUT
INSTR	INT	INV
INVALID		
KEY	KILL	
LEFT	LEFT\$	LEN
LET	LF	LINE
LINO	LINPUT	LOCK
LOCKED	LOG	LOG10
LSA	LSET	
MAGTAPE	MAP	MAR
MAR%	MARGIN	MAT
MID	MID\$	MOD
MOD%	MODE	MODIFY
MOVE		
NAME	NEXT	NOCHANGES
NODATA	NODUPLICATES	NOECHO
NONE	NOPAGE	NOQUOTE
NOREWIND	NOSPAN	NOTAPE
NOT	NUL	NUL\$
NUM	NUM\$	NUM1\$
NUM2		
OCT	OCT\$	ON
ONECHR	ONENDFILE	ONERROR
OPEN	OR	ORGANIZATION
OUTPUT		
PAGE	PI	PLACES\$
POS	POS%	PPS
PRIMARY	PRINT	PROD\$
PUT		
QUOTE		
RAD\$	RANDOM	RANDOMIZE
RCTRLC	RCTRLO	READ
RECORD	RECORDSIZE	RECOUNT
REF\$	RELATIVE	REM
RESET	RESTORE	RESUME
RETURN	RIGHT	RIGHT\$
RND		
SCRATCH	SEG\$	SEQUENTIAL
SGN	SI	SIN
SLEEP	SO	SP
SPACE\$	SPAN	SQR
SQRT	STATUS	STEP
STOP	STR\$	STREAM
STRING\$	SUB	SUBEND
SUBEXIT	SUM\$	SWAP%
SYS		

BASIC RESERVED WORDS

TAB	TAN	TAPE
TERMINAL	THEN	TIM
TIME	TIMES	TO
TRMS	TRN	TYPE\$
UNALIGNED	UNLESS	UNLOCK
UNTIL	UPDATE	USAGE\$
USING	USR\$	
VAL	VARIABLE	VIRTUAL
VPS\$	VT	
WAIT	WHILE	WITH
WRITE		
XLATE	XOR	
ZER		

APPENDIX E

ASCII CODE

Table E-1
ASCII Table

ASCII Decimal Number	Character	Meaning
0	NUL	Null
1	SOH	Start of heading
2	STX	Start of text
3	ETX	End of text
4	EOT	End of transmission
5	ENQ	Enquiry
6	ACK	Acknowledge
7	BEL	Bell
8	BS	Backspace
9	HT	Horizontal tab
10	LF	Line feed
11	VT	Vertical tab
12	FF	Form feed
13	CR	Carriage return
14	SO	Shift out
15	SI	Shift in
16	DLE	Data link escape
17	DC1	Device control 1
18	DC2	Device control 2
19	DC3	Device control 3
20	DC4	Device control 4
21	NAK	Negative acknowledgement
22	SYN	Synchronous idle
23	ETB	End of transmission block
24	CAN	Cancel
25	EM	End of medium
26	SUB	Substitute
27	ESC	Escape
28	FS	File separator
29	GS	Group separator
30	RS	Record separator
31	US	Unit separator
32	SP	Space or blank
33	!	Exclamation mark
34	"	Quotation mark
35	#	Number sign
36	\$	Dollar sign
37	%	Percent sign
38	&	Ampersand
39	'	Apostrophe

ASCII CODE

Table E-1 (Cont.)
ASCII Table

ASCII Decimal Number	Character	Meaning
40	(Left parenthesis
41)	Right parenthesis
42	*	Asterisk
43	+	Plus sign
44	,	Comma
45	-	Minus sign or hyphen
46	.	Period or decimal point
47	/	Slash
48	0	Zero
49	1	One
50	2	Two
51	3	Three
52	4	Four
53	5	Five
54	6	Six
55	7	Seven
56	8	Eight
57	9	Nine
58	:	Colon
59	;	Semicolon
60	<	Left angle bracket
61	=	Equal sign
62	>	Right angle bracket
63	?	Question mark
64	@	At sign
65	A	Uppercase A
66	B	Uppercase B
67	C	Uppercase C
68	D	Uppercase D
69	E	Uppercase E
70	F	Uppercase F
71	G	Uppercase G
72	H	Uppercase H
73	I	Uppercase I
74	J	Uppercase J
75	K	Uppercase K
76	L	Uppercase L
77	M	Uppercase M
78	N	Uppercase N
79	O	Uppercase O
80	P	Uppercase P
81	Q	Uppercase Q
82	R	Uppercase R
83	S	Uppercase S
84	T	Uppercase T
85	U	Uppercase U
86	V	Uppercase V
87	W	Uppercase W
88	X	Uppercase X
89	Y	Uppercase Y
90	Z	Uppercase Z
91	[Left square bracket
92	\	Back slash
93]	Right square bracket
94	^	Circumflex or up arrow

ASCII CODE

Table E-1 (Cont.)
ASCII Table

ASCII Decimal Number	Character	Meaning
95	← or _	Back arrow or underscore
96	/	Grave accent
97	a	Lowercase a
98	b	Lowercase b
99	c	Lowercase c
100	d	Lowercase d
101	e	Lowercase e
102	f	Lowercase f
103	g	Lowercase g
104	h	Lowercase h
105	i	Lowercase i
106	j	Lowercase j
107	k	Lowercase k
108	l	Lowercase l
109	m	Lowercase m
110	n	Lowercase n
111	o	Lowercase o
112	p	Lowercase p
113	q	Lowercase q
114	r	Lowercase r
115	s	Lowercase s
116	t	Lowercase t
117	u	Lowercase u
118	v	Lowercase v
119	w	Lowercase w
120	x	Lowercase x
121	y	Lowercase y
122	z	Lowercase z
123	{	Left brace
124		Vertical line
125	}	Right brace
126	~	Tilde
127	DEL	Delete

APPENDIX F

SUMMARY OF BASIC-PLUS-2 STATEMENTS, FUNCTIONS, AND OPERATORS

This appendix summarizes the BASIC-PLUS-2 statements, functions, and operators. For ease of reference, the upper-right corner of each statement description contains the section number(s) in the manual where that statement is described. The three summaries are contained in the following sections:

- F.1 Statements
- F.2 Functions
- F.3 Operators

F.1 STATEMENTS

CALL 5.1.1

CALL name [(actual arguments)]

200 CALL SUB1 (A,B)

The CALL statement transfers control to a specified subprogram, transfers parameters, and saves the state of the calling program. Parameters contained in the argument list must agree in type and number with the corresponding SUB statement.

CHAIN 5.2

CHAIN string [LINE line number]

15 CHAIN "SEE" LINE 70

The CHAIN statement passes control to a specified program. If no line number is specified, execution starts at the beginning of the program.

CHANGE 6.3.5

CHANGE list TO string variable

or

CHANGE {string variable } TO list
 {string expression}

25 CHANGE A TO A\$

The CHANGE statement converts a list of integers (real numbers are truncated) into a string of characters and vice versa. The length of the string is determined by the value found in element 0 of the list.

SUMMARY OF BASIC-PLUS-2 STATEMENTS, FUNCTIONS, AND OPERATORS

CLOSE 8.1.2 Terminal-Format
9.5.2 Virtual-Array
and Record

CLOSE $\left[\left[\# \right] \text{ expression(s)} \right]$

150 CLOSE #6,8

The CLOSE statement terminates I/O to a device and writes all active buffers. The number sign and file expressions are optional. When no files are specified, all files currently open are closed.

COM $\left[\text{MON} \right]$ 5.3

COM $\left[\text{(name)} \right]$ list

50 COM (TEST) A,B,C

The COMMON statement defines variables whose values are shared among program segments (for example, a main program and subprograms) connected by a SUB statement or between two programs connected by a CHAIN statement.

DATA 3.1.3.2

DATA constant(s)

50 DATA 4.3, "string", 18, 42%

The DATA statement allows you to provide a pool of information that is accessible to the program by means of a READ statement. A DATA statement must be the only statement on the line and, when you specify more than one item, you must separate them with commas. DATA statements cannot be continued.

DEF (single-line) 6.7.1

DEF FNa $\left[(b_1, b_2, b_3, \dots, b_n) \right] = \text{expression}$

10 DEF FNX (A,B)=A*B

The DEF statement establishes a user-defined function. The function name can be any valid variable name and must begin with FN. The variable type determines the function type. The optional arguments represent dummy parameters and cannot contain array elements. The function definition can refer to any of the dummy parameters or to other program variables but the definition cannot be recursive. Single-line user-defined functions are local to the main program or subprogram in which they are contained.

DEF (multi-line) 6.7.2

DEF FNa $\left[(b_1, b_2, b_3, \dots, b_n) \right], \left[(c_1, c_2, c_3, \dots, c_n) \right]$

10 DEF FNX (A,B), C

The multi-line DEF establishes user-defined functions and allows you to include other statements in the body of the function. The function name can be any variable name preceded by FN. Any statement can appear in a function except SUB, SUBEND, RETURN or another DEF. The DATA and DIM statements are not local to the function definition. A GOTO, GOSUB, ONGOTO, or ONGOSUB transfer outside the function is not allowed. The function definition must end with an FNEED statement.

SUMMARY OF BASIC-PLUS-2 STATEMENTS, FUNCTIONS, AND OPERATORS

DEF* (multi-line) 6.7.3

```
DEF* FNa [(b1,b2,b3,...bn)],[c1,c2,c3,...cn]
```

```
10 DEF* FNZ%
```

This statement has been added for compatibility with other BASICs. The distinguishing asterisks appear only in the DEF statement and not in program references to the function. The statement permits global references to function parameters and places parameter values in global locations. Transfers are allowed into and out of DEF* functions; however, random transfers can produce unpredictable results. If you transfer out of the function, be sure to transfer in and exit through the FNEED statement.

DELETE 9.6

```
DELETE file exp [,INVALID line no.]
```

```
60 DELETE PROG5, INVALID 500
```

The DELETE operation is used on relative and indexed files. The operation erases an existing record from the file. The INVALID clause shifts control to the specified line if the operation fails.

DIM[ENSION] 1.8.1

```
DIM subscripted variable(s)
```

```
30 DIM B(2,3)
```

The DIM statement reserves storage for arrays. The size of the reserved storage is determined by the subscripts (constant). A maximum of two subscripts is permitted and, when two are used, they must be separated by a comma.

