pdp11

BASIC-11
Language Reference Manual

Order No. DEC-11-LIBBB-A-D

digital

BASIC-11
Language Reference Manual

Order No. DEC-11-LIBBB-A-D

digital equipment corporation · maynard, massachusetts

First Printing, September 1976

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 be used or copied only in accordance with the terms of such license.

Digital Equipment Corporation assumes no responsibility for the use or reliability of its software on equipment that is not supplied by DIGITAL.

Copyright © 1976, 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-10
DECCOMM	DECsystem-20	TYPESET-11

CONTENTS

			Page
PREFACE			vii
DOCUMENTATION CONVENTIONS			ix
CHAPTER	1	PROGRAMMING IN BASIC	1-1
	1.1	INTRODUCTION	1-1
	1.2	STRUCTURE OF A BASIC PROGRAM	1-1
	1.3	BASIC CHARACTER SET	1-2
	1.4	LINE FORMAT	1-3
	1.5	STATEMENTS	1-4
	1.5.1	Single and Multi-Statement Lines	1-5
	1.6	DOCUMENTING PROCEDURES (REM STATEMENT)	1-6
	1.7	ENTERING BASIC PROGRAMS	1-7
	1.8	USING BASIC WITHOUT WRITING A PROGRAM (IMMEDIATE MODE)	1-9
CHAPTER	2	ELEMENTS OF BASIC	2-1
	2.1	TERMINOLOGY	2-1
	2.2	CONSTANTS	2-1
	2.2.1	Numeric Constants	2-1
	2.2.2	Integer Constants	2-3
	2.2.3	String Constants	2-3
	2.3	VARIABLES	2-4
	2.3.1	Numeric Variables	2-5
	2.3.2	Integer Variables	2-5
	2.3.3	String Variables	2-6
	2.3.4	Subscripted Variables	2-7
	2.4	FORMING EXPRESSIONS	2-8
	2.4.1	Arithmetic Expressions	2-8
	2.4.2	String Expressions	2-10
	2.4.3	Relational Expressions	2-11
	2.4.4	Functions	2-12
	2.5	ASSIGNING VALUES TO VARIABLES (LET STATEMENT)	2-13
	2.6	ARRAYS	2-14
	2.6.1	Dimensioning Arrays (DIM Statement)	2-16
CHAPTER	3	INPUT AND OUTPUT	3-1
	3.1	SUPPLYING DATA	3-1
	3.1.1	INPUT Statement	3-1
	3.1.2	LINPUT Statement	3-4
	3.1.3	READ, DATA, and RESTORE Statements	3-5
	3.2	CHECKING OUTPUT (PRINT STATEMENT)	3-8
	3.2.1	Printing Zones - The Comma and the Semicolon	3-9
	3.2.2	Output Format for Numbers and Strings	3-12
	3.2.3	Printing with the TAB Function	3-13
CHAPTER	4	CONTROL STATEMENTS	4-1
	4.1	SHIFTING CONTROL OF THE PROGRAM	4-1
	4.1.1	Unconditional Transfer (GO TO Statement)	4-1

			Page
	4.1.2	Multiple Branching (ON GO TO and ON THEN Statements)	4-3
	4.1.3	Conditional Transfer (IF THEN and IF GO TO Statements	4-3
	4.2	EXECUTION OF LOOPS	4-6
	4.2.1	FOR and NEXT Statements	4-7
	4.2.2	Nested Loops	4-11
	4.3	STOPPING PROGRAM EXECUTION (END AND STOP STATEMENTS)	4-12
	4.4	SUBROUTINES	4-13
	4.4.1	GOSUB and RETURN Statements	4-14
	4.4.2	ON GOSUB Statement	4-17
CHAPTER	5	FUNCTIONS	5-1
	5.1	TYPES OF FUNCTIONS AVAILABLE	5-1
	5.2	NUMERIC FUNCTIONS	5-1
	5.2.1	Trigonometric Functions (SIN, COS, ATN, and PI Functions)	5-2
	5.2.2	Algebraic Functions	5-4
	5.2.2.1	Square Root Function (SQR Function)	5-4
	5.2.2.2	Exponential and Logarithm Functions (EXP, LOG, and LOG10 Functions)	5-4
	5.2.2.3	Integer Function (INT Function)	5-6
	5.2.2.4	Absolute Value Function (ABS Function)	5-8
	5.2.2.5	Sign Function (SGN Function)	5-9
	5.2.3	Random Numbers (RND Function and RANDOMIZE Statement)	5-9
	5.3	STRING FUNCTIONS	5-12
	5.3.1	String Manipulation Functions	5-12
	5.3.1.1	Finding the Length of a String (LEN Function)	5-12
	5.3.1.2	Trimming Trailing Blanks Off a String (TRM$ Function)	5-13
	5.3.1.3	Finding the Position of a Substring (POS Function)	5-13
	5.3.1.4	Copying Segments from a String (SEG$ Function)	5-15
	5.3.2	Conversion Functions	5-16
	5.3.2.1	Character and ASCII Code Conversions (ASC and CHR$ Functions)	5-17
	5.3.2.2	Numbers and Their String Representation Conversions (VAL and STR$ Function)	5-18
	5.3.2.3	Binary and Octal to Decimal Conversions (BIN and OCT Functions)	5-20
	5.4	USER-DEFINED FUNCTIONS (DEF STATEMENT AND FN FUNCTION)	5-21
	5.5	DATE AND TIME FUNCTIONS (DAT$ AND CLK$ FUNCTIONS)	5-26
CHAPTER	6	WORKING WITH DATA FILES	6-1
	6.1	INTRODUCTION TO DATA FILES	6-1
	6.2	FILE CONTROL STATEMENTS	6-1
	6.2.1	Opening a File (OPEN Statement)	6-2
	6.2.2	Closing a File (CLOSE Statement)	6-3
	6.3	USING SEQUENTIAL FILES	6-4
	6.3.1	Reading Data from a Sequential File (INPUT # and LINPUT # Statements)	6-4
	6.3.2	Storing Data in a Sequential File (PRINT # Statement)	6-5
	6.3.3	Checking for the End of Input File (IF END # Statement)	6-7
	6.3.4	Restoring a File to the Beginning	6-8

			Page
		(RESTORE # Statement)	
	6.4	USING VIRTUAL ARRAY FILES	6-8
	6.4.1	Dimensioning Virtual Arrays (DIM # Statement)	6-9
	6.5	RENAMING A FILE (NAME STATEMENT)	6-10
	6.6	DELETING A FILE (KILL STATEMENT)	6-11
CHAPTER	7	FORMATTED OUTPUT - THE PRINT USING STATEMENT	7-1
	7.1	INTRODUCTION TO PRINT USING	7-1
	7.2	PRINTING NUMBERS WITH PRINT USING	7-2
	7.2.1	Specifying the Number of Digits	7-3
	7.2.2	Specifying the Location of the Decimal Point	7-3
	7.2.3	Printing a Number That is Larger Than the Field	7-4
	7.2.4	Printing Numbers with Special Symbols	7-5
	7.2.4.1	Printing Numbers with a Trailing Minus Sign	7-5
	7.2.4.2	Printing Numbers with Asterisk Fill	7-6
	7.2.4.3	Printing Numbers with Floating Dollar Signs	7-6
	7.2.4.4	Printing Numbers with Commas	7-7
	7.2.5	Printing Numbers in E Format	7-8
	7.2.6	Fields Which Exceed BASIC's Accuracy	7-8
	7.3	PRINTING STRINGS WITH THE PRINT USING STATEMENT	7-8
	7.3.1	1-Character String Fields	7-9
	7.3.2	Printing Strings in Left-Justified Format	7-9
	7.3.3	Printing Strings in Right-Justified Format	7-9
	7.3.4	Printing Strings in Centered Fields	7-10
	7.3.5	Printing Strings in Extended Fields	7-11
	7.4	SUMMARY OF THE PRINT USING STATEMENT FORMAT	7-11
	7.4.1	Format of Numeric Fields	7-13
	7.4.2	Format of String Fields	7-14
	7.5	PRINT USING STATEMENT ERROR CONDITIONS	7-15
	7.5.1	Fatal Error Conditions	7-16
	7.5.2	Nonfatal Error Conditions	7-16
CHAPTER	8	PROGRAM SEGMENTATION	8-1
	8.1	SEGMENTING PROGRAMS WITH THE CHAIN STATEMENT	8-1
	8.1.1	Preserving Variables Through CHAIN (COMMON Statement)	8-3
	8.2	MERGING PROGRAM SEGMENTS (OVERLAY STATEMENT)	8-6
	8.3	CALLING A ROUTINE WRITTEN IN ANOTHER LANGUAGE (CALL STATEMENT)	8-9
CHAPTER	9	BASIC-11 COMMANDS	9-1
	9.1	KEY COMMANDS	9-1
	9.2	LISTING YOUR PROGRAM (LIST AND LISTNH COMMANDS)	9-2
	9.3	EXECUTING A PROGRAM (RUN AND RUNNH COMMANDS)	9-3
	9.4	DELETING PROGRAM LINES (DEL COMMAND)	9-3
	9.5	ERASING THE PROGRAM (NEW, SCR, AND CLEAR COMMANDS)	9-4
	9.6	PROGRAMS IN FILES	9-4
	9.6.1	Saving the Program in a File (SAVE and REPLACE Commands)	9-5
	9.6.2	Restoring a Program from a File (OLD and APPEND Commands)	9-6
	9.6.3	Running a Program from a File	9-7

			Page
	9.6.4	Deleting a Program File (UNSAVE Command)	9-8
	9.7	CHANGING THE PROGRAM NAME (RENAME COMMAND)	9-8
	9.8	EDITING A LINE (SUB COMMAND)	9-9
	9.9	RESEQUENCING A PROGRAM (RESEQ COMMAND)	9-10
	9.10	SAVING A COMPILED PROGRAM (COMPILE COMMAND)	9-13
	9.11	CHECKING THE LENGTH OF A PROGRAM (LENGTH COMMAND)	9-13
APPENDIX A		ERROR MESSAGES	A-1
APPENDIX B		SUMMARY OF BASIC-11 STATEMENTS, FUNCTIONS, AND COMMANDS	B-1
	B.1	SUMMARY of BASIC-11 STATEMENTS	B-1
	B.2	SUMMARY of BASIC-11 FUNCTIONS	B-5
	B.3	SUMMARY of BASIC-11 COMMANDS	B-7
APPENDIX C		ASCII CHARACTER SET	C-1
INDEX			Index-1

FIGURES

FIGURE	8-1	Segmenting a Program with the CHAIN Statement	8-2
	8-2	CHAIN with COMMON	8-4
	8-3	Segmenting a Program with the OVERLAY Statement	8-7

TABLES

TABLE	2-1	Number Notations	2-2
	2-2	Arithmetic Operators	2-9
	2-3	Arithmetic Operator Precedence	2-10
	2-4	Arithmetic Relational Operators	2-11
	2-5	String Relational Operators	2-12
	7-1	Format Characters for Numeric Fields	7-13
	7-2	Format Characters for String Fields	7-15
	9-1	BASIC-11 Key Commands	9-1
	A-1	Abbreviated Error Messages	A-2
	A-2	BASIC-11 Error Messages	A-3
	A-3	Error Conditions in Functions	A-11
	B-1	Summary of Statements	B-1
	B-2	Summary of Arithmetic Functions	B-5
	B-3	Summary of String Functions	B-6
	B-4	Summary of Commands	B-8
	C-1	ASCII Character Set	C-1

PREFACE

This manual describes the features of the BASIC-11 language. This manual assumes you are familiar with the standard Dartmouth BASIC language and have a knowledge of programming concepts. If you are totally unfamiliar with BASIC, you should read an introduction to BASIC before reading this manual.

This manual describes:

- Structure of BASIC-11 programs
- Elements of BASIC-11
- BASIC-11 statements
- BASIC-11 functions
- BASIC-11 commands
- BASIC-11 error messages

This manual describes the features common to all versions of BASIC-11, but it does not document system-dependent features and procedures. For this information, see your system's BASIC-11 user's guide.

DOCUMENTATION CONVENTIONS

These are the documentation conventions which are used throughout this manual.

The following symbols have special meaning.

Symbol	Meaning
(CTRL/x)	While pressing the CTRL key, type the letter indicated after the slash
(RET)	Type the RETURN key
(RUBOUT)	Type the RUBOUT key

In addition, this manual uses certain conventions when describing the format of statements, functions, and commands.

These are:

Convention	Meaning
[ ]	The enclosed elements are optional. For example: [LET] variable=expression
{ }	A choice of one element among two or more possibilities, for example: IF relational expression {THEN statement / THEN line number / GO TO line number}
...	Preceding element can be repeated as indicated. For example: CLOSE #exprl,#expr2,...
Items in capital letters and special symbols	Type these elements exactly as they appear in the format, for example: LET RUN # Items in capital letters are called keywords.
Items in lower case letters	Replace these elements according to the description provided in text. See below for list of commonly used lower case items.

This list describes some lower case items commonly used in format descriptions. The general meaning of each item is given. Unless a specific format description places restrictions on an item, its general meaning applies.

Lower Case Item	Abbreviation	Meaning
expression	expr	Any valid BASIC-11 expression. It is always a numeric expression (see Section 2.4.1) unless the description specifically states that it can be a numeric or string expression (see Section 2.4.2). For example: (5*SIN(X))^Y
file specification	-	A file specification in the format described in your BASIC-11 user's guide.
integer	int	Any positive integer number constant or any positive numeric constant that could be an integer if a percent sign was put after it. For example: 5%, 3%, 2, 7
line number	-	Any line number as described in Section 1.4. For example: 10, 100, 32767
string	-	Any string expression (see Section 2.4.2). For example: "ABC", C$+SEG$(A$,3,4)
variable	var	A floating point, integer or string variable (see Section 2.3)

If there is more than one lower case word in a format, the words are numbered 1, 2, 3, etc.. For example:

```
CLOSE #expr1,#expr2,#expr3,...
```

Throughout this manual, the term BASIC means BASIC-11 or any BASIC-11 system.

To differentiate between what BASIC prints and what you type, the user type-in is printed in red ink. For example:

```
RUNNH

WHAT NUMBERS? 5,10
THE SUM IS 15

READY
```

All user type-in is terminated by the (RET) key unless the text indicates a different terminator.

CHAPTER 1

PROGRAMMING IN BASIC

1.1 INTRODUCTION

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

BASIC provides an interactive human/machine relationship by allowing you to communicate directly with its processor. It is a conversational programming language which uses simple English-like statements and familiar math notations to perform an operation.

BASIC-11, a BASIC language available on PDP-11 systems, 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 write and execute more complex and efficient programs.

1.2 STRUCTURE OF A BASIC PROGRAM

A BASIC program consists of a set of statements using certain language elements and syntax described in the following chapters. Expressions, line numbers, and statements are joined to solve a particular problem, with each line containing instructions to BASIC.

A BASIC program can be one line or several lines long. This is a complete program:

```
10 PRINT "THIS IS A 1-LINE PROGRAM."
```

Each line begins with a number that identifies the line as a statement and indicates the order of statement execution. Each statement starts with a word specifying the type of operation to be performed.

Examine the following program. What is the result of adding the value of the variable B to the value of the variable C?

```
10 FOR I=1 TO 5
20 INPUT B,C
30 LET A=B+C
40 PRINT "B+C= ";A
50 NEXT I
60 END
```

Look at the structure of this program. There are line numbers, calculations, instructions, and a statement to stop the program.

In lines 10 and 50, you establish a loop through which the program runs five times and then stops.

```
10 FOR I=1 TO 5
50 NEXT I
```

In line 20, the program requests input from you. You supply the values of the variables B and C during program execution.

```
20 INPUT B,C
```

In line 30, you place the result of the addition in variable A.

```
30 LET A=B+C
```

Without line 40, your program would run but you would not see the results.

```
40 PRINT "B+C= ";A
```

Line 60 ends your program. The END statement is optional.

```
60 END
```

Shifting and transferring control, looping and supplying data are all part of a BASIC program.

1.3 BASIC CHARACTER SET

If you look closely at a BASIC program, such as the one previously shown, you will notice that it consists of letters, numbers, and symbols arranged in a certain syntax. These characters can be considered the alphabet of the BASIC language.

BASIC uses the full ASCII (American Standard Code for Information Interchange) character set for its alphabet (see the ASCII Table in Appendix C). This set consists of:

1. Upper-case letters A through Z
2. Lower-case letters a through z
3. Numbers 0 through 9
4. Special characters
5. Nonprinting characters

This character set enables you to include any ASCII character as part of a program. BASIC translates what you type. Some characters are processed and some are ignored.

BASIC translates characters in the following manner:

1. Letters a through z - BASIC translates all lower-case ASCII characters to upper-case characters.

2. Non-printing (i.e., control characters) and null characters (i.e., space, tab) - BASIC ignores them.

3. All other characters - BASIC accepts these characters unchanged.

String constants are a different matter (as described in Section 2.2.3). Everything you type into a string constant is interpreted literally by BASIC. Consequently, in a string constant:

1. All lower-case alphabetics (a,b,c) remain lower-case.

2. All non-printing and null characters are accepted, including spaces and tabs.

BASIC also accepts all characters in a REM statement (see Section 1.6).

System editing characters affect terminal output format only. Therefore, you need not be concerned, at this point, with the way BASIC handles them. (System editing characters, such as CTRL/U are described in Section 9.1.)

1.4 LINE FORMAT

The format of a line in a BASIC program is as follows:

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

Every line in a BASIC program must begin with a number. This number must be a positive integer within the range 1 to 32767 inclusive. 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.

Leading zeroes, as well as spaces, have no effect on the number. For example, these numbers are all the same to BASIC:

```
00010
   10
0 1 0
```

There are several reasons why BASIC requires line numbers:

1. To tell BASIC the order in which to execute the program.

2. To aid you in correcting and updating a program.

3. To provide a reference for conditional and unconditional transfers of control. (See Chapter 4)

You can use consecutive line numbers like 1,2,3 and 4. For example:

```
1 A=5
2 B=10
3 C=A+B
4 END
```

however, a useful practice is to write line numbers in increments of 10. This method allows you to insert additional statements between existing lines. The following program illustrates line number increments.

```
10 A=5
20 B=10
30 C=A+B
40 END
```

This program assigns values to two variables, adds them, and places the result in variable C. If you want the program to print the results of line 30, add the PRINT statement with a line number in the range 31 to 39 inclusive. For example, if you add a line numbered 33, the program looks like this:

```
10 A=5
20 B=10
30 C=A+B
33 PRINT C
40 END
```

Unlike integer constants (see Section 2.2.2), line numbers cannot have a percent sign (%).

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

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

can also be typed

```
10 LETA=B+C
```

or

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

These three lines are the same to BASIC; they will all be listed as:

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

In BASIC, you terminate a line by pressing the RETURN key. Pressing this key provides a carriage return/line feed sequence.

1.5 STATEMENTS

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

1. Executable statements specify the action of a program by telling BASIC what operation to perform (i.e. PRINT, GO TO, READ).

2. Non-executable statements describe the characteristics and arrangement of data, editing information, and statement functions that you include in your program (i.e., DATA and REM).

1.5.1 Single and Multi-Statement Lines

You have the option, with BASIC, of writing either one statement or many statements on one line. However, you cannot continue a BASIC statement from one line to the next.

A single statement line consists of:

1. A line number from 1 to 32767

2. A statement keyword

3. The body of the statement

4. A line terminator (RETURN key)

When typing your program, end each line by pressing the RETURN key.