DIM # 8.2.1

```
DIM # expression, array(s) [=integer]
```

```
50 DIM #2, A(10,15), B(50)
```

This statement allocates space for the specified arrays on the file associated with the logical number. Storage is allocated at the beginning of the file such that the rightmost subscript varies the fastest. The default string storage length is 16 and the space is pre-allocated.

END 4.6

```
END
```

```
100 END
```

The END statement terminates program execution and closes all files. It is optional. When used, END must be the last statement in the program.

SUMMARY OF BASIC-PLUS-2 STATEMENTS, FUNCTIONS, AND OPERATORS

FIND

9.6

```
FIND file exp [,RECORD num exp]
      [,KEY # num exp {GT
                       GE} string exp]
      [,LOCKED line no.] [,INVALID line no.]
```

```
50 FIND #7, RECORD 25, LOCKED 120
```

The FIND operation causes a RECORD search in the specified file. For sequential files, the FIND starts at the beginning of the file and locates each successor record for each FIND operation. Relative files allow the specification of a record number. Indexed files allow the specification of a key or a sequential search through the key table. The RECORD and KEY specifications are restricted to relative and indexed files.

FNEND

6.7.2
6.7.3

FNEND

```
40 FNEND
```

The FNEND statement causes an exit from a user-defined function and signals the function's logical and physical end.

FNEXIT

6.7.2
6.7.3

FNEXIT

```
70 FNEXIT
```

The FNEXIT statement is equivalent to a GOTO n, where n is the line number of the FNEND statement for the current multi-line DEF. FNEXIT is legal only inside a multi-line DEF.

FOR

4.4.1

```
FOR variable=num exp1 TO num exp2 STEP[num exp3]
```

```
25 FOR I=1 TO 5 STEP 2
```

The FOR statement initiates and controls a loop. A simple numeric variable must be used after the FOR, and the same variable must appear in the required NEXT statement. The first numeric expression is the initial loop value; the second expression is the terminating loop value. The optional STEP expression is the loop increment; +1 is the default. Transfer into an uninitialized loop is illegal.

FOR (conditional)

4.4.3

```
FOR variable=num exp1 [STEP num exp2] {WHILE} conditional exp
                                     {UNTIL}
```

```
80 FOR I=1 UNTIL I>10
```

The conditional FOR statement duplicates the previous FOR statement except that loop termination is determined by a false expression in the WHILE clause or a true expression in the UNTIL clause.

SUMMARY OF BASIC-PLUS-2 STATEMENTS, FUNCTIONS, AND OPERATORS

GET

9.6

```
GET file exp [ ,KEY # num exp {GE} string exp ]
                {GT}
                {EQ}
                [RECORD num exp]
                [LOCKED line no.] [ ,INVALID line no.]
```

50 GET #5

The GET operation reads a record from a specified file into a buffer. On sequential files, GET operations are performed on succeeding records starting at the beginning of the file. Relative files allow the specification of a record number, and indexed files allow the specification of a key name.

GOSUB

4.7.1

```
GOSUB line number
```

25 GOSUB 120

The GOSUB statement transfers control to a subroutine that begins at a specified line number.

GOTO

4.1

```
GOTO line number
```

40 GOTO 85

The GOTO statement unconditionally transfers control to a specified line number.

IF

4.3

```
IF conditional exp [ , ] {THEN} line number
                    {GOTO}
```

```
IF conditional exp [ , ] THEN statement(s)
```

```
IF conditional exp [ , ] {THEN} line number ELSE {line number}
                    {GOTO}                    {statement(s)}
```

```
IF conditional exp [ , ] THEN statement(s) ELSE {line number}
                    {statement(s)}
```

25 IF A=0 THEN PRINT "A EQUALS 0"

The various forms of the IF statement allow branches in the program. The IF statement can also cause execution of statements except the following:

DIM, REM, DATA, END, DEF, FNEND, SUB, SUBEND, WHILE, UNTIL, NEXT, and IMAGE.

SUMMARY OF BASIC-PLUS-2 STATEMENTS, FUNCTIONS, AND OPERATORS

IFEND#

8.1.7.1

```
IFEND # expression, { THEN } { statement }
                   { THEN } { line number }
                   { GOTO }
```

```
25 IFEND # 6 THEN 50
```

The IFEND # statement checks for the end of file. If the file pointer is at the end of the file, you can transfer control to another line of the program or execute a statement.

IFMORE#

8.1.7.2

```
IFMORE # expression, { THEN } { statement }
                   { GOTO } { line number }
```

```
10 IFMORE # 7 GOTO 100
```

The IFMORE # statement tests whether the file pointer is at the end of file. If NOT at the end, BASIC executes the statement or goes to the line number specified.

IMAGE

10.5

```
IMAGE unquoted string
      :unquoted string
```

```
10 :##.## ##.##
```

The IMAGE statement is used in conjunction with the PRINT USING statement. The characters following the colon, or keyword IMAGE, define the format of output. IMAGE must be the only statement on the line and cannot contain comments.

INPUT

3.1.1

```
INPUT variable(s)
```

```
25 INPUT A,B,C%
```

The INPUT statement allows you to type in data to the program from the terminal. The program requests data by printing a question mark on the terminal and then waiting for you to respond.

INPUT

8.1.3

```
INPUT [#]expression, variable(s)
```

```
25 INPUT #6, A,B,C
```

The INPUT # statement acts very much as the INPUT statement. However, the INPUT # statement requests data from a terminal-format file rather than from the terminal.

SUMMARY OF BASIC-PLUS-2 STATEMENTS, FUNCTIONS, AND OPERATORS

INPUT LINE and LINPUT 3.1.2

INPUT LINE string variable(s)
LINPUT string variable(s)

15 INPUT LINE A\$, B\$

The INPUT LINE statement allows a character string (ending with a line terminator) to be input to a specified variable. The line terminator is included in the string with INPUT LINE but discarded with LINPUT.

INPUT LINE # and LINPUT # 8.1.3.1

INPUT LINE[#]expression, string variable(s)
LINPUT[#]expression, string variable(s)

10 INPUT LINE # 4, A\$, B\$

The INPUT LINE # and LINPUT # statements read strings from a terminal-format file.

KILL 8.3.2

KILL string expression

10 KILL "SALARY"

The KILL statement deletes a file from storage.

LET 1.7

{LET variable(s)=expression}
{ variable(s)=expression}

10 A=65

The LET statement assigns constants and expressions to variables. The keyword LET is optional.

LINPUT

See INPUT LINE

LINPUT

See INPUT LINE #

MAP 9.4

MAP (name) [{ALIGNED }
{UNALIGNED}] element(s)

10 MAP (Buff1) A\$, B\$, C

The MAP statement associates a named buffer with a file. Specified data in the element list is moved from the file to the buffer on a GET and from the buffer to the file on a PUT.

MARGIN 8.1.8

MARGIN [#expression,] num exp

10 MARGIN 5

The MARGIN statement allows you to modify the margin of your terminal or terminal-format file.

SUMMARY OF BASIC-PLUS-2 STATEMENTS, FUNCTIONS, AND OPERATORS

MAT INPUT 7.4.1

MAT INPUT array(s)

50 MAT INPUT A

The MAT INPUT statement allows element values to be entered in an array. Input is read from the terminal. Elements are stored in row order as they are typed.

MAT PRINT 7.4.2

MAT PRINT array(s)

120 MAT PRINT A;

The MAT PRINT statement outputs each element of a specified array.

MAT READ 7.4.3

MAT READ array(s)

50 MAT READ B,C

The MAT READ statement reads the values into elements of a 1- or 2-dimensional array from a DATA statement.

MOVE 9.7

MOVE [{ALIGNED }] {FROM} file exp, I/O list
 {UNALIGNED} {TO }

15 MOVE TO #5, A\$, B, C(), FILL%

The MOVE statement associates the data in a record with the variables you specify in the I/O list.

NAMEAS 8.3.1

NAME string1 AS STRING2

15 NAME "MONEY" AS "ACCNTS"

The NAME-AS statement renames a file without changing the contents of the file or the file number associated with it.

NEXT 4.4.1
4.4.3

NEXT variable(s)

15 NEXT I

The NEXT statement terminates a FOR, WHILE, or UNTIL loop. The variable must correspond with the variable in the initial FOR statement. Nested loops cannot cross each other.

NODATA 3.1.3.4
8.1.7.3

NODATA [#expression,] line number

25 NODATA # 4, 85

BASIC checks for the end of file (or data in a DATA statement) and goes to the line number specified if no data is left.

SUMMARY OF BASIC-PLUS-2 STATEMENTS, FUNCTIONS, AND OPERATORS

ONERROR

4.8.1

ONERROR GOTO line number

25 ONERROR GOTO 50

The ONERROR statement allows control to shift to an error-handling routine. Substituting the BACK clause for a line number in an ONERROR GOTO statement causes control to shift to an error-handling routine in the calling program.

ON GOSUB

4.7.2

ON num exp GOSUB line number(s)

50 ON A+B GOSUB 80, 95, 100

The ON GOSUB statement is used to conditionally transfer control to one of several subroutines or to one of several entry points into one or more subroutines.

ONGOTO

4.2

ON num exp {GOTO} line number(s)
 {THEN}

20 ON J% GOTO 85

The ON GOTO statement allows you to transfer control to another line of the program.

ONTHEN

See ONGOTO

OPEN

8.1.1 Terminal-Format
8.2.2 Virtual Array
9.5.1 Record

OPEN filename exp [FOR {INPUT }
 {OUTPUT}]

AS [FILE] [#] expression

,[ORGANIZATION] {SEQUENTIAL } [{FIXED }
 {RELATIVE } [{VARIABLE }
 {INDEXED } [{STREAM }]]

[,ACCESS {READ }] [ALLOW {NONE }
 {WRITE } [{READ }
 {MODIFY } [{WRITE }
 {SCRATCH }] [{MODIFY }]]

{,MAP (mapname)
{,RECORDSIZE num exp}

[,INVALID line no.] [,LOCKED line no.]

[{ ,DOUBLEBUF }
[{ ,BUFFER [#] num exp }]

[,BLOCKSIZE num exp]

[{ ,SPAN }
[{ ,NOSPAN }]]

SUMMARY OF BASIC-PLUS-2 STATEMENTS, FUNCTIONS, AND OPERATORS

```
[,BUCKETSIZE num exp ]
[,CLUSTERSIZE num exp ]
[,MODE num exp] [,DENSITY num exp]
[,PRIMARY [KEY]name ]
[,ALTERNATE [KEY]name] [NODUPLICATES]
                        [NOCHANGES]
```

10 OPEN "FILE" FOR INPUT AS FILE 4

The OPEN statement enables you to create a new file or access an existing file. You can use the OPEN statement to access terminal-format files as well as record files.

PAGE 8.1.9

```
PAGE [#expression,] num exp
```

20 PAGE #2, 60

The PAGE statement allows you to set a PAGE size of any positive number of lines.

PRINT 3.2

```
PRINT [expression(s)]
```

30 PRINT A+B

The PRINT statement causes the data you specify to be output on the terminal. The expression list can be expressions, variables, or quoted strings separated by a comma or a semicolon. Commas cause output to terminal print zones; semicolons ignore the print zones.

PRINT # 8.1.4

```
PRINT [#]expression, list
```

65 PRINT # 6, A, B+C

The PRINT # statement writes data into the specified terminal-format file.

PRINT USING 10.1

```
PRINT USING string, list
```

10 PRINT USING "###.##", A,B,C

The PRINT USING statement causes output to be printed in a specified format. The list indicates the elements to be printed or a line reference to an IMAGE statement.

PUT 9.6

```
PUT file exp [,RECORD num exp ]
              [ {,MAP line no. } ]
              [ {,COUNT num exp} ]
              [,LOCKED line no. ]
              [,INVALID line no.]
```

25 PUT #7, RECORD 15

SUMMARY OF BASIC-PLUS-2 STATEMENTS, FUNCTIONS, AND OPERATORS

The PUT statement writes a record from a buffer to a specified file. The RECORD clause is used for relative files; the KEY clause is used for indexed files. Sequential files allow PUT operations only at the end of the file. The MAP and COUNT clauses define the size of the record.

RANDOMIZE 6.1.3

RANDOM
RANDOMIZE

10 RANDOM

The RANDOMIZE statement changes the starting point of the RND function to a new unpredictable location.

READ 3.1.3.1

READ variable(s)

75 READ A,B%,C\$, D(5)

The READ statement directs BASIC to read from a list of values built into a data block by a DATA statement.

REM 1.3.1

REM[comment]

30 REM this is a comment

The REM statement contains user-written comments and has no effect on program execution.

RESTORE # (or RESET) 3.1.3.3

RESTORE[#]expression] 8.1.5
9.5.2

30 RESTORE #3

The RESTORE # statement resets the file pointer to the specified terminal-format file or record file to its beginning from the current position of the file. RESTORE without an expression restores the data in a DATA statement.

RESUME 4.8.2

RESUME [line number]

50 RESUME 35

The RESUME statement is the last statement in an error-handling subroutine. If no line number is specified, control is shifted back to the beginning of the line that contained the error. If a line number is specified, control is shifted to that line.

RETURN 4.7.1

RETURN

60 RETURN

The RETURN statement is the last statement in a subroutine. It shifts control to the statement following the last executed GOSUB statement.

SUMMARY OF BASIC-PLUS-2 STATEMENTS, FUNCTIONS, AND OPERATORS

SCRATCH 8.1.6
9.5.4

SCRATCH #file exp

25 SCRATCH #6

The SCRATCH statement allows you to truncate the file. SCRATCH can be used only if the file was OPENED with ACCESS SCRATCH.

SLEEP 4.5.1

SLEEP num exp

10 SLEEP A*B

The SLEEP statement causes a temporary halt in execution. The length of delay is determined by the value of the expression in seconds.

STOP 4.6

STOP

110 STOP

The STOP statement causes a halt in program execution. Files are not closed and a message indicating the location of the halt is printed.

SUB 5.1

SUB name [(dummy argument(s))]

40 SUB TEST (A,B%)

The SUB statement marks the beginning of a subprogram and defines the type and number of subprogram parameters.

SUBEND 5.1

SUBEND

25 SUBEND

The SUBEND statement marks the end of the subprogram and returns control to the calling program. It must appear at the end of all subprograms.

SUBEXIT 5.1

SUBEXIT

45 SUBEXIT

The SUBEXIT statement returns control to the calling program. It has the same effect as GOTO n, where n is the line number of the appropriate SUBEND statement. SUBEXIT is legal only inside a subprogram.

UNTIL 4.4.5

UNTIL conditional exp

50 UNTIL I=0

The UNTIL statement sets up a loop that must have a corresponding NEXT statement. The loop executes until the expression is true.

SUMMARY OF BASIC-PLUS-2 STATEMENTS, FUNCTIONS, AND OPERATORS

UPDATE

9.6

```
UPDATE file exp [ { ,MAP line no. } ]
                 [ ,COUNT exp ]
                 [ ,INVALID line no. ]
```

The UPDATE statement changes an existing record in the file. The new record size, as defined in the MAP or COUNT clause, must be the same as the record it replaces. On sequential files, an UPDATE must be preceded by a successful GET or FIND.

WAIT

4.5.2

```
WAIT num exp
```

```
40 WAIT 15
```

The WAIT statement specifies the maximum number of seconds allowed for input before an error is generated. A zero or null value disables the WAIT.

WHILE

4.4.5

```
WHILE conditional exp
```

```
75 WHILE A%<10%
```

The WHILE statement also sets up a loop that must have a NEXT statement. The WHILE expression is evaluated before each loop iteration. If the expression is true, BASIC executes the statements in the loop. If the expression is false, BASIC executes the statements following the NEXT statement.

F.2 FUNCTIONS

<u>Function</u>	<u>Usage</u>
ABORT	is a system function that causes an exit from a running program.
ABS(x)	returns absolute value of x.
ASCII(x\$)	returns the decimal ASCII value of the first character of a specified string.
ATN(x)	returns the arctangent of x in radians.
ATN2(x,y)	returns the value of the angle whose tangent is equal to x divided by y.
CHR\$(x%)	returns the character equivalent of the ASCII value x%.
CLK\$	returns an 8-character string representing the time of day (hh:mm:ss).
COS(x)	returns the cosine of x.
COT(x)	returns the cotangent of the specified argument.

SUMMARY OF BASIC-PLUS-2 STATEMENTS, FUNCTIONS, AND OPERATORS

CTRLC	enables a user program to trap CTRL/C.
DAT\$	returns an 8-character string representing the current date (dd-mmm-yy).
DATE\$(0%)	returns the current date in the form mm/dd/yy.
DATE\$(x%)	returns the date according to the formula; $x = \text{day of the year} + (\text{years since 1970} * 1000)$ in the form dd-mmm-yy.
ECHO	is a system function that enables echoing on the specified channel. The ECHO function reverses the affect of the NOECHO function.
EDIT\$(string,n%)	converts a string according to the integer specified in the table.
EXP(x)	returns the value of e^x where $e=2.71828$.
FIX(x)	returns the truncated value of x.
INSTR(z%,x\$,y\$)	returns the position of substring x\$ in main string y\$ starting at position z%.
INT(x)	returns the integral part of x.
LEFT\$(x\$,y%)	returns a substring of x\$ beginning at the leftmost position for a total length of y characters. (Also LEFT(x\$,y%))
LEN(x\$)	returns the number of characters in x\$.
LINO(x)	returns a line number. The LINO function resolves line number references after a RESEQUENCE command is given.
LOG(x)	returns the natural logarithm of x.
LOG10(x)	returns the common logarithm of x.
MAR%	returns the margin width of a terminal-format file.
MID\$(string,n1%,n2%)	returns a substring of string starting at position n1% with n2% characters.
MOD(x,y)	returns the real result of $x \bmod y$, which is equal to $x - y * \text{INT}(x/y)$.
MOD%(x,y)	returns the integer result of $x \bmod y$, which is the remainder of x/y .
NOECHO	system function that inhibits echoing on specified channel. The NOECHO function reverses the effect of the ECHO function.
PI	is a constant value, 3.14159.
POS(x\$,y\$,z%)	returns the position of substring x\$ in main string y\$ beginning at position z%. (See also INSTR.)

SUMMARY OF BASIC-PLUS-2 STATEMENTS, FUNCTIONS, AND OPERATORS

POS% (n)	returns the horizontal print position of the terminal-format file specified by channel n.
PPS% (n)	returns the total page count of the terminal-format file specified by channel n.
RAD\$(x%)	converts the integer x% to its RADIX-50 equivalent.
RCTRLC	is a system function that reverses the effect of the CTRLC function. (CTRLC enables a user program to trap CTRL/C.)
RCTRL0	is a system function that cancels the effect of typing CTRL/O on the specified channel.
RIGHT\$(x\$,y%)	returns a substring of x\$ that ranges from the first character to the yth character.
RND	returns a random number between 0 and 1. The argument is optional.
SEG\$(x\$,y%,z%)	returns a substring of x\$ that ranges from the yth character to the zth character.
SGN(x)	returns 1 if x is positive, 0 if x is zero, and -1 if x is negative.
SIN(x)	returns the sine of x in radians.
SPACE\$(x)	produces a string of x spaces.
SQR(x)	returns the square root of x. (SQRT(x) is also valid.)
STR\$(x)	returns the value of an expression as a string without leading and trailing blanks. (NUM\$(x) is also valid.)
STRING\$(x%,y%)	creates a string of x length whose characters represent the ASCII value of y.
TAB(x%)	moves the print head to the xth position.
TAN(x)	returns the tangent of x in radians.
TIME\$(n%)	returns time as n minutes before midnight.
TIME\$(0%)	returns current time.
TIME(0)	returns clock time in seconds since midnight.
TIME(1%)	returns used CPU time in tenths of seconds.
TIME(2%)	returns connect time in minutes.
USR%	returns a string containing the user's current connected directory and structure.
VAL (n\$)	returns the real value of a real string.