This is a single statement line:

```
10 LET A=B*C/G*F
```

To enter more than one statement on a single line (multi-statement line), separate each complete statement with a backslash (\). This 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
```

There is only one line number for a multi-statement line. Consequently, you should take this into consideration if you plan to transfer control to a particular statement within a program. 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 discussion of individual statements in this manual.

A multi-statement line consists of:

1. A line number from 1 to 32767
2. A statement keyword
3. The body of the statement
4. A backslash (\)
5. As many repetitions of 2, 3, and 4 as you want
6. A statement keyword
7. The body of the statement
8. A line terminator (RETURN key)

1.6 DOCUMENTING PROCEDURES (REM STATEMENT)

BASIC allows you to document your methods, insert notes and comments, or leave yourself messages in your program. This type of documentation is known as a remark or comment. BASIC provides the REM statement for this purpose. The REM statement has the following format:

```
REM comment
```

where:

comment is anything you want to say.

You may place a REM statement anywhere in your program because it does not affect program execution. Remarks do, however, use memory area which you may need for exceptionally long programs.

The REM statement can be either the only statement on the line

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

or it can be one of several statements in a multi-statement line.

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

BASIC ignores anything in a line following the keyword REM until it reaches a backslash (\) or a line terminator. The backslash terminates the REM statement as it does all other statements in BASIC. Obviously, the only printing character you should not include in a remark is a backslash.

You can use the line number of a REM statement in a reference from another statement (i.e., GO TO); however, in this case, BASIC ignores the REM statement and proceeds to execute the next non-REM statement following the line referenced. (See Section 4.1.1 for the GO TO statement.)

Remember that BASIC prints the remarks on the terminal only when you list the program. (See Section 9.2 for a description of the LIST command.)

1.7 ENTERING BASIC PROGRAMS

It is very easy to enter BASIC programs. Simply type the program lines in the format described in Section 1.4. (Start each line with a line number and terminate with the RETURN key.) You can type the lines in any order, BASIC stores them in numeric order by line number. For example, you can type:

```
10 A=5
20 B=10
30 C=A+B
40 PRINT C
50 END
```

Or you can type:

```
50 END
20 B=10
40 PRINT C
10 A=5
30 C=A+B
```

They are equivalent to BASIC.

If you type a line and want to change it, simply type a new line with the same line number. BASIC replaces the old line with the new. For example, if you type the program listed above and want to change line 30 to be A*B instead of A+B, simply type:

```
30 C=A*B
```

If you want to delete a line completely, simply type the line number followed by a RETURN key. To delete line 50, the statement that ends the program, type:

```
50
```

If you are typing a line and realize that you have made a mistake before you type the RETURN key, there are two ways to correct the error. You can type the RUBOUT key, which deletes the last character that you typed, until you have deleted the characters which were errors and then retype the rest of the line. Or you can type CTRL/U which tells BASIC to ignore the entire line. See Section 9.1 for more information on these and other key commands.

BASIC stores your program in an area in the computer's memory. Each time you enter a new line, BASIC stores it in your area. Each time you delete a line, BASIC erases it from your area. Each time you replace a line, BASIC erases the existing line and stores the new line.

BASIC has a set of commands which allow you to list, execute, or modify the program in your program storage area. BASIC commands consist of a keyword followed by optional specifications. Do not type a line number before a command. (See Section 1.8 for a description of immediate mode statements, statements which are not preceded by a line number.)

One BASIC command is the LIST command. If you want to see the program lines that you have already entered, type the LIST command. Type

```
LIST
```

BASIC prints out a header line followed by the lines you have typed. When it is done, it prints the READY message. The READY message means that BASIC is ready to accept a program line, command, or immediate mode statement. For example:

```
LIST

NONAME       30-JUL-76  15:43:27

10 A=5
20 B=10
30 C=A*B
40 PRINT C

READY
```

Another BASIC command is the RUN command. If you want BASIC to execute your program, type the RUN command. After BASIC executes your program, it prints the READY message. For example:

```
RUN

NONAME       30-JUL-76  15:43:44

 50

READY
```

If you want BASIC to list the program lines or to execute your program without printing the header line, type LISTNH or RUNNH, respectively (NH stands for No Header).

BASIC has commands to:

- List your program (LIST and LISTNH).
- Execute your program (RUN and RUNNH).
- Delete program lines (DEL).
- Erase your program (NEW, SCR, and CLEAR).
- Save your program (SAVE and REPLACE) to a file.
- Restore your program (OLD and APPEND) from a file.
- Delete programs saved in files (UNSAVE).
- Change the program name (RENAME).
- Edit a line that you have entered (SUB).
- Resequence the line numbers in your program (RESEQ).
- Save a compiled program (COMPILE).
- Find out how big your program is (LENGTH).

See Chapter 9 for a complete description of these commands.

BASIC always prints the READY message when it completes execution of a command.

1.8 USING BASIC WITHOUT WRITING A PROGRAM (IMMEDIATE MODE)

You do not have to write a program to use BASIC. BASIC can execute most statements as soon as you type them. To have BASIC execute a statement immediately, type the statement without a line number. Statements typed without a line number are called immediate mode statements. Immediate mode statements differ from commands in that you can type the same statement with a line number but a command with a line number is meaningless.

If you type the statement PRINT 4+5 with a line number, BASIC stores the program line for later execution. For example:

```
10 PRINT 4+5
```

But if you type the line without a line number BASIC executes the line and then prints the READY message.

```
PRINT 4+5
 9

READY
```

You can enter several immediate mode statements in a row. For example:

```
LET A=5

READY
PRINT A*12
 60

READY
```

Or you can enter several immediate mode statements on one line. Separate each statement with a backslash. When you type the RETURN key, BASIC executes all the statements on the line. For example:

```
A=5 \ B=14 \ C=.3729 \ PRINT A*B,SIN(C)
 70             .364138

READY
```

You can use BASIC as a powerful calculator by using immediate mode. For example after typing the previous immediate mode statement, type:

```
PRINT "THE HEIGHT IS";B*SIN(C)+A;"METERS"
THE HEIGHT IS 10.1004 METERS

READY
```

You can use FOR-NEXT loops (see Section 4.2.1) in immediate mode if you can enter the entire loop on one line. For example:

```
FOR I= 1 TO 10 \ PRINT I, SQR(I) \NEXT I
 1             1
 2             1.41421
 3             1.73205
 4             2
 5             2.23607
 6             2.44949
 7             2.64575
 8             2.82843
 9             3
 10            3.16228

READY
```

Another use of immediate mode is to debug (find and correct the errors in) your program. First you execute your program with a RUN command. Then you can use immediate mode PRINT statements to find the values of variables in the program. You can also use the LIST command to look at the program lines. After making any needed changes you can continue the program with an immediate mode GO TO statement (see Section 4.1.1). For more information on program debugging, see the description of the STOP statement in Section 4.3 and the description of error messages in Appendix A.

You can use most BASIC statements in immediate mode. You can not use the INPUT or LINPUT statement (see Sections 3.1.1 and 3.1.2) in immediate mode. If you do, BASIC prints the ?ILLEGAL IN IMMEDIATE MODE (?IIM) error message. Certain other immediate mode statements, though they do not cause an error message, are ignored by BASIC. These immediate mode statements that are ignored are COMMON, DATA, DEF, and DIM.

CHAPTER 2

ELEMENTS OF BASIC

2.1 TERMINOLOGY

In order to write programs in BASIC, you must be familiar with the terms and phrases used to describe the program elements. You will probably recognize most of these terms from previous experience; however, the following sections define these terms within the context of BASIC.

2.2 CONSTANTS

A constant is a quantity with a fixed value. In BASIC, you can enter a constant as part of a program or have BASIC read it from another file during program execution.

There are three types of constants in the BASIC language:

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

2.2.1 Numeric Constants

A numeric constant is one or more decimal digits, either positive or negative, in which the decimal point is optional. BASIC assumes a decimal point exists to the immediate right of the number if you do not include one. For example, the numeric constant

```
184
```

is equivalent to

```
184.
```

The following are all valid numeric constants:

```
 5        42861
74         -125
 6.           .95
```

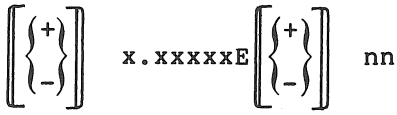
BASIC accepts numeric constants within the approximate range

```
10^-38<n<10^+38
```

where n is the numeric constant you specify.

If you type a numeric constant in a program that is outside this range, BASIC prints a fatal error message to that effect. This means that your program will not execute until you replace the numeric constant with one in the proper range.

However, you can input very large numbers and very small numbers (within this range) by using a method similar to scientific notation. Use the following format:




```
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 a digit from 0 to 9.

. is the decimal point.

E represents the words "times 10 to the power of".

nn is the 2-digit exponential value (the power of 10).

This method of mathematical shorthand is called E 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 whole number. The integer constant is the exponent. It can be 0 but never blank. Thus you can type:

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

You are actually positioning the decimal point internally by using E notation. 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 2-1 shows the different methods of writing the same number.

Table 2-1
Number Notations

STANDARD NOTATION	SCIENTIFIC NOTATION	E NOTATION
1000000	1 X 10^6	1.00000E+06
10000000	1 X 10^7	1.00000E+07
100000000	1 X 10^8	1.00000E+08
1000000000000	1 X 10^12	1.00000E+12

BASIC uses floating point format when storing and calculating most numbers. Integers, however, are handled in a slightly different manner. (See Section 2.2.2.)

The following are examples of numeric constants:

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

BASIC stores numbers to a certain degree of accuracy (or precision). See your BASIC-11 user's guide for the accuracy of numbers.

2.2.2 Integer Constants

An integer constant is a whole number (no fractional part) written without a decimal point. BASIC adds a decimal point internally to the right of the integer. Therefore, to distinguish a numeric constant (floating point number) from an integer constant (whole number), type an integer constant as 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
34 1/2        95
```

You should use integers instead of floating point numbers to store whole numbers because integer storage requires less memory and integer arithmetic operations take less execution time than the equivalent floating point operations. Saving space is most important when you are using large arrays (see Section 2.6).

In BASIC you can type integer constants within the range

```
-32768%  to  +32767%
```

If you specify a number outside this range, BASIC prints a fatal error message telling you that you must replace the number with one within the proper limits.

2.2.3 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.

Each character in a string constant can be a letter, a number, a space, or any ASCII character except a line terminator (RETURN key). The value of the string constant is determined by all the characters between the delimiters including spaces.

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

1. Lower-case letters (a-z)

2. Leading, trailing, and embedded spaces

3. Tabs

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

```
LISTNH
10 PRINT "DIGITAL"
20 END

READY
RUNNH

DIGITAL

READY
```

In order to make BASIC print quotation marks, you must enclose them within another pair of quotation marks, either double or single.

```
LISTNH
10 PRINT 'HE SAID, "GOOD MORNING!"'
20 END

READY
RUNNH

HE SAID, "GOOD MORNING!"

READY
```

Here are some examples of string constants:

```
"THIS IS A STRING CONSTANT."
'SO IS THIS.'
"TONY'S TENNIS RACKET"
```

Include both the starting and ending delimiters when typing a string constant in a program. These delimiters must be of the same type (both double quotation marks or both single quotation marks).

These examples are incorrect:

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

2.3 VARIABLES

A variable is an unknown quantity that may change during program execution. In BASIC, each variable refers to a distinct location in the computer's memory. Each location holds one value at any one time. The number it holds is the value of the variable corresponding to the location.

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

BASIC accepts three types of variables:

1. Simple numeric variables (floating point)
2. Integer variables
3. String variables

2.3.1 Numeric Variables

A numeric variable is a named location in which a single numeric value (floating point number) is stored. You name a numeric variable with a single letter or a single letter followed by a single digit. For example, the following are simple numeric variables:

C1	L
M	B5
F6	Z2

The following are not numeric variables:

6	A.2
BC	.E
4D	25

Before program execution, BASIC sets all numeric variables to 0. If you require an initial value other than 0, you can assign it with the LET statement (Section 2.5). 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 0 before program execution, you should not rely on this feature. A good programming practice to follow is to set all variables to 0 at the beginning of the program. You can do this with a series of LET statements (see Section 2.5) or by using READ and DATA statements (see Section 3.1.3).

You can assign a new value to a variable at any point in your program. BASIC always uses the most recent value that you have assigned.

2.3.2 Integer Variables

An integer variable (like a numeric variable) is a named location in which a single value can be stored. Using an integer variable in your program indicates that the value forthcoming is a whole number (no fractional part).

You name an integer variable with a single letter or a single letter and a single digit, terminated by a percent sign (%). For example, the following are integer variables:

```
 A%        C8%
B1%        D%
```

The following are not integer variables:

```
A      B2
1B%    123%
```

If you include an integer variable in a program, then the value you supply for it must be an integer constant. If a numeric constant (floating point 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. Consequently, you should not assign a nonintegral value to an integer variable, but if you do you must be aware how the truncation will affect your calculations.

You can use an integer variable anywhere that you can use a numeric variable as long as you only plan to store whole number values in the range -32768 to 32768.

If you assign an integer constant to a numeric variable, BASIC prints the integer value as an integer but stores the real number internally. This method takes more storage space because of the fraction that BASIC must maintain.

NOTE

You can use numeric and integer variables with the same leading alphanumerics in the same program. D1 and D1% represent two different variables.

2.3.3 String Variables

A string variable is a named location used to store alphanumeric strings. You name a string variable with a letter, an optional digit, and a dollar sign ($). The dollar sign ($) must be the last character in the name. The following are examples of string variables:

```
C1$
L$
Z2$
M$
F6$
```

These are not string variables:

```
Cl
L
2Z$
$M
F6
```

The dollar sign ($) not only represents a string variable to BASIC, but it also indicates a string variable to anyone who reads the program.

BASIC initializes all string variables to a length of 0 (null string) before the start of each program execution. During execution, the length of a character string associated with a string variable can vary from 0 (signifying a null or empty string) to 255 characters.

Note that a simple numeric variable, an integer variable, and a string variable that begin with the same alphanumeric characters can represent three distinct variable names. The following names are all legal within a single BASIC program:

```
A5    a simple numeric variable
A5%   an integer variable
A5$   a string variable
```

2.3.4 Subscripted Variables

A subscripted variable is a numeric variable, an integer variable, or a string variable with one or two subscripts appended to it. The subscripts can be any positive expression (see Section 2.4.1). The value of the subscript can be 0 to 32767.

The subscript, in a subscripted variable, is a pointer to a specific location in a list or table in which an unknown value is stored. (See Section 2.6 for more information on lists and tables, also known as arrays.) 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 numeric, integer, or string data.

To name a subscripted variable, start with a simple numeric, 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 because the lists start with the item with a 0 subscript:

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

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)
```

In the following table, the arrow indicates the element pointed to by the subscripted variable A%(4,6):

```
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<-------A%(4,6)
```

Notice that all the elements in this table have a value of 0.

BASIC accepts the same alphanumeric characters for a simple numeric variable and a subscripted variable within the same program. However, do not use the same alphanumeric characters for two arrays (Section 2.6) with a different number of subscripts.

This is acceptable in the same program:

```
D      simple numeric variable

D(8)   subscripted variable
```

This is not acceptable in the same program:

```
D(8)      one subscript

D(3,6)    two subscripts
```

2.4 FORMING EXPRESSIONS

An expression is a symbol or a group of symbols that BASIC can evaluate. These symbols can be numbers, strings, constants, variables, functions (Section 2.4.4), array references (Section 2.6), or any combination of these, separated by one of the following:

1. Arithmetic operators (to form arithmetic expressions)
2. Relational operators (to form relational expressions)
3. String operators (to form string expressions)

2.4.1 Arithmetic Expressions

BASIC allows you to perform addition, subtraction, multiplication, division, and exponentiation with the following operators:

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

Performing an 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 floating point expression
```

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

```
A*B% = a floating point expression

6.8*5% = 34.0
```

The value of an integer expression is truncated. For example 3%/5% is equal to 0% not .6.

Table 2-2 provides examples of arithmetic operators and their meaning.

Table 2-2
Arithmetic Operators

OPERATOR	EXAMPLE	MEANING
+	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

Note that in general, you cannot place two arithmetic operators consecutively in the same expression. The exception is the unary minus. For example:

```
A*-B is valid  and A*(-B) is valid.
```

BASIC evaluates expressions according to operator precedence. Each arithmetic operator joining 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.

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).

Table 2-3 lists the arithmetic operators in the order BASIC evaluates them:

Table 2-3
Arithmetic Operator
Precedence

^	HIGHEST
- (unary minus)	
*,/	
+,-	LOWEST

Operators shown on the same line have equal precedence. BASIC evaluates operators of the same precedence level from left to right. Note that 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. (35*8) = 280 Parenthetical expression
2. 15^2 = 225 Exponentiation (left-most expression)
3. 12^2 = 144 Exponentiation
4. 225+144= 369 Addition (left-most expression)
5. 369-280= 89 Subtraction

2.4.2 String Expressions

BASIC provides the plus sign (+) (and the equivalent ampersand (&)) as an operator for string expressions. By using this operator you can attach one string to the end of another. This operation is called concatenation.

Consider the following example:

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

READY
RUNNH

GOODBYE

READY
```

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

2.4.3 Relational Expressions

A relational operator is a symbol used to compare one value of a variable or expression to another within a BASIC program, thus creating a relational expression. As explained in Section 4.1.3, use relational expressions with the IF THEN statement to create conditional transfers.

NOTE

It is illegal to compare a numeric or integer expression to a string expression using a relational operator. In a relational expression, both expressions being operated on must be of the same data type, string or numeric. But you can compare an integer expression to a floating point expression and vice versa.

Table 2-4 provides examples of arithmetic relational operators and their meaning.

Table 2-4
Arithmetic 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

BASIC accepts =< but converts it to <=; => to >= and >< to <>.

When you use a relational operator to compare the value of one or more alphanumeric characters, 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 C for the ASCII Table.) The comparison is made, character by character, left to right, according to the ASCII values until BASIC finds a difference in value.

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

```
10 A$="ABC"
20 B$="DEF"
30 IF A$<B$ GO TO 50
40 PRINT B$ \ GO TO 60
50 PRINT A$
60 END
```

When BASIC executes line 30, it compares strings A$ and B$ to determine if A$ occurs first in alphabetic sequence. In this case, it does, and the program shifts control to line 50 (see Section 4.1 for shifting control of a program). If string B$ occurred before string A$, program execution would continue to the next statement following the comparison (i.e., line 40).

BASIC compares strings just as you compare words in alphabetical order. BASIC compares the first two characters, 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 2-5 provides examples of string relational operators and their meaning.

Table 2-5
String Relational Operators

OPERATOR	EXAMPLE	MEANING
=	A$ = B$	Strings A$ and B$ are equal
<	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$

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. This means that "DIGITAL" is equal to "DIGITAL ".

2.4.4 Functions

Functions perform a series of mathematical or string operations. You provide the arguments, the input to the function, and the function computes the result. You can use functions instead of doing tedious calculations yourself.

To use a function, simply include the function name and argument list in any expression. The function name is usually three letters, for example, SQR, SIN, and POS. The argument list follows the function name and is enclosed in parentheses. The number and type of arguments are listed in the description. You use a function in an expression in the same way that you include a constant or a variable in an expression.

The function calculates and returns the result to BASIC. BASIC continues to evaluate the expression as if you had specified the result in place of the function. Consider the following examples which demonstrate the SQR function which returns the square root of the argument.

Program with SQR Function

```
LISTNH
10 A=10*SQR(49)+5
20 PRINT A

READY
RUNNH

 75

READY
```

Program with Numeric Constant

```
LISTNH
10 A=10*7+5
20 PRINT A

READY
RUNNH

 75

READY
```

The square root of 49 is 7. In the example on the left, BASIC treats the 7 that the function returns in the same way that it treats the 7 specified by the user in the program on the right.

BASIC provides mathematical and string functions and, in addition, lets you define your own functions. See Chapter 5 for a complete description of all BASIC functions.

2.5 ASSIGNING VALUES TO VARIABLES (LET STATEMENT)

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

[LET] variable = expression

where:

variable — is assigned a new value.

expression — specifies the new value.

The variable and expression can either both be numeric or both be string data types. (The keyword LET is optional.)

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. Here is an example:

```
10 LET A=482.5
```

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

```
10 A=482.5
```

BASIC also evaluates any formula 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, BASIC converts the mode of the value to whatever the mode of the variable is, floating point or integer. For example,

```
10 A%=9.5
```

is the same as

```
10 A%=9
```

Refer to Section 2.3 for a description of variables.