SUMMARY OF BASIC-PLUS-2 STATEMENTS, FUNCTIONS, AND OPERATORS

VAL%(n\$) returns the integer value of an integer string.

VPS%(n) returns the vertical print position of the terminal-format file specified by channel n.

XLATE converts a string from one storage code into another.

F.3 OPERATORS

Table F-1
Arithmetic Operators

Operator	Example	Meaning
^ or **	5^2 or 5**2	exponentiation
*	A*B	multiplication
/	A/B	division
+	A+B	addition
-	A-B	subtraction, unary minus

Table F-2
Logical Operators

Operator	Example	Meaning
NOT	NOT A	logical negative of A
AND	A AND B	logical product of A and B
OR	A OR B	logical sum of A and B
XOR	A XOR B	logical exclusive OR of A and B
EQV	A EQV B	A is logically equivalent to B
IMP	A IMP B	logical implication of A and B

SUMMARY OF BASIC-PLUS-2 STATEMENTS, FUNCTIONS, AND OPERATORS

Table F-3
Relational Operators

Operator	Example	Meaning
=	A=B	A is equal to B
<	A<B	A is less than B
>	A>B	A is greater than B
<= or =<	A<=B	A is less than or equal to B
>= or =>	A>=B	A is greater than or equal to B
# or <> or ><	A<>B	A is not equal to B
==	A==B	A is approximately equal to B
+,&	A\$+B\$	string concatenation

A is approximately equal to B (A==B) if the difference between A and B is less than 10^{-6} .

If A\$ and B\$ are strings, the relation (==) is true if the contents of A\$ and B\$ are the same in length and composition.

INDEX

- ABORT system function, 6-37
- ABS function, 6-8
- Absolute value function, 6-8
- Access,
 - Random, 9-4
 - Sequential record file, 9-3
 - Simultaneous record, 9-23
- ACCESS clause, 8-2, 8-14, 9-14
- Accessing a record file, 9-12
- Account descriptor, TOPS-20, B-2
- Actual arguments, 5-3
- Adding matrices, 7-4
- Addition operator, 1-13
- Aine notation, 1-7
- Algebraic functions, 6-2, 6-4
- ALIGNED clause, 9-8, 9-24
- ALLOW clause, 8-3, 8-14, 9-14
- ALTERNATE key, 9-22
- ALTERNATE KEY clause, 9-15
- Ampersand, 1-5
- AND operator, 1-18
- Arctangent, 6-2
- Arctangent (two-argument), 6-2
- Arguments in subprogram, Dummy, 5-3
- Arguments in suprogram, Actual, 5-3
- Arithmetic expressions, 1-13
- Arithmetic operators, 1-13, 1-14, F-16
- Array,
 - Dimensioning an, 7-1, 7-2
 - Initializing an, 7-1
 - Inputting, 7-7
- Array I/O, 7-6
- Array printing, 7-8
- Arrays, 1-23
 - Using, 7-1
- ASCII code conversion, 6-22
- ASCII function, 6-22
- ASCII table, E-1
- Assigning values to variables, 1-21
- Assignment matrix operation, 7-4
- Assignment statement, 1-22
- Asterisks in PRINT USING, 10-12
- ATN function, 6-2
- ATN2 function, 6-2
- Attribute specification, File, B-6
- BACK clause, 4-22
- Backslash in line, 1-4
- Backslashes in PRINT USING, 10-13
- BASIC,
 - Elements of, 1-1
 - Entering, 2-2
 - Leaving, 2-3
 - Logging off from, 2-4
 - Running, 2-1
 - Using, 2-1
- BASIC command, 2-1
- BASIC command format, 2-2
- BASIC command level, 2-1, 2-2
- BASIC commands, Immediate mode and, 2-37
- BASIC line, Keyword in, 1-2 Line number in, 1-2
- BASIC program structure, 1-1
- BASIC reserved words, D-1
- BASIC statement summary, F-1
- BASIC work area, 2-1
- Batch, C-1
- Batch control file, C-1
- Batch job, Submitting, C-2
- Batch log file, C-2
- Batch switches, C-2
- Blank COMMON, 5-9
- BLOCKSIZE clause, 9-15
- Branching, Multiple, 4-3
- BUCKETSIZE clause, 9-15
- Buffer, MAP, 9-8
- BUFFER clause, 9-15
- Buffer mapping, I/O, 9-24
- BUILD command, 2-12
- BY clause, 4-11

INDEX

- C (uppercase) in PRINT USING, 10-13
- CALL,
 - COMMON across, 5-7
- Call by reference, 5-3, 5-4
- Call by value, 5-3, 5-4
- CALL statement, F-1
- Carets in PRINT USING, 10-12
- CATALOG command, 2-15
- CHAIN,
 - COMMON across, 5-9
- CHAIN statement, 5-5, F-1
- CHANGE statement, 6-24, F-1
- CHANGES clause, 9-16
- Changing line numbers, 2-10
- Character conversions, 6-22
- Character set, 1-2
- Character string conversion, 6-24
- Character translation, 1-2
- CHECK switch, 2-20, 2-21
- Checking,
 - Syntax, 2-20
- Checking directory, 2-15
- Checking input, 2-5
- CHR\$ function, 6-22
- Circumflexes in PRINT USING, 10-12
- Clause,
 - ACCESS, 8-2, 8-14, 9-14
 - ALIGNED, 9-8, 9-24
 - ALLOW, 8-3, 8-14, 9-14
 - ALTERNATE KEY, 9-15
 - BACK, 4-22
 - BLOCKSIZE, 9-15
 - BUCKETSIZE, 9-15
 - BUFFER, 9-15
 - BY, 4-11
 - CHANGES, 9-16
 - CLUSTERSIZE, 9-15
 - COUNT, 9-20
 - DENSITY, 9-15
 - DOUBLEBUF, 9-15
 - DUPLICATES, 9-16
 - ELSE, 4-4
 - FIXED, 9-13
 - GO BACK, 4-20, F-9
 - GOTO, 4-3, 4-4, 4-20, F-9
 - INPUT, 8-2
 - INVALID, 8-3, 8-14, 9-15
 - LOCKED, 8-3, 8-14, 9-15
 - MAP, 9-14
 - MODE, 9-15
 - NOCHANGES, 9-16
 - NODUPLICATES, 9-16
 - NOSPAN, 9-15
 - ORGANIZATION, 9-13
- Clause (Cont.)
 - OUTPUT, 8-2
 - PRIMARY KEY, 9-15
 - RECORDSIZE, 9-7, 9-15, 9-20
 - SPAN, 9-15
 - STEP, 4-11
 - STREAM, 9-13
 - THEN, 4-3, 4-4
 - UNALIGNED, 9-8, 9-24
 - UNTIL, 4-11
 - VARIABLE, 9-13
 - WHILE, 4-11
- Clearing memory, 2-9
- Clearing the work area, 2-30
- CLK\$ function, 6-27
- Clock function, 6-27
- CLOSE # statement, 8-4
- CLOSE statement, 9-12, 9-18, F-2
- Closing record file, 9-18
- Closing terminal-format file, 8-4
- Closing virtual-array file, 8-13, 9-18
- CLUSTERSIZE clause, 9-15
- Codes,
 - File protection, B-7
- Combining programs, 2-16
- Comma formatting, 3-9
- Command,
 - BASIC, 2-1
 - BUILD, 2-12
 - CATALOG, 2-15
 - CONTINUE, 2-28
 - DEBUG, 2-34
 - DELETE, 2-8
 - DO, 2-17, 2-18
 - GOODBYE, 2-4
 - HELP, 2-3
 - LIST, 2-5, 2-24
 - MLIST, 2-21, 2-22
 - MODE, 2-21
 - MONITOR, 2-3
 - NEW, 2-4
 - OLD, 2-14, 2-24
 - OLDLSA, 2-13
 - QUIET, 2-20
 - RENAME, 2-15
 - RESEQUENCE, 2-10
 - RUN, 2-9, 2-24
 - SAVE, 2-11
 - SCRATCH, 2-9, 2-30
 - SCRATCH ALL, 2-31
 - SCRATCH IMMEDIATE, 2-31
 - SCRATCH RESET, 2-30
 - START, 2-33

INDEX

- Command (Cont.)
 - STATUS, 2-19, 2-20
 - SYSTEM, 2-3
 - UNSAVE, 2-15
 - VERBOSE, 2-21
 - WEAVE, 2-16, 2-17
- Command error messages, A-1
- Command file, 2-17, 2-18, 2-20
- Command format,
 - BASIC, 2-2
- Command level,
 - BASIC, 2-1, 2-2
- COMMAND switch, 2-20, 2-21
- Commands,
 - Immediate-mode, 2-25
- Commands in control file,
 - C-2
- Comment field, 1-6
- Comment field in line, 1-5
- Comment in multi-statement line, 1-6
- Comments in program, 1-5
- COMMON,
 - Blank, 5-9
- COMMON across CALL, 5-7
- COMMON across CHAIN, 5-9
- COMMON statement, 5-6, F-2
- Compilation error messages,
 - A-2, A-11
- Compiler mode, 2-21
- Compiler switches,
 - Disabling, 2-19, 2-20
 - Enabling, 2-19, 2-20, 2-21
- Compiler warning messages,
 - 2-20
- CON MAT value, 7-2
- Conditional execution of statement, 4-25
- Constants, 1-7
 - Integer, 1-8
 - Range of integer, 1-8
 - Real, 1-7
 - String, 1-9
- Constants characters in,
 - String, 1-9
- Constants quotes in,
 - String, 1-9
- Contexts,
 - Immediate-mode, 2-26
- Continuation character in line, 1-5
- Continuation line, 1-4
- CONTINUE command, 2-28
- Control,
 - Program, 4-4
- Control (unconditional),
 - Program, 4-1
- Control between program,
 - Transfer, 5-5
- Control conditional transfer,
 - Program, 4-4
- Control error handling,
 - Program, 4-20
- Control file,
 - Batch, C-1
 - Commands in, C-2
- Control multiple branching,
 - Program, 4-3
- Control subroutine transfer,
 - Program, 4-19
- Conversion,
 - ASCII code, 6-22
- Conversion functions, 6-22
- Correcting program mistakes,
 - 2-6, 2-7
- COS function, 6-2
- Cosine, 6-2
- COT function, 6-2
- Cotangent, 6-2
- COUNT clause, 9-20
- CPU time function, 6-30
- Creating a program, 2-4
- Creating a record file,
 - 9-12
- Creating LSA file, 2-13
- Creating terminal-format file, 8-2
- Creating virtual-array file,
 - 8-13
- Creation ^of loops, 4-7
- CTRL/C in logging in, B-1
- CTRL/C interrupt execution,
 - 2-9
- CTRL/O supresses printout,
 - 2-9
- CTRL/U,
 - Deleting with, 2-7
 - Erasing with, 2-7
- CTRLC system function, 6-36