You can also assign a string expression to a variable as well as a numeric expression. However, you cannot mix strings and numeric expressions in the same LET statement. If you do, BASIC prints the ?NUMBERS AND STRINGS MIXED (?NSM) error message. The following is an example of a string assignment:

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

READY
RUNNH

HELLO

READY
```

Refer to Sections 2.2.3 and 2.3.3 for information on strings.

Note that you can place a LET statement anywhere in a multi-statement line:

```
10 DIM A(7) \ I=42 \ PRINT I
```

2.6 ARRAYS

An array is like a group of variables. Each element in the array, like a variable, is an unknown quantity. BASIC stores the current value of each element of the array in a location in memory in the same way that it stores a value for a variable. An element of an array is different from a variable in that all the elements of an array share the same name but each variable has its own name. You specify which element you want in an array by specifying its subscript (see Section 2.3.4).

Arrays break down into two types: lists and matrices. A list is a horizontal or vertical group of items (1-dimensional); a matrix is a table of items consisting of rows and columns (2-dimensional). Both types can store either numeric or string data.

You should reserve space for any array you use with a DIM statement. But if you do not reserve space, BASIC reserves space for arrays with a maximum subscript(s) of 10 (i.e., A(10) and A(10,10).

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

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

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

Figure 2-1 ARRAY B

Note that if you reference an array with the wrong number of subscripts, BASIC prints the ?INCONSISTENT NUMBER OF SUBSCRIPTS (?INS) message.

You should use an array instead of many variables when you are going to be doing the same operations to each element. This allows you to use loops (see Section 4.2) to perform the operation. Consider these two examples:

Program with Array

```
5 DIM A(10)
10 FOR I=0 TO 10
20 LET A(I)=I
30 NEXT I
```

Program with Separate Variables

```
5 A=0
10 B=1
20 C=2
30 D=3
40 E=4
50 F=5
60 G=6
70 H=7
80 I=8
90 J=9
100 I=10
```

As you can see, the program with the array is much shorter. (See Section 2.6.1 for a description of the DIM statement.) With larger arrays the difference would be even greater. Arrays also require less memory to store than an equivalent number of variables.

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 illegal within the same program (i.e., A(5) and A(3,4) are illegal in the same program).

2.6.1 Dimensioning Arrays (DIM Statement)

The DIM statement allows you to set up 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.

The DIM statement has the following format:

```
line number DIM list
```

where:

list — is a list of array specifications separated by commas. Each array specification is in the form:

```
variable(integer1 [,integer2])
```

where each integer must be a whole number constant (no decimal point) but does not have to have a percent sign.

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

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.

Arrays are stored as if the right-most subscript varied the fastest. For example:

```
10 DIM A(2,5)
```

provides storage space like this:

```
0,0  0,1  0,2  0,3  0,4  0,5  1,0  1,1  ...  2,4  2,5
```

When using the DIM statement to set the maximum values for the subscripts, you are not obligated to fill every storage space you allocate.

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.

The following example sets up storage for a matrix with 20 elements:

```
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:

```
LISTNH
10 REM MATRIX CHECK PROGRAM
20 DIM C(6,10)
30 FOR I=0 TO 6
40 LET C(I,0)=I
50 FOR J=0 TO 10
60 LET C(0,J)=J
70 PRINT C(I,J);
80 NEXT J \ PRINT \ NEXT I
90 END

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
```

Notice that a variable has a value of 0 until you assign it another value.

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

```
LISTNH
10 DIM A$(5)
20 FOR I=0 TO 5
30 INPUT A$(I)
40 NEXT I
50 PRINT A$(0),A$(5)

READY
RUNNH

? NAME
? ACCOUNT
? AGE
? BALANCE
? STATUS
? LOCATION
NAME            LOCATION

READY
```

CHAPTER 3

INPUT AND OUTPUT

3.1 SUPPLYING DATA

BASIC has three methods of supplying data to a program:

1. The INPUT statement - requires that you interact with the computer while the program is running.

2. The READ, DATA, and RESTORE statements - require that you build a data block within the program.

3. The file statements - require that you manipulate files outside the main program. See Chapter 6 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 variable1 [,variable2, variable3,...]

where:

variable(s) are assigned the value(s) that you enter.

The variables can be numeric, integer, string, or subscripted variables or any combination of these separated by commas. Consider the INPUT statement as another means of assigning values to variables.

When you run your program, BASIC stops at the line designated by the INPUT statement and prints a question mark (?). BASIC then waits for you to type one value for each variable requested in the INPUT statement. Separate the values with commas. Press the RETURN key after you finish typing all the values.

The following example requires that you type three values after the question mark (?).

```
LISTNH
10 INPUT A,B,C
20 END

READY
RUNNH

? 5,6,7          Your Response

READY
```

The INPUT statement tells BASIC to accept the forthcoming data from the user terminal. BASIC accepts the values left to right. When you type all the necessary data, type a line terminator (RET). 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 another question mark (?) when you press the RETURN key.

```
LISTNH
10 INPUT A,B
20 END

READY
RUNNH

? 5
? 6

READY
```

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 to that effect.

```
LISTNH
10 INPUT A,B,C
15 PRINT A,B,C
20 END

READY
RUNNH

? 5,6,7,8
?EXCESS INPUT IGNORED AT LINE 10
 5             6             7

READY
```

The extra value entered (8) is ignored.

The values you supply must be the same data type as the variables in the INPUT statement, (i.e., strings for string variables, integers for integer variables). You can type strings with or without quotation marks. For example,

```
LISTNH
10 INPUT A,B$,C
20 PRINT A+C;B$
30 INPUT A$,B$
40 PRINT A$
50 PRINT B$

READY
RUNNH

? 5,DOGS,7
12 DOGS
? "ARE",FED
ARE
FED

READY
```

If you include quotation marks delimiting a string, be sure to type both beginning and ending delimiters. If you forget the end quotation mark, BASIC reads the rest of the line as the entire string. You will also receive an ?INPUT STRING ERROR (?ISE) message. For example:

```
LISTNH
10 INPUT A$,B
20 END

READY
RUNNH

? "ABC,5
?INPUT STRING ERROR AT LINE 10
?
```

BASIC reads ABC,5 as the string for variable A$, and then requests data for variable B.

If you type a string without quotation marks, BASIC ignores any leading or trailing spaces.

You must specify a whole number (with or without a percent sign) for an integer variable. If you specify a number with a decimal point or a fraction for an integer variable, BASIC prints the ?BAD DATA RETYPE (?BRT) error message and requests the input again.

You may wonder how you can tell what kind of data to respond with when all you see is a question mark (?). By adding a PRINT statement (see Section 3.2) you clarify the program's request for data. This is useful because

1. You may not remember the number of variables or their type (numeric or string).

2. Someone else may be running your program and does not know what the program is asking.

Preceding the INPUT statement with a PRINT statement is good programming practice. Following the INPUT statement with a PRINT statement allows you to see the results of your computations.

```
LISTNH
20 PRINT "PLEASE TYPE 3 INTEGERS";
30 INPUT B%,C%,D%
35 A%=B%+C%+D%
40 PRINT A%
50 END

READY
RUNNH

PLEASE TYPE 3 INTEGERS? 25,50,75
 150

READY
```

NOTE

The INPUT # statement (see Section 6.3.1) is used to input values from a file. Logical unit 0 is the user terminal.

```
10 INPUT #0, X,Y,Z
```

is equivalent to

```
10 INPUT X,Y,Z
```

except that a question mark (?) does not print on the terminal with the first form.

3.1.2 LINPUT Statement

The LINPUT statement has essentially the same function as the INPUT statement. However, LINPUT is used exclusively for string data. Use the following format:

```
LINPUT string variable1 [,string variable2,...]
```

where:

string variable(s) assigned the value(s) of all the characters you type up to the line terminator(s).

All variables must be string variables in a LINPUT statement.

The LINPUT statement accepts and stores all characters including quotation marks and commas except the line terminator. The terminator is not stored with the string.

```
LISTNH
10 LINPUT B$
20 PRINT B$
30 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, you will receive a ?EXCESS INPUT IGNORED (?EII) error message from BASIC. The INPUT would take the comma after the word "HERE!", as the delimiter of the string.

3.1.3 READ, DATA, and RESTORE Statements

Another way you can supply data to a program is to build a data block for BASIC to read during execution. This means that you do not interact with BASIC while the program is running. Instead, you supply a pool of data to the program in advance. Two statement keywords are involved in this process: READ and DATA.

The READ statement has the following format:

READ variable1[,variable2,variable3,...]

where:

variable(s) are assigned the value(s) listed in the DATA statements.

All variables should be separated by commas.

```
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.

The DATA statement has the following format:

line number DATA constant1 [,constant2,constant3,...]

where:

constant(s) can be numeric, integer, or string (quoted or unquoted) constants.

The order of the data types of the constants must be the same as the data types of the variables when they are read. The format of the constant is the same format you use for the INPUT statement.

The program will run faster with READ and DATA statements as opposed to the INPUT statement simply 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 legal without at least one DATA statement. However, you can have more than one DATA statement as long as there is one READ statement in the program.

```
10 READ A,B,C,D,E,F
20 PRINT C
30 DATA 17,25,30
40 DATA 43,76,29
```

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.

If you build your READ statement with more variables than you include in the data block, BASIC prints the ?OUT OF DATA (?OOD) error message.

The following is an example of a READ and DATA sequence.

```
10 READ A,B,C1,D2,E4,Y$,Z$,Z1$
20 DATA 2.3,-4.2654,3,-6,12,"CAT",DOG,"MOUSE"
30 PRINT A,B,C1,D2,E4,Y$,Z$,Z1$
```

BASIC assigns values as follows:

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

When you run the program, you get the following results:

```
RUNNH

 2.3          -4.2654        3             -6             12
CAT           DOG            MOUSE

READY
```

READ and DATA are useful to initialize the values of variables and arrays at the beginning of your program. To do this place your READ statements at the beginning of the program. You can put the DATA statements anywhere in the program but it is useful to put them all at the end of the program just before the END statement.

You can read a numeric constant into a string variable. For example:

```
LISTNH
10 READ A$
20 PRINT A$
30 DATA 8.25
40 END

READY
RUNNH

8.25

READY
```

But if you try reading a string constant into a numeric variable or a floating point number into an integer variable, BASIC prints the ?BAD DATA READ (?BDR) error message.

In some programs you may need to read the same data more than once. BASIC provides the RESTORE statement for this purpose.

The format of the RESTORE statement is:

```
RESTORE
```

The RESTORE statement resets the data pointer to the beginning of the first DATA statement in the program. The values are read as though for the first time; therefore, the same variable names may be used the second time through the data. Consider this example:

```
10 READ B,C,D
20 RESTORE
30 READ E,F,G
40 DATA 6,3,4,7,9,2
50 END
```

The READ statement in line 10 reads the first three values in the DATA statement, line 40.

```
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 on line 30 reads the first three values. BASIC reads these values as though for the first time.

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

If RESTORE was not there, READ on line 30 would read the last three values.

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

NOTE

The RESTORE # statement (see Section 6.3.4) is used to restore files to their beginning.

3.2 CHECKING OUTPUT (PRINT STATEMENT)

Another useful statement to include in your program is the PRINT statement. The PRINT statement has the following format:

PRINT [list]

where:

list — contains the items to be printed. They can be any string or numeric expressions or TAB functions (see Section 3.2.3) and can be separated by commas or semicolons (see Section 3.2.1).

The PRINT statement prints a list of elements on the terminal when you execute your program. In this way, you can see the results of your computations or add comments to clarify your requests for input. (The PRINT statement can be placed anywhere in a multi-statement line.)

Using the PRINT statement without arguments causes a blank line to appear in the output.

```
LISTNH
10 PRINT "THIS EXAMPLE LEAVES A BLANK LINE"
20 PRINT
30 PRINT "BETWEEN TWO LINES"
40 END

READY
RUNNH

THIS EXAMPLE LEAVES A BLANK LINE

BETWEEN TWO LINES

READY
```

You can print blank lines to improve the readability of your output.

When an element in the list is an expression rather than a simple variable or constant, BASIC evaluates the expression before printing the value. Therefore, the PRINT statement performs two functions in one, calculating expressions and printing the results. For example:

```
LISTNH
10 A=45 \ B=55
20 PRINT A+B
30 END

READY
RUNNH

 100

READY
```

After running this program, BASIC prints 100 on your terminal, not 45+55. If you put quotes around the variables this is what happens:

```
LISTNH
10 A=45 \ B=55
20 PRINT "A+B"
30 END

READY
RUNNH

A+B

READY
```

If you plan to have someone else run your program, you can clarify your requests for input with a PRINT statement. (Refer to Section 3.1.1 for more information on the INPUT statement.) Include literal strings (Section 2.2.3) as in the following example:

```
10 PRINT "WHAT ARE YOUR VALUES OF X,Y, AND Z";
20 INPUT X,Y,Z
30 LET R=SQR(X^2+Y^2+Z^2)
40 PRINT "THE RADIUS VECTOR EQUALS";
45 PRINT R
50 END
```

When you run this program, BASIC prints:

```
RUNNH

WHAT ARE YOUR VALUES OF X,Y,Z? _25,40,50
THE RADIUS VECTOR EQUALS 68.7386

READY
```

Notice that you enclose the strings in quotation marks so that BASIC prints them exactly as you type them in. In line 40 of the previous example, a semicolon ends the list of expressions. Placing a semicolon or a comma after the string makes BASIC print the expressions in the next PRINT statement (line 45) on the same line as the string. If the separator is not there, BASIC performs a carriage return/line feed and begins printing in the first column of the next line.

```
THE RADIUS VECTOR EQUALS
 68.7386
```

See Section 3.2.1 for more information on commas and semicolons.

3.2.1 Printing Zones - The Comma and the Semicolon

A terminal line consists of an integral number of zones, each zone containing 14 spaces. When you use the PRINT statement, you can control the placement of your output within these zones by using the legal separators, comma (,) and semicolon (;). (See Section 7.1 for a description of the PRINT USING statement for more flexibility in formatting output.)

The comma signals BASIC to move the printing element to the beginning of the next print zone and begin printing there. 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:

```
LISTNH
5 INPUT A,B,C,D,E,F
10 PRINT A,B,C,D,E,F
20 END

READY
RUNNH

? 5,10,15,20,25,30
 5             10            15            20            25
 30

READY
```

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.

```
LISTNH
10 A=5 \ B=10
20 PRINT A,,B
30 END

READY
RUNNH
 5                            10

READY
```

To print an output line in a more compact format, use the semicolon as the separator between variables. A semicolon in a PRINT statement causes no motion of the printing element.

```
LISTNH
10 A=5 \ B=10
20 PRINT A;B
30 END

READY
RUNNH

 5  10

READY
```

Placing a comma or semicolon after the last item in a PRINT statement causes the terminal printer to remain at the same line in anticipation of another PRINT statement.

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

```
LISTNH
10 INPUT X,Y,Z
20 PRINT X,Y,
30 PRINT Z
40 END

READY
RUNNH

? 5,10,15
 5             10            15

READY
```

The following example illustrates the three options you have for placing either a comma, a semicolon, or nothing after the last item of the PRINT statement:

```
LISTNH
10 FOR I=1 TO 10
20 PRINT I
30 NEXT I \ PRINT
40 FOR J=1 TO 10
50 PRINT J,
60 NEXT J \ PRINT
70 FOR K=1 TO 10
80 PRINT K;
90 NEXT K
100 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

READY
```

Commas and semicolons also allow you to control the placement of string output. For example:

```
LISTNH
10 PRINT "FIRST ZONE",,"THIRD ZONE",,"FIFTH ZONE"
20 END

READY
RUNNH

FIRST ZONE                    THIRD ZONE                    FIFTH ZONE

READY
```

Because of the extra comma between strings, BASIC skips every other printing zone before stopping to print each string. (Placing a semicolon between string constants is optional.)

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 with no leading or trailing spaces. (Quotation marks are not printed unless delimited by another pair.)

```
LISTNH
10 PRINT 'PRINTING "QUOTATION" MARKS'
20 END

READY
RUNNH

PRINTING "QUOTATION" MARKS

READY
```

BASIC precedes negative numbers with a minus sign and positive numbers with a space. A space is always placed after the right-most digit of a number. BASIC does not print a percent sign after printing an integer.

```
LISTNH
10 PRINT -1
20 PRINT 25;50%
30 END

READY
RUNNH
-1
 25  50

READY
```

BASIC does not add any spaces to strings.

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 numeric if they are within the range

$$.01<n<999999$$

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
.01	.01
.0099	9.90000E-03
999999	999999
1000000	1.00000E06

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

The following example shows how BASIC prints various numbers:

```
LISTNH
10 FOR I=1 TO 20
20 PRINT 2^(-I),I,2^I
30 NEXT I
40 END

READY
RUNNH

 .5             1             2
 .25            2             4
 .125           3             8
 .0625          4             16
 .03125         5             32
 .015625        6             64
 7.81250E-03    7             128
 3.90625E-03    8             256
 1.95313E-03    9             512
 9.76563E-04    10            1024
 4.88281E-04    11            2048
 2.44141E-04    12            4096
 1.22070E-04    13            8192
 6.10352E-05    14            16384
 3.05176E-05    15            32768
 1.52588E-05    16            65536
 7.62939E-06    17            131072
 3.81470E-06    18            262144
 1.90735E-06    19            524288
 9.53674E-07    20            1.04858E+06

READY
```

3.2.3 Printing with 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(expression);
```

where expression is the number of the desired printing position. BASIC evaluates the expression and truncates the result to an integer.

The TAB function does not cause anything to be printed; it positions the terminal print head. Then the PRINT statement takes over and begins printing in the column denoted by the argument with the TAB.

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 on your terminal, or anywhere in between.

The TAB function can only be used 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 ignores the TAB.

You can use more than one TAB function in the same PRINT statement by placing them between elements.

The following is an example of several TAB functions in conjunction with one PRINT statement:

```
LISTNH
10 PRINT "NAME"$TAB(15)$"ADDRESS"$TAB(30)$"PHONE NO."
20 END

READY
RUNNH

NAME            ADDRESS         PHONE NO.

READY
↑               ↑               ↑
Column 0        Column 15       Column 30
```

Without tabs 15 and 30, BASIC would print

```
RUNNH

NAMEADDRESSPHONE NO.

READY
```

Here is an example of printing numbers:

```
LISTNH
10 A=100 \ B=29 \ C=35
20 PRINT A$TAB(10)$B$TAB(40)$C
30 END

READY
RUNNH

 100        29                                  35

READY
↑          ↑                                   ↑
Column 0   Column 10                           Column 40
```

Notice that semicolons act as separators in the preceding example.

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

```
LISTNH
10 A=100
20 B=200
30 C=300
40 PRINT A,TAB(7),B,TAB(14),C

READY
RUNNH

 100           200           300

READY
```

The extra commas move the printer to the next zone (see Section 3.2.1), then the printer is past the position indicated by TAB and BASIC ignores the TAB.

Change line 40 to:

```
40 PRINT A;TAB(7);B;TAB(14);C
RUNNH

 100    200     300

READY
```

CHAPTER 4

CONTROL STATEMENTS

4.1 SHIFTING CONTROL OF THE PROGRAM

In a BASIC program, control ordinarily passes from one statement to the next statement in ascending order according to line numbers.

```
10 A=5*942 \ B=116/7 \ C=A*B
20 PRINT "C = "$C
30 PRINT
40 END
```

However, you may alter the normal sequence of statement execution to:

1. Repeat a set of statements
2. Stop and check the values
3. Terminate the program

You can divert execution from the main stream to another portion of a program; execution will continue from that point. This transferring of control is known as branching.

The following sections describe the statements that allow you to shift control and change the sequence of execution.

4.1.1 Unconditional Transfer (GO TO Statement)

The GO TO statement causes the statement which it identifies to be executed next, regardless of that statement's position within the program.

The format of the GO TO statement is:

```
GO TO line number
```

where:

line number specifies the next program line to be executed.

The specified line number can be smaller or larger than the line number of the GO TO statement. Thus, you have the option to skip any number of lines in either direction.

BASIC executes the statement at the line number specified by GO TO and continues the program from that point. Consider the example:

```
30 GO TO 110
```

When BASIC executes line 30, it branches control to line 110. BASIC interprets the statement exactly as it is written. Go to line 110. It is a simple imperative instruction. There are no rules or conditions governing the transfer.

Consider the following sample program with a GO TO statement:

```
LISTNH
10 A=2
20 GO TO 40
30 A=SQR(A+14)
40 PRINT A,A*A
50 END

READY
RUNNH

 2             4

READY
```

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 GO TO statement is either the only statement on the line or the last statement in a multi-statement line. If you place a GO TO in the middle of a multi-statement line, BASIC does not execute the rest of the statements on the line.

```
25 A=ATN(B2) \ GO TO 50 \ PRINT A
```

BASIC never executes the PRINT statement on line 25 because the GO TO statement shifts control to line 50.

If you specify a non-executable statement in a GO TO statement such as a REM statement, BASIC transfers control to the next executable statement after the one specified. For example:

```
LISTNH
10 A=2
20 GO TO 40
30 A=SQR(A+14)
40 REM NOW PRINT THE RESULTS
50 PRINT A,A*A
60 END

READY
RUNNH

 2             4

READY
```

At line 20, BASIC transfers control to line 40. Line 40 is a REM statement, a nonexecutable statement. BASIC executes the next sequential statement, line 50. So that the statement GO TO 40 in this case is equivalent to a statement GO TO 50.

NOTE

Before you use the GO TO statement, be sure you know how to use the CTRL/C key command to stop your program from running in an infinite loop. See Section 9.1 for information on CTRL/C.

4.1.2 Multiple Branching (ON GO TO and ON THEN Statements)

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

The ON GO TO statement has the following format:

```
ON expression {GO TO} line number1 [[,line number2,...]]
              {THEN }
```

where the expression is any legal BASIC numeric expression. The keywords GO TO and THEN are interchangeable. Line numbers must be separated by commas.

The ON GO TO statement is also known as a computed GO TO because of its dependency on the value of the numeric expression. When BASIC executes the ON GO TO 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. If the value is less than 1 or greater than the number of line numbers in the list, BASIC prints the ?CONTROL VARIABLE OUT OF RANGE (?CVO) error message.

Consider this example:

```
200 ON A GO TO 50,20,100,300
```

If A=1, go to line 50 (first line number in the list).
If A=2, go to line 20 (second line number in the list).
If A=3, go to line 100 (third line number in the list).
If A=4, go to line 300 (fourth line number in the list).
If A<1 or If A>4 } BASIC prints an error message.

As you can see, the line numbers in the list can be in any order.

4.1.3 Conditional Transfer (IF THEN and IF GO TO Statements)

The IF THEN statement provides a transfer of control depending on the truth of a relational expression (see Section 2.4.3).

The format of the IF THEN statement is:

```
IF relational expression {GO TO line number}
                         {THEN line number }
                         {THEN statement   }
```

where:

relational expression	is the condition to be tested. It can either be an arithmetic or string relational expression.
line number	specifies the line to be executed if the condition is true.
statement	is executed if the condition is true. The statement can be any BASIC statement including another IF THEN statement.

If you specify a line number and the value of the relational expression is true, then control is transferred to the specified line number. For example:

```
20 IF A=3 THEN 200
```

When A is equal to 3 (the relation is true), control passes to line 200. The implication is that when A is not equal to 3, control does not pass to line 200. Instead, control passes to the next sequential statement after line 20.

You can also use string expressions as in this example:

```
300 IF C$="OUTPUT" GO TO 10
```

If the value of the string variable C$ is equal to the ASCII value of "OUTPUT", control passes to line 10. See Section 2.4.3 for string relational expressions.

If you specify a statement after THEN and the value of the relational expression is true, the statement is executed. If the value of the relational expression is false, the statement is not executed and control is passed to the next program line. For example:

```
10 IF A=1 THEN PRINT "A=1"
```

Here is a complete program illustrating the IF THEN statement:

```
LISTNH
5 REM PROGRAM TO COMPARE TWO NUMBERS
10 PRINT "INPUT VALUE OF A"; \ INPUT A
20 PRINT "INPUT VALUE OF B"; \ INPUT B
30 IF A=B THEN PRINT "A EQUALS B" \ GO TO 80
40 IF A<B THEN 60
50 PRINT "B IS LESS THAN A" \ GO TO 80
60 PRINT " A IS LESS THAN B"
80 END

READY
RUNNH

INPUT VALUE OF A? 2
INPUT VALUE OF B? -3
B IS LESS THAN A

READY
```

Care should be taken placing the IF THEN statement in a multi-statement line. The following rules govern the transfer of control:

1. If THEN is followed by a line number:

   If the THEN clause contains a line number and the condition is true, control passes to that line number. If, however, the condition is false, control passes to the statement following the THEN clause. For example:

```
LISTNH
10 A=5
20 IF A=2 THEN 30 \ PRINT A \ GO TO 40
30 PRINT "THE CONDITION IS TRUE."
40 END

READY
RUNNH

 5

READY
```

   Because the condition is false, BASIC executes the statement following the THEN clause. If you interpret the backslash, in this case, to mean "otherwise", you can see the alternatives:

   > If A is equal to 2, transfer control to line 30; otherwise, print the value of A and then transfer control to line 40.

2. If THEN is followed by a statement:

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

```
5 INPUT A
10 IF A=1 THEN PRINT A; \ PRINT "TRUE CASE" \ GO TO 20
15 PRINT "NOT=1"
20 END
```

   If A is equal to 1, BASIC prints:

```
RUNNH

? 1

 1 TRUE CASE

READY
```

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

If A is not equal to 1, BASIC prints:

```
RUNNH

? 5
NOT=1

READY
```

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.

All other THEN clauses are considered to be followed by the next line of the program:

```
10 INPUT A,B,C
20 IF A>B THEN IF B<C THEN PRINT "B<C" \ GO TO 30
25 PRINT "A<=B OR B>=C"
30 END
```

The statement GO TO 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 OR B>=C

READY
```

4.2 EXECUTION OF LOOPS

At some point, you may find that you are typing the same statements many times in a program. Instead of typing them over and over again, make BASIC execute them over and over again. You can accomplish this by building a loop in your program.

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.

For example, consider the following two programs to print the numbers from 1 to 10.

Program Without Loop

```
LISTNH
10 PRINT 1
20 PRINT 2
30 PRINT 3
40 PRINT 4
50 PRINT 5
60 PRINT 6
70 PRINT 7
80 PRINT 8
90 PRINT 9
100 PRINT 10
110 END

READY
RUNNH

 1
 2
 3
 4
 5
 6
 7
 8
 9
 10

READY
```

Program With Loop

```
LISTNH
10 I%=1%
20 PRINT I%
30 I%=I%+1%
40 IF I%<=10% THEN 20
50 END

READY
RUNNH

 1
 2
 3
 4
 5
 6
 7
 8
 9
 10

READY
```

The program on the right first initializes a control variable, I%, in line 10. It then executes the body of the loop, line 20. Finally, it increments the control variable in line 30 and compares it to a final value in line 50.

The following section shows you how to build a loop with the FOR and NEXT statements.

4.2.1 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 set up a loop wherein BASIC tests for a condition automatically each time it runs through the loop. You decide how many times you want the loop to run, and you set the terminating condition.

The FOR statement has the following format:

FOR variable = expr1 TO expr2 [STEP expr3]

where:

variable is a simple numeric variable known as the loop index.

expr1 is the initial value of the index and can be any numeric expression.

expr2 is the terminating condition and can be any numeric expression.

expr3 is the incremental value of the index. The STEP size is optional; if specified, it can be positive or negative. If not specified, the default is +1. Expr3 can be any numeric expression.

The NEXT statement has the following format:

NEXT variable

where:

variable must be the same variable named in the corresponding FOR statement.

For example:

```
20 FOR M = 30 TO 90 STEP 3
50 NEXT M
```

M 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 90. When BASIC reaches the NEXT statement and M has a value of 87, BASIC adds 3 to M and tests the result against the terminating value. The result, 90, is not greater than the terminating value, also 90, so BASIC executes the loop again. When BASIC reaches the NEXT statement again, it adds 3 to M, producing 93. Because this is greater than the terminating value, BASIC terminates the loop. BASIC terminates the loop by subtracting 3 from M, which returns M to its last value used in the loop, 90, and then by transferring control to the next sequential statement after the NEXT statement.

The FOR and NEXT statements must be used together. You cannot use one without the other. If you do, an error condition results. The FOR statement defines the beginning of the loop; the NEXT statement defines the end. You are actuallly building a counter in your program to determine the number of times the loop is to execute.

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

```
LISTNH
10 FOR I% = 1% TO 10%
20 PRINT I%,
30 NEXT I%
40 PRINT I%
50 END

READY
RUNNH

 1
 2
 3
 4
 5
 6
 7
 8
 9
 10
 10

READY
```

In this program, the initial value of the index variable 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.

Notice that when control passes from the loop, the last value of the loop variable is retained. Although BASIC increments the control variable until it is greater than the terminating value, BASIC subtracts the STEP value to return the control variable to the value last used in the body of the loop. Therefore, I% equals 10 on line 40.

You can modify the index variable within the loop.

```
10 FOR I = 2 to 44 STEP 2
20 LET I = 44
30 NEXT I
40 END
```

The loop in this program only executes 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 index variable is greater than the terminal value, the loop is never executed.

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

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

```
10 FOR I = 20 to 2 STEP -2
```

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

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

NOTE

You cannot transfer control into a loop that has not been initialized with a FOR statement. The following is illegal in a BASIC program:

```
10 REM THIS IS ILLEGAL
20 GO TO 40
30 FOR I=1 TO 20
40 PRINT I
50 NEXT I
60 END
```

Line 20 shifts control to line 40, bypassing line 30. This is illegal in BASIC.

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

```
LISTNH
10 FOR I=1 TO 10 STEP 5 \ NEXT I \ PRINT "I =";I
20 END

READY
RUNNH

I =  6

READY
```

The calculation of the index values (initial, final, and step size) is subject to precision limitations inherent in 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:

```
LISTNH
20 FOR X=0 TO 10 STEP .1
30 T=X+T
40 NEXT X
50 PRINT "X=";X,"T=";T

READY
RUNNH

X= 9.9          T= 495

READY
```

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 and X is incremented, X is not exactly equal to 10. It is slightly larger than 10, so the loop stops. Whenever possible, it is advisable to use indices that have integer values because the loop will then be executed the correct number of times.

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

```
10 I=10
20 FOR I=1 TO I*2
30 NEXT I
```

BASIC evaluates I*2 in line 20 and calculates a value of 20 before it assigns a value of 1 to I. The previous example is equivalent to

```
20 FOR I=1 TO 10*2
30 NEXT I
```

4.2.2 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, i.e., the one that starts first must be completed last. Loops cannot overlap.

The following example shows legal and illegal forms of nested loops:

LEGAL

```
10 FOR A%=1% TO 10%
20 FOR B=2 TO 20
30 NEXT B
40 NEXT A%
```

LEGAL

```
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%
```

ILLEGAL

```
10 FOR M=1 TO 10
20 FOR N=2 TO 20
30 NEXT M
40 NEXT N
```

The following is a program with a nested loop:

```
LISTNH
10 PRINT "I","J"
15 PRINT
20 FOR I%=1 TO 2
30 FOR J%=1 TO 3
40 PRINT I%,J%
50 NEXT J%
60 NEXT I%
70 END

READY
RUNNH

 I       J

 1        1
 1        2
 1        3
 2        1
 2        2
 2        3

READY
```

INSIDE LOOP

OUTSIDE LOOP

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

```
LISTNH
5 DIM X(5,10)
10 FOR A=1 TO 5
20 FOR B=2 TO 10 STEP 2
30 X(A,B)=A+B
40 NEXT B
50 NEXT A
55 PRINT X(5,10)
60 END

READY
RUNNH

 15

READY
```

4.3 STOPPING PROGRAM EXECUTION (END AND STOP STATEMENTS)

There are three methods of halting program execution:

1. Using the END statement
2. Executing the program line with the highest line number (the end of the program)
3. Using the STOP statement

The END statement has the following format:

```
END
```

The END statement is optional. If you include an END, it must have the largest line number in the program. Transferring control to an END statement via a GO TO or IF THEN statement terminates program execution and closes all files (see Chapter 6 for a description of data 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 execute a STOP or an END statement in a program, executing the last statement of the program terminates program execution and closes all files. This is equivalent to executing an END statement.

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
```

where n is the line number of the STOP statement.

You can place several STOP statements at various points in a single program. The flow of logic can then be seen throughout the program. This is a useful debugging tool in determining program flow in large programs.

After execution of a STOP statement, you can print variables, change values of variables, and then continue execution with an immediate mode GO TO statement. (See Section 1.8 for a discussion of immediate mode commands.)

The STOP statement halts execution but it does not close files. To cause BASIC to close files after program termination, use the END statement or no terminating statement.

4.4 SUBROUTINES

A subroutine is a block of statements that performs an operation and then returns control of the program to the point from which it came. Including a subroutine in a program allows you to repeat a procedure in several places without writing the procedure several times.

Subroutines are like functions (Section 2.4.4) in that you reference them in another part of the program. However, unlike functions, you do not name a subroutine or specify an argument. Instead, you include the GOSUB and RETURN statements which transfer control of the program to a subroutine and then return control from that subroutine to the normal course of 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). 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 legal BASIC statement including a REM statement. Note that you do not have to transfer to the first line of the subroutine. Instead, you can include several entry points and returns in and out of the same subroutine. Similarly, you can nest subroutines (one subroutine within another) up to 20 levels.

The following sections describe the building of subroutines with the GOSUB and RETURN statements. For more flexibility in using subroutines, see Section 4.4.2, the ON GOSUB statement.

4.4.1 GOSUB and RETURN Statements

The GOSUB statement has the following format:

```
GOSUB line number
```

where:

line number specifies the entry point in the subroutine.

When BASIC executes the GOSUB statement, it stores the location of the next sequential statement after the GOSUB statement and then it transfers control to the line specified.

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 exit only through a RETURN statement.)

The RETURN statement has the following format:

```
RETURN
```

BASIC has a table where it can store up to 20 locations of statements following a GOSUB. Each time a GOSUB is executed, BASIC stores another location on the list. Each time a RETURN is executed, BASIC retrieves the last location entered on the list and transfers control to it. 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:

```
LISTNH
10 INPUT A,B,C
15 IF A=-9999 GO TO 70
20 GOSUB 40
30 PRINT D
35 GO TO 10
40 REM --THIS IS A SUBROUTINE
50 D=A*B-C
60 RETURN
70 END

READY
RUNNH

? 5,10,15
 35
? -9999,0,

READY
```

When BASIC executes line 20, it stores the location of the next statement, PRINT D. Then it transfers control to line 40. When it reaches the RETURN statement at line 60, it transfers control to the statement after the GOSUB (e.g., the PRINT D statement).

The following is an example of several calls to the same subroutine:

```
10 A=0
20 GOSUB 60 \ A=5
30 GOSUB 60 \ A=6
40 GOSUB 60
50 GO TO 110
70 FOR I=1 TO 5
80 LET A=A+B(I)
90 NEXT I
95 PRINT A
100 RETURN
110 END
```

The same subroutine on line 60 is called three times. Notice that only one RETURN statement is necessary.

The following program is another illustration of the GOSUB and RETURN statements:

```
LISTNH
10 REM -- THIS PROGRAM ILLUSTRATES GOSUB AND RETURN
20 DEF FNA(X)=ABS(INT(X))
30 PRINT "THIS SUBROUTINE USES THE QUADRATIC FORMULA."
40 PRINT "ENTER THREE COEFFICIENTS"$
50 INPUT A,B,C
55 PRINT
60 PRINT "SOLUTIONS FOR COEFFICIENTS ENTERED ARE"
70 PRINT "SHOWN FIRST, THEN SOLUTIONS FOR ABSOLUTE"
80 PRINT " VALUE COEFFICIENTS."
85 PRINT
90 GOSUB 160
100 LET A=FNA(A)
110 LET B=FNA(B)
120 LET C=FNA(C)
130 PRINT
140 GOSUB 160
145 PRINT
150 GO TO 310
160 REM--THIS SUBROUTINE PRINTS OUT THE SOLUTIONS
170 REM--OF THE EQUATION A*X^2+B*X+C=0
180 PRINT "THE EQUATION IS "$A$"*X^2 +"$B$" *X + "$C
190 LET D=B*B-4*A*C
200 IF D<>0 THEN 230
210 PRINT "ONLY ONE SOLUTION...X="$-B/(2*A)
220 RETURN
230 IF D<0 THEN 270
240 PRINT "TWO SOLUTIONS...X=("$
250 PRINT (-B+SQR(D))/(2*A)$") AND ("$(-B-SQR(D))/(2*A)$")"
260 RETURN
270 PRINT "IMAGINARY SOLUTION...X=("$
280 PRINT -B/(2*A)$"+"$SQR(-D)/(2*A)$"I)"
285 PRINT "      AND ("$
290 PRINT -B/(2*A)$"-"$SQR(-D)/(2*A)$"*I)"$
300 RETURN
310 END

READY
RUNNH

THIS SUBROUTINE USES THE QUADRATIC FORMULA.
ENTER THREE COEFFICIENTS? 3,-5,-6

SOLUTIONS FOR COEFFICIENTS ENTERED ARE
SHOWN FIRST, THEN SOLUTIONS FOR ABSOLUTE
 VALUE COEFFICIENTS.

THE EQUATION IS 3 *X^2 + -5 *X + -6
TWO SOLUTIONS...X=( 2.47481 ) AND (-.808143 )

THE EQUATION IS 3 *X^2 +  5 *X +  6
IMAGINARY SOLUTION...X=(-.833333 + 1.14261 I)
     AND (-.833333 - 1.14261 *I)

READY
```

For more versatility in using subroutines, see the next section, the ON GOSUB statement.

4.4.2 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 into one or more subroutines. The ON GOSUB statement has the following format:

ON expression GOSUB line number1[,line number2,...]

where the expression is any legal BASIC expression. Line numbers must be separated by commas.

The ON GOSUB statement works like the ON GO TO statement (Section 4.1.2). 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; if it is 2, control passes to the second line number specified; if it is 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 the ?CONTROL VARIABLE OUT OF RANGE (?CVO) error message. The following is an example of an ON GOSUB statement:

```
20 ON A+B GOSUB 200,300,120
```

If A+B=1, enter the subroutine at line 200 (first line number in list).

If A+B=2, enter the subroutine at line 300 (second line number in list).

If A+B=3, enter the subroutine at line 400 (third line number in list).

If A+B<1
or
If A+B>4
} BASIC prints an error message.

The line numbers to which BASIC branches can be either the first line of a subroutine or an entry point to a subroutine.

CHAPTER 5

FUNCTIONS

5.1 TYPES OF FUNCTIONS AVAILABLE

Functions perform a series of numeric or string operations on the arguments you specify and return a result to BASIC (see Section 2.4.5). BASIC provides mathematical functions, string functions, functions that you can define, and functions that give you the current day and time.

5.2 NUMERIC FUNCTIONS

BASIC provides numeric functions to perform standard mathematical operations. For example, it is often necessary to find the sine of an angle. You can do this by looking it up in a table of sine values or by using BASIC's SIN function. If you are not familiar with the mathematical functions described in this chapter, see a trigonometry or algebra textbook.

BASIC provides the following trigonometric functions:

1. Sine function (SIN)
2. Cosine function (COS)
3. Arctangent function (ARC)

In addition BASIC provides a special function, PI, which returns the value of pi (π), a frequently used trigonometric constant.

BASIC provides algebraic functions to find:

1. The square root of a number (SQR)
2. The value of e, an algebraic constant, raised to any power (EXP)
3. The logarithm of a number (LOG and LOG10)
4. The integral part of a number (INT)
5. The absolute value of a number (ABS)
6. The sign of a number (SGN)

BASIC also provides a function RND which returns a random number. You can use this function when you are trying to simulate an unpredictable situation with a BASIC program.

All BASIC's numeric functions return floating point values, not integer values. It is important that these functions return values in floating point format to determine whether an expression is integer or floating point (see Section 2.4.1). All functions have a 6-digit accuracy.