- DAT\$ function, 6-28
- Data inputting, 3-1
- Data printing, 3-1
- DATA statement, 3-6, 7-9, F-2
- Date functions, 6-27
- DATE\$ function, 6-28
- DEBUG command, 2-34

INDEX

- Decimal point in PRINT
 - USING, 10-11
- DEF * statement, F-3
- DEF * statetment, 6-43
- DEF * switch, 2-21
- DEF statement, 6-37, 6-41, F-2
- Default,
 - NEW command, 2-4
- DEFS,
 - Multi-line, 6-42
- DEFS,
 - Single-line, 6-39
- Delaying program execution, 4-14
- DELETE command, 2-8
- DELETE key,
 - Using the, 2-7
- DELETE statement, 9-4, 9-5, 9-19, 9-20, 9-21, F-3
- Deleting by line number, 2-8
- Deleting file, 8-15
- Deleting partial line, 2-7
- Deleting record file, 9-19
- Deleting stored program, 2-15
- Deleting with CTRL/U, 2-7
- DENSITY clause, 9-15
- DET matrix function, 7-6
- Determinant of matrix, 7-6
- Device specification, B-5
- Diagnostic messages, A-1
- DIM # statement, 8-12, F-3
- DIM statement, 1-23, 1-24, F-3
- DIMENSION statement, F-3
- Dimensioning an array, 7-1, 7-2
- Directory,
 - Checking, 2-15
- Directory function, 6-31
- Directory specification, B-5
- Disable echo function, 6-35
- Disabling compiler switches, 2-19, 2-20
- Division operator, 1-13
- DO command, 2-17, 2-18
- Documentation in program, 1-6
- Documentation in program line, 1-5
- Dollar signs in PRINT USING, 10-12
- DOUBLEBUF clause, 9-15
- Dummy arguments, 5-3
- DUPLICATES clause, 9-16
- Dynamic mapping, 9-24
- E (uppercase) in PRINT
 - USING, 10-13
- Echo function,
 - Disable, 6-35
 - Enable, 6-35
- ECHO system function, 6-35
- EDIT\$ function, 6-20
- Element,
 - FILL, 9-8
- Elements in list, 1-12
- Elements in table, 1-12
- Elements of BASIC, 1-1
- ELSE clause, 4-4
- Enable echo function, 6-35
- Enabling compiler switches, 2-19, 2-20, 2-21
- END statement, 4-15, F-3
- Entering BASIC, 2-2
- EQV operator, 1-18
- Erasing with CTRL/U, 2-7
- ERL error variable, 4-23
- ERN\$ error variable, 4-23
- ERR error variable, 4-23
- Error handling, 4-20
- Error messages, A-12, A-17
- Error messages,
 - Command, A-1
 - Compilation, A-2, A-11
- Error variable ERL, 4-23
- Error variable ERN\$, 4-23
- Error variable ERR, 4-23
- Error variables, 4-23
- Evaluating expressions, 1-20
- Example,
 - Indexed record file, 9-28.
 - Relative record file, 9-27
 - Sequential record file, 9-26, 9-27
- Excess data in INPUT, 3-2
- Exclamation point in line, 1-6
- Exclamation point in PRINT
 - USING, 10-13
- Executing a program, 2-9
- Execution,
 - Delaying program, 4-14
 - Interrupting program, 2-9
 - Stopping program, 4-15
- Execution (CTRL/C),
 - Interrupt, 2-9
- Execution messages, A-12
- Execution of loops, 4-7

INDEX

- Execution of statement,
 - Conditional, 4-25
- Existing file,
 - Working with, 2-12
- Exit program function, 6-37
- EXP function, 6-5
- Exponential function, 6-5
- Exponentiation operator,
 - 1-13
- Expressions, 1-13
 - Arithmetic, 1-13
 - Evaluating, 1-20
 - Integer, 1-14
 - Logical, 1-17
 - Relational, 1-15
 - String, 1-14
- Extended-field strings,
 - 10-10

- Fatal messages, A-3
- Field,
 - Comment, 1-6
 - Key, 9-4
- File,
 - Accessing a record, 9-12
 - Batch control, C-1
 - Batch log, C-2
 - Closing record, 9-18
 - Closing terminal-format,
 - 8-4
 - Closing virtual-array,
 - 8-13, 9-18
 - Command, 2-17, 2-18, 2-20
 - Commands in control, C-2
 - Creating a record, 9-12
 - Creating LSA, 2-13
 - Creating terminal-format,
 - 8-2
 - Creating virtual-array,
 - 8-13
 - Deleting, 8-15
 - Deleting record, 9-19
 - Indexed record, 9-2
 - Line-sequenced ASCII,
 - 2-12
 - Opening indexed, 9-17
 - Opening record, 9-16,
 - 9-17
 - Opening relative, 9-17
 - Opening sequential, 9-16
 - Opening terminal-format,
 - 8-2
 - Opening virtual-array,
 - 8-13
 - Recalling LSA, 2-13
 - Relative record, 9-2
- File (Cont.)
 - Renaming, 8-15
 - Restoring record, 9-18
 - Restoring terminal-format,
 - 8-8
 - Saving program in, 2-11,
 - 2-12
 - Sequential record, 9-2
 - Terminal-format, 8-1,
 - 8-13
 - Truncating record, 9-18
 - Truncating
 - terminal-format, 8-9
 - Virtual-array, 8-1
 - Working with existing,
 - 2-12
- File access,
 - Sequential record, 9-3
- File attribute
 - specification, B-6
- File example,
 - Indexed record, 9-28
 - Relative record, 9-27
 - Sequential record, 9-26,
 - 9-27
- File functions,
 - Terminal-format, 6-31
- File generation
 - specification, B-6
- File I/O,
 - Terminal-format, 8-4
- File name specification,
 - B-6
- File OPEN syntax,
 - Indexed, 9-17
 - Relative, 9-17
 - Sequential, 9-16
- File operations,
 - Record, 9-12, 9-19
- File organization,
 - Record, 9-1
- File pointer,
 - Moving, 9-18
- File protection
 - specification, B-7
- File specifications, B-4
- File type specification,
 - B-6
- Files,
 - Record, 9-1
 - Virtual-array, 8-12
- FILL element, 9-8
- FIND statement, 9-4, 9-5,
 - 9-19, 9-20, 9-21, F-4
- FIX function, 6-9
- Fixed, 9-6
- FIXED clause, 9-13
- FNEND statement, 6-42, F-4

INDEX

- FNEXIT statement, 6-42, F-4
- FOR (conditional) statement, 4-11, F-4
- FOR modifier, 4-29
- FOR statement, 4-7, F-4
- FOR statement with test, 4-12
- Format,
 - Fixed-length record, 9-7
 - Line, 1-2
 - Stream-format record, 9-7
 - Variable-length record, 9-7
- Format of line, 1-2
- Formats,
 - Record, 9-6
- Formatted output, 10-1
- Formatting,
 - Comma, 3-9
 - Numbers, 3-12, 10-2
 - Semicolon, 3-9
 - String, 3-12, 10-8
 - Strings PRINT USING, 10-8
- Function,
 - ABORT system, 6-37
 - ABS, 6-8
 - Absolute value, 6-8
 - ASCII, 6-22
 - ATN, 6-2
 - ATN2, 6-2
 - CHR\$, 6-22
 - CLK\$, 6-27
 - Clock, 6-27
 - COS, 6-2
 - COT, 6-2
 - CTRLC system, 6-36
 - DAT\$, 6-28
 - DATE\$, 6-28
 - DET matrix, 7-6
 - Directory, 6-31
 - Disable echo, 6-35
 - ECHO system, 6-35
 - EDIT\$, 6-20
 - Enable echo, 6-35
 - Exit program, 6-37
 - EXP, 6-5
 - Exponential, 6-5
 - FIX, 6-9
 - Horizontal print, 6-32
 - INSTR, 6-14
 - INT, 6-7
 - Integer, 6-7
 - INV matrix, 7-6
 - LEFT\$, 6-18
 - LEN, 6-13
 - LINO, 4-24
 - LOG, 6-5
 - LOG10, 6-5
- Function (Cont.)
 - MAR\$, 6-31
 - Margin, 6-31
 - MID\$, 6-17
 - MOD, 6-12
 - Multi-line, 6-41, F-2
 - NOECHO system, 6-35
 - NUM\$, 6-26
 - Numeric, 6-1
 - Page count, 6-34
 - PI, 6-2
 - POS, 6-14
 - POS%, 6-32
 - PPS%, 6-34
 - Print position, 6-32, 6-33
 - RAD, 6-23
 - Random number, 6-10
 - RCTRLC system, 6-36
 - RCTRLLO system, 6-34
 - Resume output, 6-34
 - RIGHT\$, 6-18
 - RND, 6-10
 - SEG\$, 6-15
 - SGN, 6-9
 - Sign, 6-9
 - SIN, 6-2
 - Single-line, 6-38, 6-39, 6-40, F-2
 - SPACE\$, 6-19
 - SQR, 6-5
 - Square root, 6-5
 - STR\$, 6-26
 - String extract, 6-15
 - String length, 6-13
 - String trimming, 6-14
 - STRING\$, 6-19
 - TAB, 3-12
 - TAN, 6-2
 - TIME, 6-29, 6-30
 - TIME\$, 6-29, 6-30
 - TRM\$, 6-14
 - TRN matrix, 7-5
 - USR\$, 6-31
 - VAL, 6-26
 - VAL%, 6-26
 - Vertical print, 6-33
 - VPS%, 6-33
 - XLATE, 6-24
- Functions, 1-20, 6-1
- Functions,
 - Algebraic, 6-4
 - Conversion, 6-22
 - Date, 6-27
 - Matrix, 7-5
 - Numeric, 6-1
 - String, 6-13
 - String search, 6-14