5.2.1 Trigonometric Functions (SIN, COS, ATN, and PI Functions)

BASIC provides functions, SIN and COS, to find the sine and cosine of an angle. In addition you can use the ATN function to find the arctangent of a number, the angle whose tangent is equal to the number. BASIC also provides the PI function, which returns a value of 3.14159, an approximation of pi. Pi is a transcendental number and can only be approximated in a decimal fraction. The format of these functions are:

```
SIN(expression)
COS(expression)
ATN(expression)
PI
```

Note that the PI function cannot have an argument. If you specify an argument with PI, BASIC prints the ?SYNTAX ERROR (?SYN) message.

Although BASIC does not have a tangent function, you can find the tangent of a number by using the following trigonometric equation:

$$\text{tangent (angle)} = \frac{\text{sine (angle)}}{\text{cosine (angle)}}$$

BASIC requires that the arguments for the SIN and COS function be expressed as angles in radian measure. It also returns the value of the arctangent in radians in the range -pi/2 to pi/2 radians. (There are 2 *pi radians in a full circle.) If you want to measure angles in degrees (360 degrees in a full circle), you can use the following equation:

$$\text{radians} = \frac{\text{degrees} * pi}{180}$$

Consider the following program which converts an angle in degrees to an angle in radians and then calculates and prints the sine, cosine, and tangent of the angle. Finally the program prints the arctangent of the value returned by the tangent function. The arctangent should be the same as the original angle.

```
LISTNH
10 PRINT "SUPPLY AN ANGLE IN DEGREES"
20 PRINT "ENTER -9999 TO END"
25 PRINT
30 PRINT "ANGLE","SIN","COS","TAN","ATN"
40 PRINT "IN RADIANS",,,,"IN RADIANS"
45 PRINT
50 PRINT "ANGLE IN DEGREES";
60 INPUT X
65 IF X=-9999 THEN 32767
70 R=X*PI/180
80 IF ABS(COS(R))<.01 THEN 200
90 T=SIN(R)/COS(R)
100 PRINT R, SIN(R), COS(R), T, ATN(T)
110 GO TO 50
200 PRINT "COSINE IS TOO CLOSE TO ZERO"
210 GO TO 50
32767 END

READY
RUNNH

SUPPLY AN ANGLE IN DEGREES
ENTER -9999 TO END

ANGLE          SIN            COS            TAN            ATN
IN RADIANS                                                 IN RADIANS

ANGLE IN DEGREES? 0
 0              0              1              0              0
ANGLE IN DEGREES? 45
 .785398        .707107        .707107        1              .785398
ANGLE IN DEGREES? 10
 .174533        .173648        .984808        .176327        .174533
ANGLE IN DEGREES? 89.99
COSINE IS TOO CLOSE TO ZERO
ANGLE IN DEGREES? -9999

READY
```

At lines 50 and 60, BASIC requests that you enter an angle in degrees. At line 65 BASIC checks to see if you have entered -9999 to end the program. If you have not entered -9999, BASIC converts the angles to radians, at line 70. At line 80 BASIC checks to see if the cosine of the angle is close to 0 (this happens when the angle is close to pi/2 or 90 degrees). If the cosine is close to 0, calculating the tangent would involve division by 0, which would produce the ?DIVISION BY ZERO (?DV0) error message. Next BASIC calculates the tangent of the angle, at line 90. Finally, BASIC prints the angle in radians, its sine, cosine and tangent, and the arctangent of the tangent (which should equal the angle).

NOTE

When using this program, you type -9999 to terminate it. -9999 was chosen because it is unlikely to be entered as a value in this context. Throughout this chapter many examples include this method of program termination.

5.2.2 Algebraic Functions

BASIC provides the following algebraic functions:

Square root function (SQR)
Exponential function (EXP)
Logarithm function (LOG and LOG10)
Integer function (INT)
Absolute Value function (ABS)
Sign function (SGN)

5.2.2.1 **Square Root Function (SQR Function)** - The SQR function returns the square root of the expression you specify. A square root of a number times itself (squared) equals the original number. The format of the SQR function is:

SQR(expression)

You use this function in BASIC instead of the mathematical notation for square root ($\sqrt{\ }$).

If the value of the expression is negative, BASIC prints the nonfatal message ?NEGATIVE SQUARE ROOT (?NGS) and returns a value of 0. For example:

```
LISTNH
10 PRINT "THE SQUARE", "IS"
20 PRINT "ROOT OF"
30 PRINT 16,SQR(16)
40 PRINT -100,SQR(-100)
50 X=31.6228*31.6228
60 PRINT X,SQR(X)

READY
RUNNH

THE SQUARE      IS
ROOT OF
 16              4
-100            ?NEGATIVE SQUARE ROOT AT LINE 40
 0
 1000            31.6228

READY
```

Notice that when BASIC tries to calculate the square root of -100 it prints the error message and then prints 0 on the next line.

5.2.2.2 **Exponential and Logarithm Functions (EXP, LOG, and LOG10 Functions)** - The exponential function returns e, an algebraic constant raised to the power specified. For example EXP(1) is equal to e , approximately 2.71828, and EXP(2) is equal to e^2. The format of the EXP function is:

EXP(expression)

The LOG function returns the logarithm to the base e of the specified 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.

```
LOG(EXP(X)) = X
```

Consider the following examples. Note that the output from one example is used as input for the other.

EXP Function

```
LISTNH
10 INPUT X
15 IF X = -9999 THEN 100
20 PRINT EXP(X)
30 GO TO 10
100 END

READY
RUNNH

? 4
 54.5981
? 10
 22026.5
? 9.42100
 12344.9
? -9999

READY
```

LOG Function

```
LISTNH
10 INPUT X
15 IF X = -9999 THEN 100
20 PRINT LOG(X)
30 GO TO 10
100 END

READY
RUNNH

? 54.59815
 4
? 22026.47
 10
? 12345
 9.42101
? -9999

READY
```

You can convert logarithms to the base e to any other base log by using the following formula:

$$\log_a(N) = \frac{\log_e(N)}{\log_e(a)}$$

where you are trying to find the log of N to the base a. Consider the following examples which calculate logs to any base.

```
LISTNH
10 REM - CONVERT BASE E LOG TO ANY BASE LOG.
20 PRINT "WHAT BASE";
30 INPUT B
40 PRINT "VALUE","BASE E LOG","BASE";B;"LOG"
50 INPUT X
60 IF X = -9999 THEN 200
70 PRINT X,
80 PRINT LOG(X),
90 PRINT LOG(X)/LOG(B)
100 GO TO 30
200 END

READY
RUNNH

WHAT BASE? 2
VALUE          BASE E LOG     BASE 2 LOG
? 4
 4              1.38629        2
? 250
 250            5.52146        7.96576
? 5
 5              1.60944        2.32193
? -9999

READY
```

However, you need not use this formula to find the logarithm to the base 10 of a number. BASIC provides the LOG10 function which does this. The form of the LOG10 function is:

```
LOG10(expression)
```

You use the LOG10 the same way that you use the LOG function. The only difference is that LOG10 returns the logarithm to the base 10 instead of to the base e.

If you specify an expression in the LOG or LOG10 function whose value is negative or equal to 0, BASIC prints the nonfatal message ?BAD LOG (?BLG) and the function returns a value of 0.

5.2.2.3 **Integer Function (INT Function)** - 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:

```
LISTNH
10 PRINT INT(34.47)
20 PRINT INT(33000.9)

READY
RUNNH

 34
 33000

READY
```

The INT function always returns the value of the greatest integer that is less than or equal to the specified expression. A consequence of this is that when you specify a negative number, INT produces a number whose absolute value is larger. For example:

```
LISTNH
10 PRINT INT(-23.45)
20 PRINT INT(-14.7)
30 PRINT INT(-11)

READY
RUNNH

-24
-15
-11

READY
```

Note that the value returned by INT is a whole number in floating point format, not an integer.

You can use the INT function to round off numbers to the nearest integer by adding 0.5 to the argument. For example:

```
LISTNH
10 PRINT INT(34.67+.5)
20 PRINT INT(-5.14+.5)

READY
RUNNH

 35
-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:

number rounded off = INT(number*10^P+.5)/10^P

where P represents the number of places of accuracy.

Consider the following example which rounds numbers to the number of decimal places specified by the user. Note that P should have a positive value (tested in line 135) in the formula (which is in line 150) for P to determine the digits of accuracy to the right of the decimal point.

```
LISTNH
50 REM PROGRAM TO ROUND OFF DECIMAL NUMBERS
100 PRINT "WHAT NUMBER DO YOU WISH TO ROUND OFF";
110 INPUT N
115 IF N = -9999 THEN 100
120 PRINT "TO HOW MANY PLACES";
130 INPUT P
140 PRINT
150 LET A=INT(N*10^P+.5)/(10^P)                    Round the number
160 PRINT N"="A"TO"P"DECIMAL PLACES."
170 PRINT
180 GO TO 100
1000 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
```

And consider this program which prints numbers to the nearest thousandth (10^3). Note that in this program, P has a constant value of -3 in the formula (on line 40). Because P is negative, the accuracy is three digits to the left of the decimal point, that is the numbers are rounded to the nearest thousand.

```
LISTNH
10 REM PROGRAM TO ROUND TO NEAREST 1000
20 PRINT "WHAT NUMBER DO YOU WANT TO ROUND?
30 INPUT N
35 IF N=-9999 THEN 1000
40 A=INT(N*10^(-3)+.5)/10^(-3)
50 PRINT N;"=";A;"TO THE NEAREST THOUSAND"
55 PRINT
60 GO TO 20
1000 END

READY
RUNNH

WHAT NUMBER DO YOU WANT TO ROUND? 34339
 34339 = 34000 TO THE NEAREST THOUSAND

WHAT NUMBER DO YOU WANT TO ROUND? 11749
 11749 = 12000 TO THE NEAREST THOUSAND

WHAT NUMBER DO YOU WANT TO ROUND? -9999

READY
```

5.2.2.4 **Absolute Value Function (ABS Function)** - The ABS function returns the absolute value of the specified expression. The form of the ABS function is:

```
ABS(expression)
```

The absolute value of a number, by mathematical definition, is always positive. The absolute value of a positive number is equal to the number, but the absolute value of a negative number is equal to -1 times the number. For example:

```
LISTNH
10 INPUT X
15 IF X=-9999 THEN 100
20 X=ABS(X)
30 PRINT X
40 GO TO 10
100 END

READY
RUNNH

? -35.7
 35.7
? 2
 2
? 25E20
 2.50000E+21
? -10555567
  .05556E+07
 -9999
```

Note that the ABS function returns a floating point number even if the argument is an integer. In that case, it returns a whole number in floating point format.

5.2.2.5 **Sign Function (SGN Function)** - You can use the sign function to determine if an expression is positive, negative, or equal to 0. The format of the SGN function is:

SGN(expression)

If the expression you specify is positive, SGN returns a +1; if it is negative, SGN returns a -1; and if it is equal to 0, SGN returns a 0. For example:

```
LISTNH
10 A=-7.32
20 B=.44
30 C=0
40 PRINT "A=";A,"B=";B,"C=";C
50 PRINT "SGN(A)=";SGN(A),
60 PRINT "SGN(B)=";SGN(B),
70 PRINT "SGN(C)=";SGN(C)

READY
RUNNH

A=-7.32         B= .44          C= 0
SGN(A)=-1       SGN(B)= 1       SGN(C)= 0

READY
```

Note that SGN returns these values as real numbers, not integers.

5.2.3 **Random Numbers (RND Function and RANDOMIZE Statement)**

It is often useful to get a series of random numbers in a BASIC program. A series of random numbers is a series of numbers which are not related to each other in any way. You can use random numbers in simulating a situation in the world which is not predictable.

It is impossible for a computer to produce a series of totally random numbers because computers always produce the same results given the same starting comditions. Instead, BASIC uses complex calculations to produce a predictable series of numbers that seem unrelated. This is called a pseudo-random series.

The RND function returns a number from this pseudo-random series each time you use it. It starts from the beginning of the series whenever you initialize your program (OLD, NEW, SCR, or RUN command or CHAIN statement).

In addition, BASIC provides the RANDOMIZE statement. Every time that BASIC executes the RANDOMIZE statement it starts the RND function at a new, unpredictable (based on the current time of day) location in the series. When you are testing and changing your program you should not use the RANDOMIZE statement. If you do, you will not know if changes in the results are caused by changes in the program or the series starting off in a new location. If, on the other hand, you have tested your program and are using it, use the RANDOMIZE statement because it produces less predictable series.

The format of the RND function is:

```
RND
```

The function returns a random number in the open range 0 to 1. (An open range means the extremes, 0 and 1 in this case, are never reached.)

You can also specify an expression in RND, for example, RND(0), but do not do this. BASIC ignores the expression and only allows it because you may have written a program with another version of BASIC which requires an expression after RND.

The format of the RANDOMIZE statement is:

```
RANDOMIZE
```

Consider the following examples which contrast RND without and with RANDOMIZE.

RND without RANDOMIZE

```
LISTNH
10 PRINT RND,RND,RND,RND

READY
RUNNH

 .0407319        .528293         .803172         .0643915

READY
RUNNH

 .0407319        .528293         .803172         .0643915

READY
RUNNH

 .0407319        .528293         .803172         .0643915

READY
```

As you can see, every time the program without RANDOMIZE is run, RND produces the same series of values.

RND with RANDOMIZE

```
LISTNH
5 RANDOMIZE
10 PRINT RND,RND,RND,RND

READY
RUNNH

 4.53806E-03     .419712         .477427         .0871577

READY
RUNNH

 .258322         .181064         .761485         .93933

READY
RUNNH

 .494162         .888584         .884043         .307006

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) * RND+A
```

For example, to produce 10 numbers in the open range 4 to 6, use this program

```
LISTNH
10 FOR I=1 TO 10
20 PRINT (6-4)*RND+4,
30 NEXT I

READY
RUNNH

 4.08146        5.05659        5.60634        4.12878        4.31561
 4.73461        5.56717        4.79154        4.6447         4.74433

READY
```

Note that in line 20 of the program the general expression is used with a value of 4 for B and a value of 6 for A.

To obtain a series of random integers in the range of 0 to 40, use the expression

```
INT((41-0)*RND+0)
```

or the equivalent

```
INT(41*RND)
```

41 is used instead of 40 so that the expression can return a value of 40.

The following example produces 15 random integer values in the range 0 to 40.

```
LISTNH
10 FOR I = 1 TO 15
20 PRINT INT(41*RND),
30 NEXT I

READY
RUNNH

 1              21             32             2              6
 15             32             16             13             15
 13             26             35             16             26

READY
```

5.3 STRING FUNCTIONS

BASIC provides string functions that allow you to examine and modify strings and perform certain string to numeric conversions.

String functions that return a string have a dollar sign ($) at the end of their name. String functions that return floating point or integer numbers do not have a dollar sign.

5.3.1 String Manipulation Functions

BASIC's string operations allow you to concatenate and to compare strings, but only by also using the string manipulation functions can you analyze what a string is composed of. You can use string functions to:

1. Determine the length of a string (LEN).
2. Trim off trailing blanks from a string (TRM$).
3. Search for the position of a set of characters within a string (POS).
4. Copy a segment from a string (SEG$).

5.3.1.1 **Finding the Length of a String (LEN Function)** - You can use the LEN function to find the length, or number of characters, of a string. The LEN function returns an integer value equal to the length of the string you specify. The format of the LEN function is:

```
LEN(string)
```

Consider the following example which prints the length of a string containing all the letters in the alphabet:

```
LISTNH
10 A$="ABCDEFGHIJKLMNOPQRSTUVWXYZ"
20 PRINT LEN(A$)

READY
RUNNH

 26

READY
```

5.3.1.2 **Trimming Trailing Blanks Off a String (TRM$ Function)** - The TRM$ function returns the same string that you specify except that any trailing blanks are 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:

```
LISTNH
10 A$="ABCD   "
20 B$="EFG"
30 PRINT "BEFORE TRIMMING:",A$+B$
40 PRINT "AFTER TRIMMING:",TRM$(A$)+B$

READY
RUNNH

BEFORE TRIMMING:          ABCD   EFG
AFTER TRIMMING:           ABCDEFG

READY
```

5.3.1.3 **Finding the Position of a Substring (POS Function)** - Use the POS function to find the location of a group of characters, or substring, within a string. The format of the POS function is:

```
POS(string1,string2,expression)
```

where:

string1 is the string being searched.

string2 is the substring.

expression is the character position at which BASIC starts the search.

The POS function searches for and returns the first occurrence of string2 in string1 starting with the character position specified by expression. If POS finds the specified substring, it returns the character position of the first character of the substring. If POS does not find the specified substring, it returns a 0.

The POS function always returns an integer value.

Consider the following example which translates each name of a month to its numeric equivalent (e.g., DEC to 12):

```
LISTNH
10 T$="JANFEBMARAPRMAYJUNJULAUGSEPOCTNOVDEC"
100 PRINT "TYPE THE FIRST 3 LETTERS OF A MONTH";
110 INPUT M$
120 IF M$="END" THEN 32767
130 IF LEN(M$)<>3 THEN 200
140 M=(POS(T$,M$,1)+2)/3
150 IF M<>INT(M) THEN 200
160 PRINT M$;" IS MONTH NUMBER";M
170 GO TO 100
200 PRINT "INVALID ENTRY - TRY AGAIN" \GO TO 100
32767 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? MAY
MAY IS MONTH NUMBER 5
TYPE THE FIRST 3 LETTERS OF A MONTH? JUD
INVALID ENTRY - TRY AGAIN
TYPE THE FIRST 3 LETTERS OF A MONTH? END

READY
```

In line 140 the POS function returns the position of the input string, M$, in the string containing the first three letters of each month, T$.

If the program finds the month you specify, it prints the number of the month. If it does not find the month, it requests you to try again.

Using the POS function to translate a string to a number corresponding to the string's location in a larger string is called a table look-up. The table string is string1 and the string to be mapped is string2 in the POS function format.

There are certain possible error conditions dependent on the values of the strings and the expression:

1. If string2 (the substring) is null and string1 (the table string) is nonnull then the function returns the lesser of the value of the expression and the length of string1 plus 1.

2. If string1 is null, then the function returns 0.

3. If the expression has a value less than 1, then the function assumes a value of 1 and starts the search at the first character.

4. If the value of the expression is greater than the length of string1 and string2 is nonnull, then the function returns 0.

5.3.1.4 **Copying Segments from a String (SEG$ Function)** - Use the SEG$ function to copy a segment (or substring) from a string. The SEG$ function returns a string consisting of the characters in the string you specify between the character positions you also specify. The original string is unchanged. The format of the SEG$ function is:

SEG$(string,expression1,expression2)

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:

```
LISTNH
10 PRINT SEG$("ABCDEF",3,5)

READY
RUNNH

CDE

READY
```

There are several error conditions based on the values of the expressions and the string:

1. If expression1 is less than 1, BASIC assumes a value of 1.
2. If expression1 is greater than expression2 or the length of string, SEG$ returns a null string.
3. If expression2 is greater than the length of the string, SEG$ returns the characters from expression1 to the end of the string.
4. If expression1 equals expression2, then SEG$ returns the character at position expression1.

By using the SEG$ function and the string concatenation operator (+), you can replace a segment of a string. Consider the following example:

```
LISTNH
10 A$="ABCDEFG"
20 C$=SEG$(A$,1,2)+"XYZ"+SEG$(A$,6,7)
30 PRINT C$

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)
```

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$=SEG$(A$,1,n-1)+B$+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:

```
LISTNH
10 A$="ABCDEFGHIJK"
20 B$="123456"
30 C$=SEG$(A$,1,5)+B$+SEG$(A$,10,LEN(A$))
40 PRINT C$

READY
RUNNH

ABCDE123456JK

READY
```

The following formulas are more specific applications of the general formula.

To replace the first n characters of A$ with B$:

```
C$=B$+SEG$(A$,n+1,LEN(A$))
```

To replace all but the first n characters of A$ with B$:

```
C$=SEG$(A$,1,n)+B$
```

To replace all but the last n characters of A$ with B$:

```
C$=B$+SEG$(A$,LEN(A$)-n+1,LEN(A$))
```

To replace the last n characters of A$ with B$

```
C$=SEG$(A$,1,LEN(A$)-n)+B$
```

To insert B$ in A$ after the nth character in A$:

```
C$=SEG$(A$,1,n)+B$+SEG$(A$,n+1,LEN(A$))
```

5.3.2 Conversion Functions

BASIC provides several string functions to convert strings to numbers and numbers to strings.

You can use BASIC's functions to make the following conversions:

1. Character to ASCII code (ASC)
2. ASCII code to character (CHR$)
3. Number to its string representation (NUM$)
4. String representation of a number to a number (VAL)
5. String representing a binary number to a decimal number (BIN)
6. String representing an octal number to a decimal number (OCT)

These functions provide flexibility in manipulating both strings and numbers.

5.3.2.1 **Character and ASCII Code Conversions (ASC - and CHR$ Functions)** - BASIC uses the ASCII code to represent characters internally. The ASC function returns the decimal ASCII code of a 1-character string that you specify. The CHR$ function returns the 1-character string which has the ASCII value you specify. Both functions handle eight-bits of internal representation (decimal 0 to 255). The ASC and CHR$ functions can be used with the SEG$ function to analyze the character in a string.

The format of the ASC function is:

```
ASC(string)
```

where string must be a 1-character string. If string is a null string or contains more than one character, BASIC prints the ?ARGUMENT ERROR (?ARG) error message.

The ASC function returns an integer value.

The format of the CHR$ function is:

```
CHR$(expression)
```

Only one character is generated at a time.

The expression must be 0 or greater. BASIC treats arguments greater than 255 modulo 256 (e.g., BASIC treats 256 as a 0, 257 as a 1, etc.).

Consider the following example:

```
LISTNH
10 REM THIS PROGRAM WILL RETURN AN INPUT LETTER AND THE 2
20 REM FOLLOWING IT ALPHABETICALLY
30 PRINT "ENTER A LETTER A THRU Z, ENTER END WHEN FINISHED"
40 PRINT "LETTER","NEXT LETTER","3RD LETTER"
50 INPUT X$
55 IF X$="END" GO TO 999
60 IF X$<"A" GO TO 190
70 IF X$>"Z" GO TO 190
80 FOR F=ASC(X$) TO ASC(X$)+2          Returns ASCII value of X$
90 IF CHR$(F)>"Z" GO TO 150            Uses CHR$ to produce next
100 PRINT CHR$(F);;                    character
110 NEXT F
120 PRINT
130 GO TO 50
150 PRINT "END OF ALPHABET"
160 GO TO 50
190 PRINT "ENTRY IS NOT A LETTER A THRU Z"
210 GO TO 30
999 END

READY
RUNNH

ENTER A LETTER A THRU Z, ENTER END WHEN FINISHED
LETTER         NEXT LETTER   3RD LETTER
? E
E              F             G
? A
A              B             C
? Y
Y              Z             END OF ALPHABET
? 3
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
```

NOTE

The CHR$ function produces all printing and non-printing characters with ASCII values from 0 to 127. ASCII values between 127 and 255 produce the same character as the ASCII value minus 128, but the string produced is not equal and differs in the internal representation.

5.3.2.2 **Numbers and Their String Representation Conversions (VAL and STR$ Function)** - Two functions, VAL and STR$, convert numbers to their string representation and vice versa. You can use these functions when you want to input a numeric value in a string variable or to print a number without the spaces around it.

Consider these programs:

String Representations

```
LISTNH
10 PRINT "25"
20 PRINT "25+1"

READY
RUNNH

25
25+1

READY
```

Numbers

```
LISTNH
10 PRINT 25
20 PRINT 25+1

READY
RUNNH

 25
 26

READY
```

The program on the left prints the string representation of numbers, but the program on the right prints the numbers themselves. Note how the "25+1" on the left is printed as it is while the 25+1 on the right is evaluated to be 26.

The VAL function returns the number represented by the specified string. The format of the VAL function is:

VAL(string)

where:

string	may contain the digits 0 through 9, the letter E (for E notation) and the symbols + (plus), - (minus), and . (decimal point) and must be a string representation of a number. The string must be a numeric constant not a numeric expression.

The STR$ function converts a number to its string representation. The format of the function is

STR$(expression)

The STR$ function returns the value of expression as it would be printed by a PRINT statement but without a leading or trailing space. Use the STR$ function when you want to print a number without spaces before and after it or when you want to perform string operations or functions on a number.

Consider the following example:

```
LISTNH
5 PRINT "PROGRAM TO CALCULATE 5% INTEREST"
10 PRINT "TYPE IN AMOUNT";
20 INPUT M$                          Input string in form $xxx.xx.
30 IF POS(M$,"$",1)<>1 THEN 100      Check to see that a $ is present.
40 A$=SEG$(M$,2,LEN(M$))             Strip $ from number.
50 M=VAL(A$)                         Convert string to number.
60 I=.05*M                           Compute interest.
70 I$=STR$(I)                        Convert it to string.
80 I$=SEG$(I$,1,2+POS(I$,".",1))     Truncates interest to
85 REM                                 dollars and cents.
90 PRINT "5% INTEREST OF";M$;        Print results.
100 PRINT "IS $";I$
32767 END

READY
RUNNH

PROGRAM TO CALCULATE 5% INTEREST
TYPE IN AMOUNT? $308.90
5% INTEREST OF $308.90 IS $15.44     Note that there is
                                       no space before 15.44.

READY
```

5.3.2.3 **Binary and Octal to Decimal Conversions (BIN and OCT Functions)** - The BIN function returns the decimal value of a binary number. The binary number is represented as a string of 1s, 0s, and spaces. The BIN function ignores spaces, allowing convenient spacing of the digits. The format of the BIN statement is:

```
BIN(string)
```

The BIN function returns an integer value.

The binary number is treated as a signed 2's complement integer and its absolute value may not be larger than 2^15-1. For example:

```
LISTNH
10 PRINT BIN ("100101001")
20 PRINT BIN ("1 111 111 111 111 111")

READY
RUNNH

 297
-1

READY
```

The OCT function returns the decimal value of an octal number. The octal number is represented as a string containing the digits from 0 to 7 and spaces. The OCT function ignores spaces allowing convenient grouping of the digits. The format of the OCT function is:

```
OCT(string)
```

The OCT function returns an integer value.

The octal number is treated as a signed 2's complement integer. Its absolute value may not be larger than 2^15-1.

For example:

```
LISTNH
 10 PRINT OCT("451")
 20 PRINT OCT("177 777")

 READY
RUNNH

  297
-1

 READY
```

5.4 USER-DEFINED FUNCTIONS (DEF STATEMENT AND FN FUNCTION)

In some programs you may want to execute the same sequence of mathematical formulas in several places. You can define a sequence of operations as a user-defined function and can use this function as you use the functions BASIC provides, such as SIN and SEG$.

Names of user-defined functions consist of the letters FN, followed by a third letter, and optionally followed by a percent sign or a dollar sign. If you end the function name with a percent sign, it returns an integer. If you end the function name with a dollar sign, it returns a string. If you do not end the function name with either a percent sign or a dollar sign, then it returns a floating point number.

Legal User-Defined Function Names	Illegal User-Defined Function Names
FNA	FN1
FNC%	FNA2
FNR$	FNA%$

You must define each user-defined function once in a program with a DEF statement. You can define it anywhere in the program. The format of the DEF statement is:

DEF FNletter [[{%}]] (list)=expression
             [[{$}]]

where:

letter — is an alphabetic character that becomes part of the function name.

% — indicates the function returns an integer and becomes part of the function name.

$ — indicates the function returns a string and becomes part of the function name.

neither $ nor % indicates the function returns a real number.

list — contains between one and five dummy variables. These can be integer, floating point, or string variables. The list is in the format:

variable1 [,variable2,...,variable5]

expression — is evaluated each time the function is used. It may contain any of the dummy variables or any other variables in the program.

Ensure that the expression is the same data type, string or numeric, as indicated by the function name. If the expression is real and the function name is integer or vice versa, BASIC converts the expression to the type specified by the function name.

Once you have defined the function anywhere in the program, you can use (or call) the function. The format for calling the function is:

FNletter [{%, $}] (expression1 [,expression2,...,expression5])

where the number of expressions must be the same as the number of dummy variables in the DEF statement.

Each expression in a function call corresponds to a dummy variable in the DEF statement. When BASIC evaluates a function call, it evaluates the defining expression in the DEF statement with the expressions listed in the function call in the place of the corresponding dummy variables.

For example, the line:

```
10 DEF FNA(S)=S^2
```

causes a later statement:

```
20 LET R=FNA(4)
```

to assign a value of 16 (4^2) to R.

Consider the following two programs:

Program #1

```
LISTNH
10 DEF FNS(A)=A^A
20 FOR I=1 TO 5
30 PRINT I,FNS(I)
40 NEXT I
50 END

READY
RUNNH

 1            1
 2            4
 3            27
 4            256
 5            3125

`DY
```

Program #2

```
LISTNH
10 DEF FNS(X)=X^X
20 FOR I=1 TO 5
30 PRINT I,FNS(I)
40 NEXT I
50 END

READY
RUNNH

 1            1
 2            4
 3            27
 4            256
 5            3125

READY
```

As you can see, 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, the corresponding argument must be a string. If the DEF statement specifies a numeric variable, the corresponding argument must be numeric. BASIC converts, as necessary, a numeric argument to the type, floating point or integer, specified by the name in the DEF statement.

The defining expression can contain any constants, variables, BASIC-supplied function, or another user-defined function. 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:

```
LISTNH
10 DEF FNB(A,B)=A+X^2        Define function.
20 X=1                       Assign value to X.
30 PRINT FNB(14,87)          Evaluate function.
40 X=2                       Change value of X.
50 PRINT FNB(14,87)          Evaluate function again.
32767 END

READY
RUNNH

15                           14 + 1^2
18                           14 + 2^2

READY
```

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:

```
LISTNH
10 DEF FNA (X)=4+2           Note this function always returns
20 LET R=FNA(10)+1           a value of 6 no matter what
30 PRINT R                   the value of the argument
40 END                       is.

READY
RUNNH

 7

READY
```

Consider the following example:

```
LISTNH
1 REM MODULUS ARITHMETIC PROGRAM
5 REM FIND X MOD M
10 DEF FNM(X,M)=X-M*INT(X/M)                    Define FNM to find
15 REM                                          X MOD M.
20 REM FIND A+B MOD M
25 DEF FNA(A,B,M)=FNM(A+B,M)                    Define FNA to find
30 REM                                          A+B MOD M using FNM.
35 REM FIND A*B MOD M
40 DEF FNB(A,B,M)=FNM(A*B,M)                    Define FNB to find
41 REM                                          A*B MOD M using FNM.
45 PRINT
50 PRINT "ADDITION AND MULTIPLICATION TABLES MOD M"
55 PRINT "GIVE ME AN M"; \ INPUT M
60 PRINT \ PRINT "ADDITION TABLES MOD";M
65 GOSUB 800
70 FOR I=0 TO M-1
75 PRINT I;" ";
80 FOR J=0 TO M-1
85 PRINT FNA(I,J,M);                            Call FNA.
90 NEXT J \ PRINT \ NEXT I
100 PRINT \ PRINT \
110 PRINT "MULTIPLICATION TABLES MOD";M
120 GOSUB 800
130 FOR I=0 TO M-1
140 PRINT I;" ";
150 FOR J=0 TO M-1
160 PRINT FNB(I,J,M);                           Call FNB.
170 NEXT J \ PRINT \ NEXT I
180 GO TO 32767                                 Subroutine prints
800 REM SUBROUTINE FOLLOWS:                     table headings.
810 PRINT \ PRINT TAB(5);0;
820 FOR I=0 TO M-1
830 PRINT I; \ NEXT I \ PRINT
840 FOR I=1 TO 3*M+4
850 PRINT "-"; \ NEXT I \ PRINT
860 RETURN
32767 END

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
```

You can put DEF statements anywhere in a program. They do not have to appear before the functions are used. BASIC defines functions when a program is run. Consider this example:

```
10 DEF FNA(X)=X^2          Enter DEF statement
PRINT FNA(3)               Try calling function in
                             immediate mode
?UNDEFINED FUNCTION        Error message produced.

READY
RUNNH                      Run program, causing
                             function to be defined.

READY

PRINT FNA(3)               Now it works!
 9

READY
```

If you enter a DEF statement in immediate mode, BASIC ignores it.

If the same function is defined more than once in a program, BASIC prints the ?ILLEGAL DEFINE (?IDF) error message. Note that FNA, FNA%, and FNA$ are all different functions.

5.5 DATE AND TIME FUNCTIONS (DAT$ AND CLK$ FUNCTIONS)

BASIC provides two functions to return the current date and time. The format of these functions are:

```
DAT$
CLK$
```

For example:

```
LISTNH
10 PRINT "TODAY IS"$DAT$
20 PRINT "THE TIME IS"$CLK$

READY
RUNNH

TODAY IS 27-MAY-76
THE TIME IS 10:55:30

READY
```

CHAPTER 6

WORKING WITH DATA FILES

6.1 INTRODUCTION TO DATA FILES

You can use files to store data for future use. If you store data in your area of memory (such as with DATA statements), the data is lost each time you enter a new program. Data stored in files on peripheral devices can be used by many different programs, can be saved for future use, and can even be taken to another computer system.

There are two different kinds of BASIC data files, sequential files and virtual array files.

BASIC treats sequential files in the same way that it treats terminal input and output. When BASIC executes an INPUT statement, it requests a value from you through the terminal. When BASIC executes an INPUT # statement, it requests a value from a sequential file. When BASIC executes a PRINT statement, it writes data on the terminal. When BASIC executes the PRINT # statement, it writes data in a file. One difference between sequential files and terminal input and output is that once a file has been written BASIC can read the data.

BASIC treats virtual array files in the same way that it treats arrays in your area of memory. As you can access the elements in an array in memory in any order, you can access the elements of a virtual array file in any order. This is called random access. For comparisons of sequential files and virtual array files and arrays in memory and virtual array files, see Section 6.4.

This chapter describes data files. It is also possible to store BASIC programs in files (see Section 9.6).

6.2 FILE CONTROL STATEMENTS

Before you can access a sequential or virtual array file, you must open it, that is, associate the file with a channel number. Use the OPEN statement to do this.

If you open any files in your program, you should close them, that is, disassociate them from their channel numbers, when the program terminates. Use the CLOSE statement to do this.

6.2.1 Opening a File (OPEN Statement)

The OPEN statement can either open an existing file or create a new one. The OPEN statement associates the file with a channel number, which you use to access the file.

The format of the OPEN statement is:

```
OPEN string [{FOR INPUT }] AS FILE [#] expression
            {FOR OUTPUT}
```

where:

string	is a file specification. It can be any string expression. See your BASIC-11 user's guide for a complete description of the file specification format.
FOR INPUT	specifies opening an existing file.
FOR OUTPUT	specifies creation of a new file.
expression	is the channel number of the file. The channel number can have any integer value between 1 and the maximum allowed by the system (see your BASIC-11 user's guide). If the value of the expression is not an integer, BASIC truncates it to an integer.

The OPEN statement opens a sequential file unless the channel number specified in the OPEN statement is also specified in a DIM# statement (see Section 6.4.1). In this case, BASIC opens a virtual array file.

If you specify FOR INPUT, BASIC opens an existing file and you can only read information from it to memory.

If you specify FOR OUTPUT, BASIC creates a new file. Any existing file with the same file specification is superseded when the new file is closed (see Section 6.2.2). What actually happens to the old file is system dependent; see the BASIC-11 user's guide for your system. If you specify FOR OUTPUT for a sequential file, you can only write to the file. However, if you specify FOR OUTPUT for a virtual array file, you can either write to or read from the file (see Section 6.4).

Specifying neither FOR INPUT nor FOR OUTPUT for an existing sequential file is equivalent to specifying FOR INPUT. If the sequential file does not exist, specifying neither is equivalent to specifying FOR OUTPUT.

If you specify neither FOR INPUT nor FOR OUTPUT for an existing virtual array file and the file exists, the existing file is opened, but, unlike specifying FOR INPUT, you can both write to and read from it. To update an existing virtual file, open it with neither FOR INPUT nor FOR OUTPUT specified. Specifying neither FOR INPUT nor FOR OUTPUT for a virtual array file that does not exist is equivalent to specifying FOR OUTPUT.

Consider these examples of the OPEN statement.

```
10 OPEN "DATA1" FOR INPUT AS FILE #1
```

Opens the existing file specified by DATA1 and associates it with channel 1. You can only read from the file.

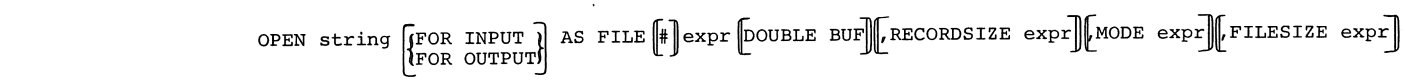
```
15 N=5
20 OPEN "MONEY" FOR OUTPUT AS FILE N
```

Creates a new file and supersedes any existing file specified by MONEY and associates it with channel 5.

```
30 OPEN "LP:" FOR OUTPUT AS FILE #2
```

Opens the line printer for output and associates it with channel 2. Note that to use a write-only device such as the line printer, you must specify FOR OUTPUT.

You can also use some system-dependent options in the OPEN statement. These allow greater flexibility in accessing files. The complete format of the OPEN statement is:



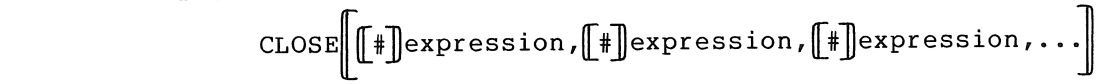
The effects of DOUBLE BUF, RECORDSIZE, MODE, and FILESIZE are described in your BASIC-11 user's guide.

6.2.2 Closing a File (CLOSE Statement)

The CLOSE statement closes the files specified and disassociates them from their channel numbers. After you close a file, you can not access the file until you re-open it.

All programs that open files should close them before terminating. If you do not close your file, you can accidently delete it. An existing file with the same file specification is not superseded until the new file is closed.

The format of the CLOSE statement is:



where:

expression	specifies the channel number of a file to be closed.

If no expressions are specified, then BASIC closes all open files.

Consider these examples of CLOSE statements:

```
100 CLOSE #1
```

Closes the file associated with channel 1.

```
110 B=4
120 CLOSE #2,B,6+1
```

Closes the files associated with channels 2,4, and 7.

```
130 CLOSE
```

Closes all open files.

If a program opens but does not close files, BASIC closes all files when it executes a CHAIN (see Section 8.1) or END statement or when it terminates the program after executing the highest numbered line in

the program. BASIC does not close files after executing a STOP statement.

6.3 USING SEQUENTIAL FILES

You use sequential files in the same way that you use terminal input and output. However, sequential files allow you to manipulate much larger amounts of data in a much shorter time than do terminal input and output.

BASIC accesses data stored in sequential files serially; you must read the entire file to read the last item of data.

You can open a sequential file for input or for output, but you can not do both at the same time. To update an existing sequential file you must open it for input, open a new file for output, then read the data from the input file and write the data including any changes you want to the output file.

The INPUT #expression and the LINPUT #expression statement (see Section 6.3.1) read data from a sequential file opened for input. Data is read from the file in the same way that it is read from the terminal.

The PRINT #expression statement (see Section 6.3.2) prints data to a sequential file opened for output. The expression after the # (number sign) must have the same value as the expression specified in the OPEN statement. BASIC stores the same data in a file that it would print on a terminal, including spaces and line terminators.

Once you have opened a data file for input, you can test for the end-of-file condition with the IF END # expression statement (see Section 6.3.3) and you can restore it to the beginning with the RESTORE # expression statement (see Section 6.3.4).

6.3.1 Reading Data from a Sequential File (INPUT # and LINPUT # Statements)

The INPUT # statement reads data from a file and assigns values to the specified variables. The format of the INPUT # statement is:

```
INPUT #expression,variable1[,variable2,variable3,...]
```

where:

expression	is the channel number of the file except if the value of the expression is 0, the values are input from the user's terminal.
variable(s)	are assigned the value(s) read in from the file. The variables can be any string, integer, floating point, or subscripted variables.

You can also use a colon (:) instead of a comma after the expression.

If the line of data in the file contains more data than there are variables, BASIC ignores the excess data. If there is not enough data, BASIC looks for more data on the next line.

Consider the following example which reads 10 numbers from a file and prints the sum.