INDEX

- Functions (Cont.)
 - Summary of, F-13
 - System, 6-34
 - Terminal-format file, 6-31
 - Time, 6-27
 - User-defined, 6-37

- Generation specification,
 - File, B-6
- GET statement, 9-4, 9-5, 9-8, 9-19, 9-20, 9-21, F-5
- GO BACK clause, 4-20, F-9
- GOODBYE command, 2-4
- GOSUB statement, 4-18, F-5
- GOTO clause, 4-3, 4-4, 4-20, F-9
- GOTO statement, 4-1, F-5

- Handling,
 - Error, 4-20
- HEADER switch, 2-20, 2-21
- Headers,
 - Program, 2-20
- HELP command, 2-3
- Hierarchy of operators, 1-21
- Horizontal print function, 6-32

- I/O,
 - Array, 7-6
 - Terminal, 3-1
 - Terminal-format file, 8-4
- I/O buffer mapping, 9-24
- IDN MAT value, 7-2
- IF modifier, 4-26
- IF statement, F-5
- IF-THEN-ELSE statement, 4-4
- IFEND # statement, 8-9, F-6
- IFMORE # statement, 8-10, F-6
- Initializing an array, 7-1
- IMAGE statement, F-6, 10-13
- Immediate mode, 2-25, 2-26, 2-27
- Immediate mode and BASIC commands, 2-37
- Immediate-mode statements,
 - Nesting, 2-35
- IMP operator, 1-18

- Implied loop, 4-25, 4-29
- In,
 - Logging, B-2
- Including data in program, 3-5
- Indexed file,
 - Opening, 9-17
- Indexed file OPEN syntax, 9-17
- Indexed file random access, 9-5
- Indexed file sequential access, 9-4
- Indexed record file, 9-2
- Indexed record file example, 9-28
- Indexed record operations, 9-21
- Inner loop, 4-10
- INPUT,
 - Excess data in, 3-2
 - Insufficient data in, 3-2
- INPUT # statement, 8-4, F-6
- INPUT clause, 8-2
- INPUT LINE # statement, 8-6, F-7
- INPUT LINE statement, 3-4, F-7
- INPUT statement, 3-1, F-6
- Inputting,
 - Data, 3-1
 - Variables, 3-1
- Inputting array, 7-7
- Inputting strings, 3-4
- INSTR function, 6-14
- Insufficient data in INPUT, 3-2
- INT function, 6-7
- Integer constants, 1-8
 - Range of, 1-8
- Integer expressions, 1-14
- Integer function, 6-7
- Integer subscripted variable, 1-12
- Integer variables, 1-11
- Interrupt execution (CTRL/C), 2-9
- Interrupting program execution, 2-9
- INV matrix function, 7-6
- INVALID clause, 8-3, 8-14, 9-15
- Inverting matrices, 7-6

- Job,
 - Submitting batch, C-2

INDEX

- Job (Cont.)
 - TOPS-20, B-2

- Key,
 - ALTERNATE, 9-22
 - PRIMARY, 9-22
- Key field, 9-4
- Key match,
 - Approximate, 9-22
 - Exact, 9-22
 - Generic, 9-22
- Key-of-reference, 9-5
- Keyword in BASIC line, 1-2
- KILL statement, 8-16, F-7

- L (uppercase) in PRINT
 - USING, 10-13
- Leaving BASIC, 2-3
- LEFT\$ function, 6-18
- LEN function, 6-13
- LET statement, 1-21, 1-22, F-7
- Line,
 - Backslash in, 1-4
 - Comment field in, 1-5
 - Comment in
 - multi-statement, 1-6
 - Continuation, 1-4
 - Continuation character in, 1-5
 - Deleting partial, 2-7
 - Documentation in program, 1-5
 - Exclamation point in, 1-6
 - Format of, 1-2
 - Leading spaces in, 1-3
 - Longest, 1-3
 - Multi-statement, 1-4
 - Single-statement, 1-4
 - Statements in a, 1-3
 - Terminator in, 1-2
 - Trailing spaces in, 1-3
- Line format, 1-2
- Line number,
 - Deleting by, 2-8
- Line number function, 4-24
- Line number in BASIC line, 1-2
- Line numbers,
 - Changing, 2-10
 - Range of, 1-2
- Line terminator, 1-2
- Line-sequenced ASCII file, 2-12

- Lines,
 - Listing program, 2-5
- LINO function, 4-24
- LINPUT # statement, 8-6, F-7
- LINPUT statement, 3-4, F-7
- LIST command, 2-5, 2-24
- List elements, 1-12
- Listing compiled program, 2-21
- Listing program lines, 2-5
- Literal constants, 1-9
- LOCKED clause, 8-3, 8-14, 9-15
- Locking,
 - Record, 9-23
- Log file,
 - Batch, C-2
- LOG function, 6-5
- LOG10 function, 6-5
- Logging in, B-2
- Logging off from BASIC, 2-4
- Logging off TOPS-20 command, B-4
- Logical expressions, 1-17
- Logical operator truth tables, 1-19
- Logical operators, 1-18, F-16
- Loop,
 - Implied, 4-25, 4-29
 - Inner, 4-10
 - Outer, 4-10
- Loops,
 - Creation of, 4-7
 - Execution of, 4-7
 - Nesting of, 4-10
- LSA file, 2-12
 - Creating, 2-13
 - Recalling, 2-13

- MAP and OPEN, 9-11
- MAP and RECORDSIZE, 9-10
- MAP buffer, 9-8
- MAP clause, 9-14
- MAP name, 9-8
- MAP space holder in, 9-8
- MAP statement, 9-8, F-7
- MAP statement rules, 9-11
- Mapping,
 - Dynamic, 9-24
 - I/O buffer, 9-24
 - Record, 9-8
- MAR\$ function, 6-31
- MARGIN # statement, 8-10
- Margin function, 6-31

INDEX

- MAT INPUT statement, 7-7, F-8
- MAT PRINT statement, 7-6, 7-8, F-8
- MAT READ statement, 7-6, 7-9, F-8
- MAT statement, 7-1
- MAT STATEMENT values, 7-2
- Match,
 - Approximate, 9-22
 - Exact, 9-22
 - Generic, 9-22
- Matrices,
 - Adding, 7-4
 - Inverting, 7-6
 - Multiplying, 7-4
 - Subtracting, 7-4
 - Transposing, 7-5
- Matrix,
 - Determinant of, 7-6
- Matrix function,
 - DET, 7-6
 - INV, 7-6
 - TRN, 7-5
- Matrix functions, 7-5
- Matrix operation,
 - Assignment, 7-4
- Matrix operations, 7-4
- Memory,
 - Clearing, 2-9
- Messages,
 - Command error, A-1
 - Compilation error, A-2, A-11
 - Compiler warning, 2-20
 - Diagnostic, A-1
 - Execution, A-12
 - Fatal, A-3
 - Nontrappable, 4-20, A-17
 - Trappable, 4-20, A-12
 - Warning, A-11
- Methods,
 - Access, 9-3
- MID\$ function, 6-17
- Minus sign in PRINT USING, 10-12
- Mistakes,
 - Correcting program, 2-6, 2-7
- MLIST command, 2-21, 2-22
- MOD function, 6-12
- Mode,
 - Compiler, 2-21
 - Immediate, 2-25, 2-26, 2-27
- MODE clause, 9-15
- MODE command, 2-21
- Modifier,
 - FOR, 4-29
 - IF, 4-26
 - UNLESS, 4-27
 - UNTIL, 4-29
 - WHILE, 4-28
- Modifiers,
 - Statement, 4-25
- MONITOR command, 2-3
- MOVE FROM, 9-24, 9-25
- MOVE statement, 9-24, F-8
- MOVE TO, 9-24, 9-25
- Moving file pointer, 9-18
- Multi-line DEFs, 6-42
- Multi-line function, 6-41, F-2
- Multi-line statement, 6-43
- Multi-statement line, 1-4
 - Comment in, 1-6
- Multiple branching, 4-3
- Multiplication operator, 1-13
- Multiplying matrices, 7-4
- Name,
 - MAP, 9-8
 - TOPS-20 user, B-2
- NAME-AS statement, 8-15, F-8
- Nesting immediate-mode statements, 2-35
- Nesting of loops, 4-10
- NEW command, 2-4
- NEXT statement, 4-7, F-8
- NOCHANGES clause, 9-16
- NODATA # statement, 8-10
- NODATA statement, 3-8, F-8
- NODEF * switch, 2-21
- NODUPLICATES clause, 9-16
- NOECHO system function, 6-35
- Nontrappable messages, 4-20, A-17
- NOSPAN clause, 9-15
- NOT operator, 1-18
- NUL\$ MAT value, 7-2
- NUM\$ function, 6-26
- Number sign in PRINT USING, 10-11
- Number to string conversion, 6-26
- Number formatting, 3-12, 10-2
- Numeric function, 6-1
- Numeric functions, 6-1

INDEX

- OLD command, 2-14, 2-24
- OLDLSA command, 2-13
- ON GOSUB statement, 4-19, F-9
- ON GOTO statement, 4-3
- ONERROR statement, 4-20, F-9
- ONGOTO statement, F-9
- ONTHEN statement, F-9
- OPEN,
 - MAP and, 9-11
- OPEN statement, 9-12, F-9
- OPEN syntax,
 - Indexed file, 9-17
 - Relative file, 9-17
 - Sequential file, 9-16
 - Terminal-format, 8-2
 - Virtual-array, 8-13
- Opening indexed file, 9-17
- Opening record file, 9-16, 9-17
- Opening relative file, 9-17
- Opening sequential file, 9-16
- Opening terminal-format file, 8-2
- Opening virtual-array file, 8-13
- Operating system,
 - Using TOPS-20, B-1
- Operations,
 - Indexed record, 9-21
 - Matrix, 7-4
 - Record file, 9-12, 9-19
 - Relative record, 9-20
 - Sequential record, 9-19
- Operator,
 - Addition, 1-13
 - AND, 1-18
 - Division, 1-13
 - EQV, 1-18
 - Exponentiation, 1-13
 - IMP, 1-18
 - Multiplication, 1-13
 - NOT, 1-18
 - OR, 1-18
 - Subtraction, 1-13
 - XOR, 1-18
- Operator precedence, 1-21
- Operators,
 - Arithmetic, 1-13, 1-14, F-16
 - Hierarchy of, 1-21
 - Logical, 1-18, F-16
 - Relational, 1-15, F-17
 - String relational, 1-16
 - Summary of, F-16
 - Unary, 1-13
- OR operator, 1-18
- Organization,
 - Record file, 9-1
- ORGANIZATION clause, 9-13
- Outer loop, 4-10
- Output,
 - Formatted, 10-1
- OUTPUT clause, 8-2
- Output function,
 - Resume, 6-34
- Output printing, 3-8