```
LISTNH
10 OPEN "NUMBER" FOR INPUT AS FILE #1
20 FOR I=1 TO 10
30 INPUT #1, N
40 T=T+N
50 NEXT I
60 PRINT "THE TOTAL IS";T
70 CLOSE #1
100 END

READY
RUNNH

THE TOTAL IS 187.3

READY
```

The LINPUT # statement inputs a string from a file. BASIC treats LINPUT # just as it treats LINPUT; all characters on the input line, including commas and quotation marks, are assigned to the string. The format of the LINPUT # statement is:

LINPUT #expression,variable1[,variable2,variable3,...]

where:

expression	specifies the channel number of the file except if it is 0, BASIC inputs a line from the user's terminal.
variable(s)	are assigned the value(s) of all the characters read from the file up to the line terminator(s). The variables can only be string variables or subscripted string variables.

You can also use a colon (:) instead of a comma after the expression.

NOTE

The INPUT # statement and the LINPUT # statement cannot be used in immediate mode.

6.3.2 Storing Data in a Sequential File (PRINT # Statement)

The PRINT # statement prints data to the specified file. The format of the statement is:

PRINT #expression[,list]

where:

expression	is the channel number of the file except if the value of the expression is 0, data is printed on the user's terminal.
list	contains the items to be printed. It can contain any numeric and string expressions and TAB functions. Items can be separated by commas or semicolons with the resulting output format the same as the format of the simple PRINT statement.

You can use a colon (:) instead of a comma after the expression.

If there are no items in the list, BASIC prints a blank line to the file. When there are no items in the list, you need not specify the comma or colon after the expression.

Consider the following example, which creates a sequential file from data stored in DATA statements.

```
LISTNH
10 OPEN "NAMES" FOR OUTPUT AS FILE #1        Opens file.
20 READ A$,A                                 Reads data from
25 REM                                       program.
30 IF A$=""THEN 32000                        Checks to see if it
35 REM                                       is the last data.
40 PRINT #1,A$;",";A                         Prints two data
45 REM                                       items
50 GO TO 20                                  separated by a
55 REM                                       comma.
25000 DATA "SARAH", 187.2, "JOE", 117.45
25010 DATA "JANE", 200, "JIM", 89
25020 DATA "MALCOLM", 125
25030 DATA "", 0
32000 CLOSE #1                               Closes the file to
32767 END                                    make it permanent.

READY
RUNNH

READY
```

After you run this program, the file specified by NAMES contains:

```
SARAH, 187.2
JOE, 117.45
JANE, 200
JIM, 89
MALCOLM, 125
```

You should print a comma (in a string constant) between each data item on a line. If you do not do this, you will not be able to input the data items from the file.

Consider the following example, which reads the data file created in the previous program and prints the results on the terminal.

```
LISTNH
10 OPEN "NAMES" AS FILE #2    Opens existing file.
20 INPUT #2,N$,N              Inputs data from file.
30 PRINT N$,N                 Prints results.
40 GO TO 20

READY
RUNNH

SARAH          187.2
JOE            117.45
JANE           200
JIM            89
MALCOLM        125

?OUT OF DATA AT LINE 20       All data in file has been read.

READY
```

Note that the line reading the file has the same number of variables as the line printing the file has data values.

Program line reading the file	Program line creating the file
`20 INPUT #2,N$,W`	`40 PRINT #1,A$;",";A`

See Section 6.3.3 for a way to detect the end of the file before BASIC prints an error message.

The PRINT #expression USING statement prints formatted data to a file (see Chapter 7).

6.3.3 Checking for the End of Input File (IF END # Statement)

Use the IF END # statement, a special form of the IF statement, to check for the end of a file. The form of the statement is:

IF END[#]expression{THEN statement / THEN line number / GO TO line number}

where:

expression	is the channel number of the file. The value of expression can not be 0 and the file associated with the expression can not be a terminal.

If the next attempt to input a value would produce the ?OUT OF DATA (?OOD) error message, BASIC executes the statement after the THEN or transfers control to the specified line number. Otherwise BASIC transfers control to the next statement as it does with an IF statement (see Section 4.1.3).

Consider the following example:

```
LISTNH
10 OPEN "NAMES" AS FILE #1
20 IF END #1 THEN PRINT "END OF FILE" \ GO TO 32000
30 INPUT #1,A$,A
40 PRINT A$,A
50 GO TO 20
32000 CLOSE #1
32767 END

READY
RUNNH

SARAH           187.2
JOE             117.45
JANE            200
JIM             89
MALCOLM         125
END OF FILE

READY
```

Line 20 checks for the end-of-file condition.

Note that the IF END # statement tests if there is one more data item. If there is one data item left when IF END checks and your INPUT # statement requests two data items, BASIC prints the ?OUT OF DATA (?OOD) error message.

6.3.4 Restoring a File to the Beginning (RESTORE # Statement)

The RESTORE # statement resets the specified sequential input file from its current position to its beginning. The format of the RESTORE # statement is:

RESTORE #expression

where:

expression is the channel number of the file to be restored.

6.4 USING VIRTUAL ARRAY FILES

Use virtual array files when you want to randomly access the data in a file or when an array is too large to fit in memory.

Virtual array files have several advantages over sequential files:

1. You can access them in a random, non-sequential manner. The last element in a virtual array can be accessed as quickly as any element. Contrast this with a sequential file where you must read the entire file before reading the last element.

2. When BASIC stores data in virtual arrays, it does not convert them to ASCII characters but rather stores them in internal binary representation. Consequently, there is no loss of precision caused by data conversion. There is some loss of precision with sequential files.

3. You can update virtual array files without copying the entire file.

Virtual array files also have several advantages over arrays stored in memory:

1. Virtual array files allow you to create much larger arrays than can be stored in available memory.

2. Data can be stored in virtual array files.

Virtual arrays also have several restrictions, which do not apply to arrays in memory:

1. Virtual array files are slower because BASIC must read the file before manipulating the data.

2. Although BASIC strings in memory can have any length (up to 255 characters), you cannot use these dynamic length strings in virtual arrays. Strings in virtual array files have a fixed maximum length, which you specify in the DIM # statement. The maximum character length can be any number from 1 to 255. Strings longer than the maximum are truncated. Strings shorter than the maximum are padded with trailing nulls. When you access an array element, all trailing nulls are removed.

3. You can only have one virtual array dimensioned in each DIM # statement.

4. You cannot use a virtual array element to store a result from a CALL statement (see Section 8.3).

6.4.1 Dimensioning Virtual Arrays (DIM # Statement)

To use a virtual array file, place a DIM # statement and an OPEN statement with the same channel in a program. After the file is opened, the elements of the array can be used in the same way as elements of an array in memory.

The form of the virtual array DIM statement is:

DIM #integer1, array [=integer2]

where:

integer1 — is an integer constant (with or without a percent sign) that specifies the channel number of the file.

array — is any 1- or 2-dimensional array. It has the same format as in the standard DIM statement, specifically:

variable(integer[,integer])

where the variable can be any string, integer, or floating point variable name and the integer(s) represents the subscripts.

integer2 — is an integer constant (with or without a percent sign) that specifies the maximum length for elements in a virtual string array. Its value must be in the range 1 to 255. If it is omitted for a string array, the maximum length is 16.

To access the data in an existing virtual array file, ensure that the DIM # statement specifies the same data type and subscripts that are specified in the program that created the file. The variable name associated with the file can be different from the original as long as it is the same variable type.

Consider the following examples:

```
10 DIM #1,A%(2000)
20 OPEN "INTDAT" FOR OUTPUT AS FILE 1
```

Creates a new 2001-element integer virtual array file. The virtual array is named A% in this program. You can assign values to the element and then use the values.

```
10 DIM #2,F9(100,10)
20 OPEN "VARAY" FOR INPUT AS FILE #2
```

Opens an existing 2-dimensional, floating point virtual array file. The virtual array is named F9 in this program. Only input is allowed; if you try to assign a value to an element of the array, BASIC prints the ?ILLEGAL I/O DIRECTION (?IID) error message.

```
10 DIM #2,A$(100)=32
20 OPEN "STRNGS" AS FILE #2
```

Opens a 1-dimensional string virtual array file where the maximum string length is 32 characters. The virtual array is named A$ in this program. If the file specified by STRNGS already exists, it is opened for updating. If no file exists, then a new one is created.

You should close a virtual file when your program terminates. If you do not close an output virtual array, it does not become permanent. If you do not close an updated virtual array, the assignments you have made may not be incorporated into the file.

As an example of a use of virtual array files, consider the problem of an information retrieval system for a small organization. Assume each employee needs a 128-character record containing name, home address, home phone, work station and phone extension. If this information is maintained in a long sequential file, it would take a long time to locate the information for any employee and it would be impossible to update without rewriting the file for every 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.

In the example below, 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 faster to search through the index file than to search through the master 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.

```
10 DIM #1,B%(1000)                          1001 elements in badge
15 REM                                      number file.
20 DIM #2,B$(1000)=128                      1001 elements in master
25 REM                                      file.
30 OPEN "BADGE" AS FILE #1                  Open badge file.
40 OPEN "MASTER" AS FILE #2                 Open master file.
50 PRINT "WHAT IS THE BADGE NUMBER";        Specify a badge number.
60 INPUT N
70 FOR I%=1 TO 1000                         Search for a match in
80 IF B%(I%)=N THEN 200                     the badge number
90 NEXT I%                                  file.
100 PRINT "NO SUCH EMPLOYEE"                If a match is not found
110 GO TO 32700                             print message and
115 REM                                     terminate.
200 PRINT "NAME IS ";SEG$(B$(I%),10,30)     If a match is found,
32700 CLOSE #1,#2                           get record of employee,
32767 END                                   B$(I%), and extract the
                                            name from the record
                                            (the name is stored
                                            from the 10th to the
                                            30th characters). Then
                                            terminate.
```

6.5 RENAMING A FILE (NAME STATEMENT)

Use the NAME statement to change the name of data files. The NAME statement does not alter the contents of the file. The format of the NAME statement is:

```
NAME string1 TO string2
```

where:

string1	is a file specification of the file to be renamed.
string2	is the new file specification. If you specify a device in string1, you must specify the same device in string2.

For example:

```
10 NAME "MONEY" TO "ACCNTS"
```

Changes the name of the file specified by MONEY to ACCNTS.

See your BASIC-11 user's guide for information on system dependent file specifications.

6.6 DELETING A FILE (KILL STATEMENT)

Use the KILL statement to delete data files. The format of the KILL statement is:

line number KILL string

where:

string	is a file specification of the file to be deleted.

After you delete a file, you can not open or access it in any way.

For example:

```
10 KILL "DATA"
```

Deletes the file specified by DATA.

See your BASIC-11 user's guide for information on system dependent file specifications.

CHAPTER 7

FORMATTED OUTPUT - THE PRINT USING STATEMENT

7.1 INTRODUCTION TO PRINT USING

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 data on the output line, and thus enables 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

```
LISTNH
10 PRINT 1
20 PRINT 100
30 PRINT 1.00000E+06
40 PRINT 100.3
50 PRINT .0123456

READY
RUNNH

 1
 100
 1.00000E+06
 100.3
 1.23456E-02

READY
```

PRINT USING

```
LISTNH
10 PRINT USING "########.##";1
20 PRINT USING "########.##";100
30 PRINT USING "########.##";1.00000E+06
40 PRINT USING "########.##";100.3
50 PRINT USING "########.##";.0123456

READY
RUNNH

       1.00
     100.00
 1000000.00
     100.30
        .01

READY
```

Note that PRINT left-justifies numbers and prints certain numbers in E format. These characteristics make it difficult to compare numbers. In contrast, PRINT USING allows you to format the numbers so that the decimal points are aligned making it much easier to compare the column of numbers.

You can designate the following formats with PRINT USING:

1. Numbers
    a. Number of digits
    b. Location of decimal point
    c. Inclusion of symbols (trailing minus signs, asterisks, dollar signs, commas)
    d. Exponential format

2. Strings

   a. Number of characters
   b. Left-justified format
   c. Right-justified format
   d. Centered format
   e. Extended field format

The format of the PRINT USING statement is:

```
PRINT USING string, list
```

where:

string	is a coded format image of the line to be printed. The string 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.

Consider the following example.

```
10 PRINT USING "HI 'LLLLL YOU WEIGH ###.# LBS.","PAUL",145
```

The format string is:

```
"HI 'LLLLL YOU WEIGH ###.# LBS."
```

and the list contains two data items:

```
the string constant "PAUL"
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 ###.#, which corresponds to the second data item, 145. When BASIC prints the line it prints each data item in the same position as its field and in the format specified by the field. The rest of the format string, namely "HI YOU WEIGH . LBS." is just a message that is printed. The output of this example is:

```
LISTNH
10 PRINT USING "HI 'LLLLL YOU WEIGH ###.# LBS.","PAUL",145

READY
RUNNH

HI PAUL  YOU WEIGH 145.0 LBS.

READY
```

The way to use format strings to print items is described in Sections 7.2 and 7.3 and is summarized in Section 7.4.

7.2 PRINTING NUMBERS WITH PRINT USING

You can use PRINT USING to print numbers in the format that you want. You can specify the number of digits reserved for a number, the location of the decimal point, certain special symbols to be printed with the number, or that the number be printed in E format.

7.2.1 Specifying the Number of Digits

With PRINT USING, you specify the number of places reserved for digits in a field. Specify the number of places with a corresponding number of number signs (#).

For example:

```
LISTNH
10 PRINT USING "###",123        Three digits reserved.
20 PRINT USING "#####",12345    Five digits reserved.

READY
RUNNH
123
12345

READY
```

If there are not enough digits to fill the field specified, BASIC prints spaces before the first digit. For example:

```
LISTNH
10 PRINT USING "#####",1
20 PRINT USING "#####",-10
30 PRINT USING "#####",1709
40 PRINT USING "#####",12345

READY
RUNNH

    1
  -10
 1709
12345

READY
```

BASIC rounds numbers printed with the PRINT USING statement. For example:

```
LISTNH
10 PRINT USING "###",126.7
20 PRINT USING "#",5.9
30 PRINT USING "#",5.4

READY
RUNNH

127             126.7 is rounded to 127.
6               5.9 is rounded to 6.
5               5.4 is rounded to 5.

READY
```

7.2.2 Specifying the Location of the Decimal Point

You can reserve places for any number of digits on both sides of the decimal point. Specify the location of the decimal point by placing a decimal point in the number signs (#) in a field. BASIC always prints the digits to the right of the decimal point even if they are 0s. Consider the following example:

```
LISTNH
10 PRINT USING "##.###";5.72
20 PRINT USING "##.###";39.3758

READY
RUNNH

 5.720
39.376

READY
```

Note that BASIC with 5.72 prints spaces to the left of the decimal point, as necessary, but prints 0s to the right of the decimal point. Also note that 39.3758 is rounded to 39.376.

If there are any number signs to the left of the decimal point, BASIC prints at least one digit to the left of the decimal point. It is 0 as necessary.

An exception to this format rule is when you specify only one number sign to the left of the decimal point for a number that is negative. In that case BASIC prints the minus sign to the left of the decimal point instead of a 0.

7.2.3 Printing a Number That is Larger Than the Field

If you have not reserved enough digits for a number, BASIC prints a percent sign (%) followed by the number and it ignores the format specified by the field. BASIC prints the number in the standard PRINT statement format. After BASIC prints the number, it completes the rest of the PRINT USING statement as usual. Consider the following example:

```
LISTNH
10 PRINT USING "###.##";256.786      Enough digits reserved.
20 PRINT USING "##.##";256.786       Not enough digits reserved.

READY
RUNNH

256.79
% 256.786

READY
```

The number is printed correctly (with rounding). There are only two number signs to the left of the decimal point; therefore, the number does not fit. The number is printed in the usual PRINT statement format, with a space before and after it and with no rounding.

Be sure to enter a number sign to the left of the decimal point for every digit in the number to the left of the decimal point. Also add one extra for the sign if the number may be negative. (For an alternative method for reserving a place for the minus sign, see Section 7.2.4.1).

For example:

Field	There are enough places for	But not enough places for
###.##	100.569	-100.569
.####	.12579	-.12579

A number can also become larger than its field because rounding increases the number of places needed. For example:

```
LISTNH
10 PRINT USING ".###",.999      Enough places reserved.
20 PRINT USING ".##",.999       Rounds to 1.00; not enough
25 PRINT                        places reserved.
30 PRINT USING "#.##",.999      Enough places reserved.

READY
RUNNH

.999                            Format as specified by field.
% .999                          % followed by number in PRINT
                                statement format.
1.00                            Enough places reserved and the
                                number is printed, correctly
READY                           rounded.
```

7.2.4 Printing Numbers with Special Symbols

You can use the PRINT USING statement to print a number with a trailing minus sign (instead of the minus sign appearing before the number), with asterisks filling blanks before the first digit, with a dollar sign printed before the first digit or with commas inserted every three digits.

7.2.4.1 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. The trailing minus sign is often used to indicate a debit but can be used with any numbers. You must use the trailing minus sign to print a number in an asterisk fill or floating dollar sign field (see Sections 7.2.4.2 and 7.2.4.3).

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.

Consider the following examples.

Standard Fields

```
LISTNH
10 PRINT USING "###.##",-10.54
20 PRINT USING "###.##",10.54

READY
RUNNH

-10.54
 10.54

READY
```

Fields with Trailing Minus Signs

```
LISTNH
10 PRINT USING "##.##-",-10.54
20 PRINT USING "##.##-",10.54

READY
RUNNH

10.54-
10.54

READY
```

7.2.4.2 **Printing Numbers with Asterisk Fill** - To print a number with asterisks (*) filling any blank spaces before the first digit, start the field with two asterisks. For example:

```
LISTNH
10 PRINT USING "**##.##",27.95
20 PRINT USING "**##.##",107
30 PRINT USING "**##.##",1007.5

READY
RUNNH

**27.95         Asterisks fill two blank spaces.
*107.00         Asterisk fills one blank space.
1007.50         No blank spaces.

READY
```

Note that the asterisks cause asterisk fill and reserve places for two digits.

To print a number which can be negative, as well as positive, in an asterisk fill field, specify a trailing minus sign in the field. For example:

```
LISTNH
10 PRINT USING "**##.##-",27.95
20 PRINT USING "**##.##-",-107
30 PRINT USING "**##.##-",-1007.5

READY
RUNNH

**27.95
*107.00-
1007.50-

READY
```

If you attempt to print a negative number in an asterisk fill field that does not include a trailing minus sign, BASIC halts execution and prints the ?PRINT USING ERROR (?PRU) message (see Section 7.5.1).

7.2.4.3 **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. If the number can be negative as well as positive, end the field with a trailing minus sign. Consider the following example:

```
LISTNH
10 PRINT USING "$$##.##",77.44
20 PRINT USING "$$##.##",304.55
30 PRINT USING "$$##.##",2211.42
35 PRINT
40 PRINT USING "$$##.##-",-125.6
45 REM
47 REM
50 PRINT USING "$$##.##-" 127.82

READY
RUNNH

 $77.44
$304.55
% 2211.42

$125.60-
$127.82

READY
```

There are enough places reserved in line 10 and 20 but not 30.

Negative value in floating dollar sign with trailing minus field.

Positive number in same field as line 40.

There were not enough places to print $ and 2211.42.

Trailing minus printed.

Trailing space printed.

Note that the dollar signs reserve places for the dollar sign and one digit; the dollar sign is always printed. Contrast this with the asterisk fill field where BASIC prints asterisks only if there would have been leading spaces.

If you attempt to print a negative number in a dollar sign field which does not include a trailing minus sign, BASIC halts execution and prints the ?PRINT USING ERROR (?PRU) message (see Section 7.5.1).

7.2.4.4 **Printing Numbers with Commas** - To have commas, inserted in a number, place a comma anywhere in the field to the left of the decimal point (including an assumed decimal point). BASIC then 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. For example:

```
LISTNH
10 PRINT USING "##,###",10000
20 PRINT USING "##,###",759
30 PRINT USING "$$#,#####",25694.3
40 PRINT USING "**#,###",7259
50 PRINT USING "####,#.##",25239

READY
RUNNH

10,000
   759
$25,694.30
**7,259
25,239.00

READY
```

Commas can be combined with $$ and **.
A comma can be anywhere in the field to the left of the decimal point.