- PAGE # statement, 8-11
- Page count function, 6-34
- PAGE statement, F-10
- Password,
 - TOPS-20, B-2
- PI function, 6-2
- POS function, 6-14
- POS% function, 6-32
- PPS% function, 6-34
- Precedence operator, 1-21
- PRIMARY key, 9-22
- PRIMARY KEY clause, 9-15
- PRINT # statement, 8-7, F-10
- Print position function, 6-32, 6-33
- PRINT statement, 3-8, F-10
- PRINT USING,
 - Asterisk-fill, 10-6
 - Asterisks in, 10-12
 - Backslashes in, 10-13
 - C (uppercase) in, 10-13
 - Carets in, 10-12
 - Circumflexes in, 10-12
 - Commas in numbers, 10-7, 10-11
 - Decimal point, 10-4
 - Decimal point in, 10-11
 - Digits and, 10-3
 - Dollar sign, 10-7
 - Dollar signs in, 10-12
 - E (uppercase) in, 10-13
 - E format, 10-8
 - Error conditions, 10-14
 - Exclamation point in, 10-13
 - Format characters, 10-11
 - Formatting strings, 10-8
 - L (uppercase) in, 10-13
 - Minus sign in, 10-12
 - Number field, 10-4
 - Number sign in, 10-11
 - Quotation mark in, 10-13
 - R (uppercase) in, 10-13

INDEX

- PRINT USING (Cont.)
 - Special symbols, 10-5
 - Trailing minus, 10-5
- PRINT USING statement, F-10, 10-1
- Printing,
 - Array, 7-8
 - Centered strings, 10-10
 - Data, 3-1
 - Left-justified strings, 10-9
 - Output, 3-8
 - Right-justified strings, 10-9
- Printout (CTRL/O),
 - Supress, 2-9
- Program,
 - Comments in, 1-5
 - Creating a, 2-4
 - Deleting stored, 2-15
 - Executing a, 2-9
 - Immediate-mode, 2-27
 - Including data in, 3-5
 - Listing compiled, 2-21
 - Recalling stored, 2-13, 2-14
 - Renaming stored, 2-15
 - Sample, 2-23
 - Stopping immediate-mode, 2-28
- Program control, 4-4
- Program control
 - (unconditional), 4-1
- Program control conditional transfer, 4-4
- Program control error handling, 4-20
- Program control multiple branching, 4-3
- Program control subroutine transfer, 4-19
- Program execution,
 - Delaying, 4-14
 - Interrupting, 2-9
 - Stopping, 4-15
- Program function,
 - Exit, 6-37
- Program headers, 2-20
- Program in file,
 - Saving, 2-11, 2-12
- Program lines,
 - Listing, 2-5
- Program mistakes,
 - Correcting, 2-6, 2-7
- Program segmentation, 5-1
- Program structure,
 - BASIC, 1-1
- Programs,
 - Combining, 2-16
- Prompt,
 - READY, 2-24
- Prompt string,
 - INPUT, 3-1
 - INPUT LINE, 3-4
 - LINPUT, 3-4
- Protection specification,
 - File, B-7
- PUT statement, 9-4, 9-5, 9-8, 9-11, 9-19, 9-20, 9-21, F-10
- QUIET command, 2-20
- Quotation mark in PRINT USING, 10-13
- R (uppercase) in PRINT USING, 10-13
- RAD function, 6-23
- RADIX-50 conversion, 6-23
- Random access, 9-4
 - Indexed file, 9-5
 - Relative file, 9-5
- Random number function, 6-10
- RANDOMIZE statement, 6-10, F-11
- Range of line numbers, 1-2
- RCTRLC system function, 6-36
- RCTRLO system function, 6-34
- READ statement, 3-5, F-11
- READY prompt, 2-24
- Real constants, 1-7
 - Range of, 1-7
- Real subscripted variable, 1-12
- Real variables, 1-10
- Recalling LSA file, 2-13
- Recalling stored program, 2-13, 2-14
- Record access,
 - Simultaneous, 9-23
- Record file,
 - Accessing a, 9-12
 - Closing, 9-18
 - Creating a, 9-12
 - Deleting, 9-19
 - Indexed, 9-2
 - Opening, 9-16, 9-17

INDEX

- Record file (Cont.)
 - Relative, 9-2
 - Restoring, 9-18
 - Sequential, 9-2
 - Truncating, 9-18
- Record file access,
 - Sequential, 9-3
- Record file example,
 - Indexed, 9-28
 - Relative, 9-27
 - Sequential, 9-26, 9-27
- Record file operations,
 - 9-12, 9-19
- Record file organization,
 - 9-1
- Record files, 9-1
- Record format,
 - Fixed-length, 9-7
 - Stream-format, 9-7
 - Variable-length, 9-7
- Record formats, 9-6
- Record locking, 9-23
- Record mapping, 9-8
- Record operations,
 - Indexed, 9-21
 - Relative, 9-20
 - Sequential, 9-19
- RECORDSIZE clause, 9-7,
 - 9-15, 9-20
- Reference,
 - Call by, 5-3, 5-4
- Relational expressions,
 - 1-15
- Relational operators, 1-15,
 - F-17
- Relational operators,
 - String, 1-16
- Relative file,
 - Opening, 9-17
- Relative file OPEN syntax,
 - 9-17
- Relative file random access,
 - 9-5
- Relative file sequential
 - access, 9-4
- Relative record file, 9-2
- Relative record file
 - example, 9-27
- Relative record operations,
 - 9-20
- REM statement, 1-5, F-11
- RENAME command, 2-15
- Renaming file, 8-15
- Renaming stored program,
 - 2-15
- RESEQUENCE command, 2-10
- Reserved words,
 - BASIC, D-1
- RESET statement, 3-7, F-11
- RESTORE # statement, 8-8,
 - F-11
- RESTORE statement, 3-7,
 - 9-12, 9-18, F-11
- Restoring record file, 9-18
- Restoring terminal-format
 - file, 8-8
- Resume output function,
 - 6-34
- RESUME statement, 4-22,
 - F-11
- RETURN statement, 4-18,
 - F-11
- RIGHT\$ function, 6-18
- RMS, 9-5
- RND function, 6-10
- Routines,
 - Error, 4-20
- Rules,
 - MAP statement, 9-11
- RUN command, 2-9, 2-24
- Running BASIC, 2-1
- Sample program, 2-23
- SAVE command, 2-11
- Saving program in file,
 - 2-11, 2-12
- Scientific notation, 1-8
- SCRATCH # statement, 8-9
- SCRATCH ALL command, 2-31
- SCRATCH command, 2-9, 2-30
- SCRATCH IMMEDIATE command,
 - 2-31
- SCRATCH RESET command, 2-30
- SCRATCH statement, 9-12,
 - 9-18, F-12
- SEG\$ function, 6-15
- Segmentation,
 - Program, 5-1
- Semicolon formatting, 3-9
- Sequential access,
 - Relative file, 9-4
- Sequential access,
 - Indexed file, 9-4
 - Sequential file, 9-3
- Sequential file,
 - Opening, 9-16
- Sequential file OPEN syntax,
 - 9-16
- Sequential file sequential
 - access, 9-3
- Sequential record file, 9-2
- Sequential record file
 - access, 9-3

INDEX

Sequential record file
 example, 9-26, 9-27
 Sequential record
 operations, 9-19
 SGN function, 6-9
 Sign function, 6-9
 Simultaneous record access,
 9-23
 SIN function, 6-2
 Sine, 6-2
 Single-line DEFS, 6-39
 Single-line function, 6-38,
 6-39, 6-40, F-2
 Single-statement line, 1-4
 SLEEP statement, 4-14, F-12
 Space holder in MAP, 9-8
 SPACE\$ function, 6-19
 SPAN clause, 9-15
 Specification,
 Device, B-5
 Directory, B-5
 File attribute, B-6
 File generation, B-6
 File name, B-6
 File protection, B-7
 File type, B-6
 Specifications,
 File, B-4
 SQR function, 6-5
 Square root function, 6-5
 START command, 2-33
 Statement,
 Assignment, 1-22
 CALL, 5-2, F-1
 CHAIN, 5-5, F-1
 CHANGE, 6-24, F-1
 CLOSE, 9-12, 9-18, F-2
 CLOSE #, 8-4
 COMMON, 5-6, F-2
 Conditional execution of,
 4-25
 DATA, 3-6, 7-9, F-2
 DEF, 6-37, 6-41, F-2
 DEF *, F-3
 DELETE, 9-4, 9-5, 9-19,
 9-20, 9-21, F-3
 DIM, 1-23, 1-24, F-3
 DIM #, 8-12, F-3
 DIMENSION, F-3
 END, 4-15, F-3
 FIND, 9-4, 9-5, 9-19,
 9-20, 9-21, F-4
 FNEND, 6-42, F-4
 FNEXIT, 6-42, F-4
 FOR, 4-7, F-4
 FOR (conditional), 4-11,
 F-4
 GET, 9-4, 9-5, 9-8, 9-19,
 Statement (Cont.)
 9-20, 9-21, F-5
 GOSUB, 4-18, F-5
 GOTO, 4-1, F-5
 IF, F-5
 IF-THEN-ELSE, 4-4
 IFEND #, 8-9, F-6
 IFMORE #, 8-10, F-6
 IMAGE, F-6, 10-13
 INPUT, 3-1, F-6
 INPUT #, 8-4, F-6
 INPUT LINE, 3-4, F-7
 INPUT LINE #, 8-6, F-7
 KILL, 8-16, F-7
 LET, 1-21, 1-22, F-7
 LINPUT, 3-4, F-7
 LINPUT #, 8-6, F-7
 MAP, 9-8, F-7
 MARGIN #, 8-10
 MAT, 7-1
 MAT INPUT, 7-7, F-8
 MAT PRINT, 7-6, 7-8, F-8
 MAT READ, 7-6, 7-9, F-8
 MOVE, 9-24, F-8
 Multi-line, 6-43
 NAME-AS, 8-15, F-8
 NEXT, 4-7, F-8
 NODATA, 3-8, F-8
 NODATA #, 8-10
 ON GOSUB, 4-19, F-9
 ON GOTO, 4-3
 ONERROR, 4-20, F-9
 ONGOTO, F-9
 ONTHEN, F-9
 OPEN, 9-12, F-9
 PAGE, F-10
 PAGE #, 8-11
 PRINT, 3-8, F-10
 PRINT #, 8-7, F-10
 PRINT USING, F-10, 10-1
 PUT, 9-4, 9-5, 9-8, 9-11,
 9-19, 9-20, 9-21, F-10
 RANDOMIZE, 6-10, F-11
 READ, 3-5, F-11
 REM, 1-5, F-11
 RESET, 3-7, F-11
 RESTORE, 3-7, 9-12, 9-18,
 F-11
 RESTORE #, 8-8, F-11
 RESUME, 4-22, F-11
 RETURN, 4-18, F-11
 SCRATCH, 9-12, 9-18, F-12
 SCRATCH #, 8-9
 SLEEP, 4-14, F-12
 STOP, 2-28, 4-15, F-12
 SUB, 5-2, F-12
 SUBEND, 5-2, F-12
 SUBEXIT, 5-2, F-12

INDEX

- Statement (Cont.)
 - Time-limit, 4-14
 - UNTIL, 4-13, F-12
 - UPDATE, 9-4, 9-5, 9-11, 9-19, 9-20, 9-21, F-13
 - WAIT, 4-14, F-13
 - WHILE, 4-13, F-13
- Statement modifiers, 4-25
- Statement rules,
 - MAP, 9-11
- Statement summary,
 - BASIC, F-1
- Statement with test,
 - FOR, 4-12
- Statements,
 - Immediate-mode, 2-25
- Statements in a line, 1-3
- Statetment,
 - DEF *, 6-43
- STATUS command, 2-19, 2-20
- STEP clause, 4-11
- STOP statement, 2-28, 4-15, F-12
- Stopping immediate-mode program, 2-28
- Stopping program execution, 4-15
- Stops,
 - If system, B-3
- Storage code conversion, 6-24
- Stored program,
 - Deleting, 2-15
 - Recalling, 2-13, 2-14
 - Renaming, 2-15
- STR\$ function, 6-26
- Stream, 9-6
- STREAM clause, 9-13
- String constants, 1-9
- String constants characters in, 1-9
- String constants quotes in, 1-9
- String expressions, 1-14
- String extract function, 6-15
- String formatting, 3-12, 10-8
- String functions, 6-13
- String length function, 6-13
- String relational operators, 1-16
- String search functions, 6-14
- String subscripted variable, 1-12
- String to number conversion, 6-26
- String trimming function, 6-14
- String variables, 1-11
- STRING\$ function, 6-19
- Strings,
 - Extended-field, 10-10
 - Inputting, 3-4
- Structure,
 - BASIC program, 1-1
 - Record, 9-1
- SUB statement, 5-2, F-12
- SUBEND statement, 5-2, F-12
- SUBEXIT statement, 5-2, F-12
- Submitting batch job, C-2
- Subprogram,
 - Dummy arguments in, 5-3
 - Exiting a, 5-2
- Subprograms, 5-1
- Subroutines, 4-17, 4-18
- Subscripted variables, 1-12
- Subtracting matrices, 7-4
- Subtraction operator, 1-13
- Summary,
 - BASIC statement, F-1
- Supress printout (CTRL/O), 2-9
- Suprogram,
 - Actual arguments in, 5-3
- Switch,
 - CHECK, 2-20, 2-21
 - COMMAND, 2-20, 2-21
 - DEF *, 2-21
 - HEADER, 2-20, 2-21
 - NODEF *, 2-21
 - WARN, 2-20, 2-21
- Switches,
 - Batch, C-2
 - Disabling compiler, 2-19, 2-20
 - Enabling compiler, 2-19, 2-20, 2-21
- Syntax,
 - Indexed file OPEN, 9-17
 - Relative file OPEN, 9-17
 - Sequential file OPEN, 9-16
 - Terminal-format OPEN, 8-2
 - Virtual-array OPEN, 8-13
- Syntax checking, 2-5, 2-20
- System,
 - Using TOPS-20 operating, B-1
- SYSTEM command, 2-3

INDEX

- System function,
 - ABORT, 6-37
 - CTRLC, 6-36
 - ECHO, 6-35
 - NOECHO, 6-35
 - RCTRLC, 6-36
 - RCTRL0, 6-34
- System functions, 6-34
- System stops,
 - If, B-3

- TAB function, 3-12
- Table,
 - ASCII, E-1
- Table elements, 1-12
- TAN function, 6-2
- Tangent, 6-2
- Terminal I/O, 3-1
- Terminal-format file I/O,
 - 8-4
- Terminal-format,
 - Testing for end, 8-9
- Terminal-format file, 8-1,
 - 8-13
- Terminal-format file,
 - Closing, 8-4
 - Creating, 8-2
 - Opening, 8-2
 - Restoring, 8-8
 - Truncating, 8-9
- Terminal-format file functions, 6-31
- Terminal-format OPEN syntax,
 - 8-2
- Terminator,
 - Line, 1-2
- Terminator in line, 1-2
- Testing for end
 - terminal-format, 8-9
- THEN clause, 4-3, 4-4
- TIME function, 6-29, 6-30
- Time functions, 6-27
- TIME\$ function, 6-29, 6-30
- Time-limit statement, 4-14
- TOPS-20 account descriptor,
 - B-2
- TOPS-20 command,
 - CONTINUE, 2-4, 2-11
 - Logging off, B-4
 - LOGIN, B-2
 - LOGOUT, B-4
 - REENTER, 2-4
 - START, 2-4
 - SUBMIT, C-2
 - UNDELETE, 2-11
- TOPS-20 job, B-2

- TOPS-20 operating system,
 - Using, B-1
- TOPS-20 password, B-2
- TOPS-20 user name, B-2
- Transfer,
 - Unconditional, 4-1
- Transfer control between program, 5-5
- Translation,
 - Character, 1-2
- Transposing matrices, 7-5
- Trappable messages, 4-20,
 - A-12
- Trigonometric functions,
 - 6-2
- TRM\$ function, 6-14
- TRN matrix function, 7-5
- Truncating record file,
 - 9-18
- Truncating terminal-format file, 8-9
- Truth tables, 1-19
- Type specification,
 - File, B-6

- UNALIGNED clause, 9-8, 9-24
- Unary operators, 1-13
- Unconditional transfer, 4-1
- UNLESS modifier, 4-27
- UNSAVE command, 2-15
- UNTIL clause, 4-11
- UNTIL modifier, 4-29
- UNTIL statement, 4-13, F-12
- UPDATE statement, 9-4, 9-5,
 - 9-11, 9-19, 9-20, 9-21,
 - F-13
- User name,
 - TOPS-20, B-2
- User-defined functions,
 - 6-37
- Using BASIC, 2-1
- Using the DELETE key, 2-7
- USR\$ function, 6-31

- VAL function, 6-26
- VAL% function, 6-26
- Value,
 - Call by, 5-3, 5-4
- Values,
 - MAT STATEMENT, 7-2
- VARIABLE clause, 9-13
- Variable ERL,
 - Error, 4-23

INDEX

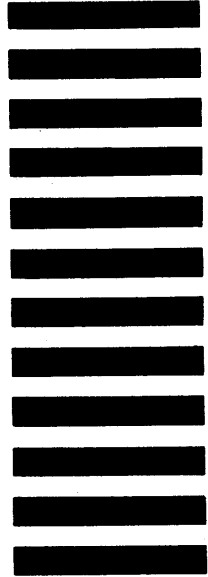
- Variable ERN\$,
 - Error, 4-23
- Variable ERR,
 - Error, 4-23
- Variable record format, 9-6
- Variables, 1-10
 - Assigning values to, 1-21
 - COMMON, 5-7
 - Error, 4-23
 - Immediate-mode, 2-27
 - Integer, 1-11
 - Real, 1-10
 - String, 1-11
 - Subscripted, 1-12
- Variables inputting, 3-1
- VERBOSE command, 2-21
- Vertical print function,
 - 6-33
- Virtual-array file, 8-1
 - Closing, 8-13, 9-18
 - Creating, 8-13
 - Opening, 8-13
- Virtual-array files, 8-12
- Virtual-array OPEN syntax,
 - 8-13
- VPS% function, 6-33
- WAIT statement, 4-14, F-13
- WARN switch, 2-20, 2-21
- Warning messages, A-11,
 - A-12, A-17
- Warning messages,
 - Compiler, 2-20
- WEAVE command, 2-16, 2-17
- WHILE clause, 4-11
- WHILE modifier, 4-28
- WHILE statement, 4-13, F-13
- Words,
 - BASIC reserved, D-1
- Work area,
 - BASIC, 2-1
 - Clearing the, 2-30
- XLATE function, 6-24
- XOR operator, 1-18
- ZER MAT value, 7-2

--- Do Not Tear - Fold Here and Tape ---

digital



No Postage
Necessary
if Mailed in the
United States



BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO. 33 MAYNARD MASS.

POSTAGE WILL BE PAID BY ADDRESSEE

SOFTWARE PUBLICATIONS
200 FOREST STREET MR1-2/E37
MARLBOROUGH, MASSACHUSETTS 01752

--- Do Not Tear - Fold Here and Tape ---

Cut Along Dotted Line