Comma printed.
Comma not printed because no digit would appear to its left.

7.2.5 Printing Numbers in E Format

To print a number in E format, place four circumflexes (^^^^, also called up-arrows) at the end of the field. The four circumflexes reserve space for the capital letter E, followed by a plus or minus sign (which indicates a positive or negative exponent, respectively), and then the 2-digit exponent. In exponential format the digits to the left of the decimal point are not filled with spaces. Instead the first nonzero digit is shifted to the leftmost place and the exponent is equal to the number of digits that the decimal point is shifted from the number in standard notation (see Section 2.2.1). Consider the following example:

```
LISTNH
10 PRINT USING "###.##^^^^",5
20 PRINT USING "###.##^^^^",1000

READY
RUNNH

 500.00E-02
 100.00E+01

READY
```

Note that 5 is shifted to the left but 1 is shifted to the right. Each exponent is adjusted.

You must reserve a place for a minus sign to the left of the decimal point.

You cannot use exponential format with asterisk fill, floating dollar sign, or trailing minus formats.

7.2.6 Fields Which Exceed BASIC's Accuracy

If a field contains more places than there are digits of accuracy, BASIC prints 0s in all the places following the last significant digit. See your BASIC-11 user's guide for the number of digits of accuracy in your system.

7.3 PRINTING STRINGS WITH THE PRINT USING STATEMENT

By using the PRINT USING statement, you can specify whether strings are printed in a left-justified, right-justified, or centered format. String fields start with a single quotation mark ('). The single quotation mark is optionally followed by a contiguous series of uppercase Ls, Rs, Cs, or Es representing left-justified, right-justified, centered, and extended string fields, respectively.

If a string is larger than its specified string field, BASIC prints as much of the string as fits and ignores the rest. The only exception is that for extended fields BASIC prints the entire string.

7.3.1 1-Character String Fields

A string field consisting of only a single quotation mark is a 1-character string field. BASIC prints the first character of the string expression corresponding to the 1-character string field and ignores all following characters. For example:

```
LISTNH
10 PRINT USING " ' ";"ABCDE"

READY
RUNNH

 A                                  BASIC ignores the characters
                                    BCDE.
READY
```

7.3.2 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 in the field, BASIC prints spaces after the string. If there are more characters than places in the field, BASIC truncates the string and does not print the excess characters.

A left-justified field is specified by a single quotation mark followed by a series of capital Ls. For example:

```
LISTNH
10 PRINT USING "'LLLLLL";"ABCD"
20 PRINT USING "'LLLL";"ABC"
30 PRINT USING "'LLLL";"12345678"

READY
RUNNH

ABCD
ABC
12345                               BASIC ignores the excess
                                    characters 678.
READY
```

7.3.3 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. If there are more characters than places in the field, BASIC prints the string as if it were in an equivalent left-justified string field.

A right-justified field is specified by a single quotation mark followed by a series of capital Rs. For example:

```
LISTNH
10 PRINT USING "'RRRRRR";"ABCD"
20 PRINT USING "'RRRRRR";"A"
30 PRINT USING "'RRRRRR";"XYZ"

READY
RUNNH

  ABCD
     A
   XYZ

READY
```

If there are more characters than places, BASIC left-justifies the string and does not print the excess characters.

7.3.4 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. If there are more characters than places in the field, BASIC prints the string as if it were in an equivalent left-justified string field.

A centered field is specified by a single quotation mark followed by a series of capital Cs. For example:

```
LISTNH
10 PRINT USING "'CCCCCC", "A"
20 PRINT USING "'CCCCCC", "AB"
30 PRINT USING "'CCCCCC", "ABC"
40 PRINT USING "'CCCCCC", "ABCD"
50 PRINT USING "'CCCCCC", "ABCDE"

READY
RUNNH

   A
  AB
  ABC
 ABCD
 ABCDE

READY
```

If there are more characters than there are places in the field, BASIC left-justifies the string and does not print the excess characters.

7.3.5 Printing Strings In Extended Fields

The extended field is the only field that ensures the printing of 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 specified by a single quotation mark followed by a series of capital Es.

Consider the following example which uses extended, left-justified, right-justified, and centered fields.

```
LISTNH
100   FS="++'CCCC++'EEEE++'LLLL++'RRRR++"
110   INPUT A$
120   IF A$="STOP" GO TO 150
130   PRINT USING F$,A$,A$,A$,A$
140   GOTO 110
150   END

READY
RUNNH

? ABCD
++ABCD ++ABCD ++ABCD ++ ABCD++
? ABCDEFG
++ABCDE++ABCDEFG++ABCDE++ABCDE++
? A
++  A  ++A    ++A    ++    A++
? AB
++ AB  ++AB   ++AB   ++   AB++
? STOP

READY
```

The underlined field has been extended. Note how the rest of the line is displaced two places.

7.4 SUMMARY OF THE PRINT USING STATEMENT FORMAT

The format of the PRINT USING statement is:

PRINT [#expr,] USING string, list

where:

expr	specifies the channel number. If it is omitted output is to the user's terminal.
string	is a coded format image of the line to be printed. The string is called the format string. It can be any string expression; however, if it is a string constant, it should be delimited with double quotation marks, not single quotation marks.
list	contains the items to be printed. The items can be any string or numeric expressions. They are separated by either commas or semicolons. Optionally, the list can end with a comma or semicolon to suppress the printing of the carriage return.

The format string contains a series of fields corresponding to the items in the list. Fields correspond to items based on the order in which they appear. BASIC prints the item at the location in the line and according to the format specified by its corresponding field.

The type (string or numeric) of each field must be the same as the type of its corresponding item.

The format string is a template of the line to be printed. Each character in a field reserves one place for the data item. Any character not in a field is printed exactly as it is. Consider the following example:

```
LISTNH
10 PRINT USING "THERE ARE #### CHIPS AND ### 'RRRR.",327,27,"PINS"

READY
RUNNH

THERE ARE  327 CHIPS AND  27  PINS.

READY
```

Compare the format string and the printed line:

```
format string:    THERE ARE #### CHIPS AND ### 'RRRR.
printed line:     THERE ARE  327 CHIPS AND  27  PINS.
```

Each column in the format string corresponds to the same column in the printed line. (The only exception is when extended string fields are used; these fields expand to the size of the string.)

It is not necessary to have the same number of fields in the format string as items in the list. If there are fewer fields than items, BASIC uses the same format string again on a new line for the remaining items. For example:

```
LISTNH
10 X=5 \ Y=10 \ C=17 \ D=185
20 PRINT USING "## + ### = ####",X,Y,X+Y,C,D,C+D

READY
RUNNH

 5 +  10 =   15
17 + 185 =  202

READY
```

The format string in line 20 contains three format fields:

```
"## + ### = ####"

  1    2      3
```

but there are six data items:

```
X  Y   X+Y    C  D   C+D

1  2    3     4  5    6
```

BASIC prints the first three data items:

```
 5 +  10 = 15
```

and then repeats the format string on a new line for the other three:

```
17 + 185 = 202
```

But if there are more fields than items, BASIC ignores the excess fields. Any characters following the first unused field are not printed. For example:

```
LISTNH
10 PRINT USING "THERE ARE ### PADS IN THE 'LLLLL STOCKROOM", 25

READY
RUNNH

THERE ARE 25 PADS IN THE

READY
```

There are two fields, ### and 'LLLLL, but only one data item. Consequently BASIC ignores the second field ('LLLLL) and does not print any of the characters after it (STOCKROOM).

The items in the list are separated by either commas or semicolons. It does not matter which you use.

If the output of a PRINT USING statement is longer than one line, BASIC prints the remainder of the statement on the next line.

7.4.1 Format of Numeric Fields

Numeric fields are composed of the special characters described in Table 7-1.

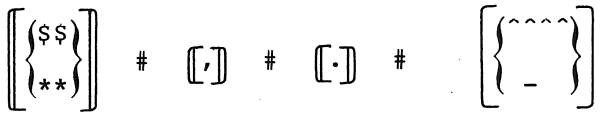
Table 7-1
Format Characters For Numeric Fields

Character	Effect on Format
# number sign	Reserves place for one digit.
. decimal point	Determines location of decimal point.
, comma	Causes a comma to be printed between every third digit.
** two asterisks	Cause leading asterisks to be printed before the first digit. The field formed is called an asterisk fill field. Asterisks 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. Dollar signs also reserve places for one dollar sign and one digit.

Table 7-1 (Cont.)
Format Characters For Numeric Fields

Character	Effect on Format
^^^^ four circumflexes	Cause number to be printed in E format. Circumflexes 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.

Valid fields can be formed by combining these characters in this format:



Each number sign (#) in the above format represents any number of number signs. One exception to this format is that neither two dollar signs nor two asterisks can be combined with four circumflexes.

For example:

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 **

7.4.2 Format of String Fields

String fields are composed of a single quotation mark optionally followed by a contiguous series of capital Ls, Rs, Cs, or Es. These characters' effect on the format are described in Table 7-2.

Table 7-2
Format Characters for String Fields

Character	Effect on Format
' single quotation mark	Starts string field and reserves place for one character.
L	Causes string to be left-justified and reserves place for one character.
R	Causes string to be right-justified and reserves place for one character.
C	Causes string to be centered in field and reserves place for one character.
E	Causes string to be left-justified, expands field, as necessary, to print the entire string and reserves place for one character.

For example:

Valid String Fields	Sample Output	Description
'LLLLL	ABC	6-character, left-justified string field
'CCCCCC	ABC	7-character, centered string field
'	A	single character string field

Invalid String Fields	Reason
'LLRRR	Cannot combine two letters in one field.
"RRRR	Field must start with single quotation mark.
'CCCC'	A single quotation mark should only be at the beginning of the field.

7.5 PRINT USING STATEMENT ERROR CONDITIONS

There are two types of PRINT USING error conditions, fatal and nonfatal. When a fatal error occurs, BASIC stops executing the program and prints the ?PRINT USING ERROR (?PRU) message. When a nonfatal error occurs, BASIC continues to execute the program, although the resulting output may not be in the format intended.

7.5.1 Fatal Error Conditions

The ?PRINT USING ERROR (?PRU) error message is produced if:

1. The format string is not a legal 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 which does not specify a trailing minus.
6. The items in the list are separated by characters other than a comma or semicolon.

7.5.2 Nonfatal Error Conditions

Nonfatal error conditions occur if:

1. A number does not fit in the field.
2. A string does not fit in the field.
3. A field contains an illegal combination of characters.
4. Text to be printed forms a valid field.

If a number is larger than the field allows, BASIC prints a percent sign followed by the number in the standard PRINT format.

If a string is larger than any field other than an extended field, BASIC truncates the string and does not print the excess 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 unintended field may cause a fatal error condition.

Consider the following examples of illegal combinations of characters in numeric fields.

Illegal Combinations

```
LISTNH
10 PRINT USING "$$******.##",5.41, 16.30

READY
RUNNH

$5**16.30

READY
```

Two dollar signs are combined with two asterisks. $$ is a complete field and **###.## forms a second valid field. $5 is printed by $$ and **16.30 is printed by **###.##.

```
LISTNH
10 PRINT USING "$$******.## 'LLL",5.41, "ABC"

READY
RUNNH

$5
?PRINT USING ERROR AT LINE 10

READY
```

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 unintended numeric field **###.##.

```
LISTNH
10 PRINT USING "##.#^^^^",5.43000E+09

READY
RUNNH

% 5.43000E+09^^^

READY
```

Field has only three not four circumflexes. The number does not fit in the field ##.#, a percent sign and the number are printed followed by the three circumflexes.

```
LISTNH
10 PRINT USING "'LLEEE","VWXYZ"

READY
RUNNH

VWXEEE

READY
```

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:

```
10 PRINT USING "THERE ARE ### # ## PENNY NAILS",123,4,16,6
```

is an attempt to print

```
THERE ARE 123 #  4 PENNY NAILS
THERE ARE  16 #  6 PENNY NAILS
```

but instead produces

```
RUNNH
THERE ARE 123 4 16 PENNY NAILS
THERE ARE   6

READY
```

To correctly print characters that would form a valid field, use a string field and place the characters as a string constant in the list. For example:

```
LISTNH
10 A$="THE BALANCE OF ACCOUNT '#### is $$###.##"
20 PRINT USING A$,"#",5634,107.56

READY
RUNNH

THE BALANCE OF ACCOUNT #5634 is $107.56

READY
```

This is also the only way to print a single or double quotation mark character with the PRINT USING statement. For example:

```
LISTNH
10 PRINT USING "HE SAID, 'I'M GOING.'",'"','"','"','"'

READY
RUNNH

HE SAID, "I'M GOING."

READY
```

CHAPTER 8

PROGRAM SEGMENTATION

8.1 SEGMENTING PROGRAMS WITH THE CHAIN STATEMENT

Segmenting a program is the process of breaking one large program into two or more smaller programs. You should segment a program if it is too large to fit in your area of memory. Another reason for segmenting programs is that it is easier to debug several small programs than to debug one large program. Once you have segmented a large program, you can debug each segment independently.

You can use the CHAIN statement to break up a program into segments. You create each segment as you create any program except that you end each segment (except the last) with a CHAIN statement. When BASIC executes the CHAIN statement, it transfers control from the current segment to a program segment stored in a file. (See Section 9.6 for information about storing programs in files.)

When BASIC executes the CHAIN statement, it closes any open files, changes the program name to the name in the file specification, and deletes the current program segment from your area. The program segment includes all program lines, variables (floating point, integer, and string), arrays, and user-defined functions (see Section 5.4) but does not include any variables or arrays that are listed in COMMON statements (see Section 8.1.1). Then BASIC loads and executes the program segment in the file you specify.

The memory required to run the entire program is the amount of memory required by the largest segment. Figure 8-1 shows how a program can be broken into three segments and how the memory required by the program can be reduced.

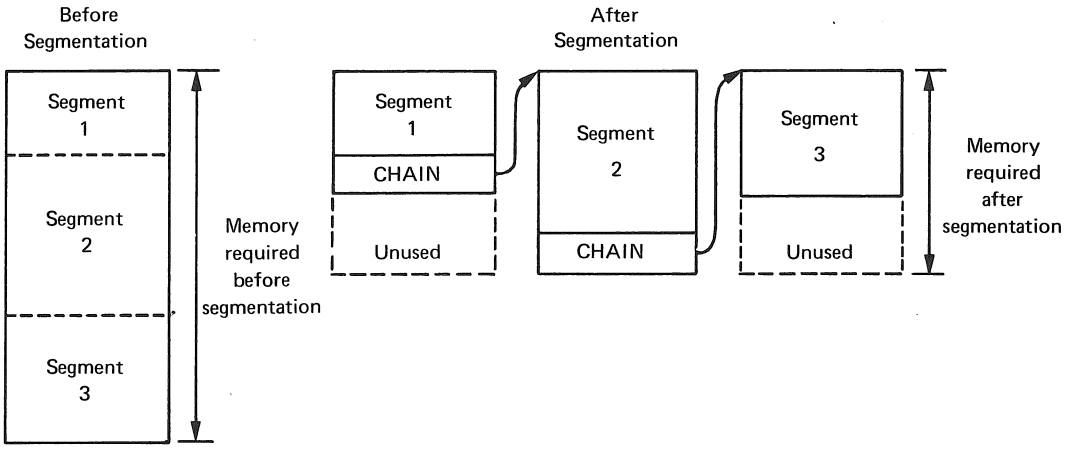


Figure 8-1 Segmenting a Program with the CHAIN Statement

Before the large program is separated into segments, the three segments are stored in your area of memory at the same time, as indicated on the left. After segmenting, the program starts with only segment 1 stored in your area of memory. After BASIC runs segment 1, segment 1 chains to segment 2, that is BASIC erases segment 1, replaces it with segment 2, and then runs segment 2. After BASIC runs segment 2, segment 2 chains to segment 3. The entire segmented program fits in the amount of memory required by the largest segment, segment 2.

The format of the CHAIN statement is:

CHAIN string [[LINE expression]]

where:

string	is a file specification. It specifies the file containing the next program segment. The string can be any string expression.
expression	specifies the line number at which BASIC starts execution in the next program segment.

If BASIC cannot find the file you specify, it closes all open files, changes the program name, and prints the ?FILE NOT FOUND (?FNF) error message. In this case BASIC does not delete any program lines, variables, arrays, or user-defined functions.

If you omit LINE and the expression, BASIC starts execution at the lowest numbered line in the next program segment. If you specify LINE and an expression whose value is not a legal line number (integers from 1 to 32767), BASIC prints the ?SYNTAX ERROR (?SYN) error message and stops program execution before it reads in the next segment. If you specify LINE and an expression and the line number is not found in the next program segment, BASIC prints the ?UNDEFINED LINE NUMBER (?ULN) error message and stops program execution after it reads in the next segment.

Consider the following example:

The file specified by "SEG1" contains:

```
 5 PRINT "SEG1 IS WORKING"                Prints identifying message.
10 OPEN "DATA1" FOR OUTPUT AS FILE #1     Opens output file.
20 FOR I=1 TO 100                         Writes out all the
30 PRINT #1,2*I                           even numbers 2 to 200
40 NEXT I                                 to the file.
50 CLOSE #1                               Closes the file.
60 CHAIN "SEG2"                           Chains to the next
70 END                                    segment.
```

The file specified by "SEG2" contains:

```
 5 PRINT "SEG2 IS WORKING"                Prints identifying message.
10 OPEN "DATA1" FOR INPUT AS FILE #1      Opens existing file.
20 FOR I=1 TO 100                         Inputs the numbers
30 INPUT #1,J                             from the files
40 T=T+J                                  and adds them together,
50 NEXT I                                 storing the total in T.
60 PRINT "THE TOTAL IS";T                 Prints the total on the
65 REM                                    terminal.
70 CLOSE #1                               Closes the input file.
80 END
```

A run of these programs produces the following output:

```
RUN SEG1                                  Run the first segment.
SEG1 IS WORKING                           Segment 1 executes and
SEG2 IS WORKING                           chains to segment 2.
THE TOTAL IS 10100                        Prints the total.

READY
```

Remember to save (see Section 9.6.1) a program containing a CHAIN statement before running it. Otherwise BASIC erases the program from your area in memory and you will not have a copy of it.

NOTE

It is faster to chain to programs which have been compiled. See Section 9.10 for a description of the COMPILE command.

8.1.1 Preserving Variables Through CHAIN (COMMON Statement)

The COMMON statement preserves data when one BASIC program chains to another. Any variables or arrays listed in COMMON statements retain the same variable names and values after CHAIN is executed.

COMMON allows one segment of a program to preserve data needed by another segment. The only alternative way for chained programs to preserve data is by file storage. However, preserving data in memory with a COMMON statement is much faster and simpler than first writing the data out to a file and then, after chaining, reading it back in. Consequently, when you want to preserve data through a CHAIN statement, use the COMMON statement.

Figure 8-2 shows how data is preserved in COMMON after the CHAIN statement.

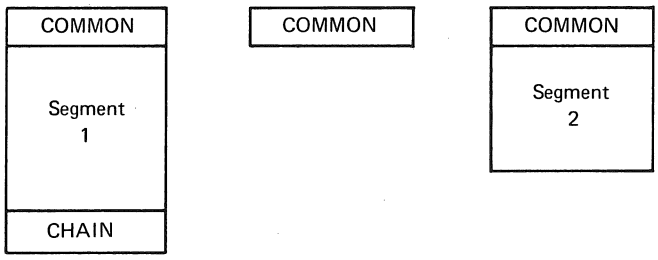


Figure 8-2 CHAIN with COMMON

At the left, segment 1 including the COMMON statements is in memory. When BASIC executes the CHAIN statement in segment 1, it erases all of segment 1 except the variables and arrays listed in COMMON statements, as illustrated in the center. Then, at the right, BASIC brings segment 2 into memory and merges it with the preserved variables and arrays.

The format of the COMMON statement is:

COMMON list

where:

list specifies the variables and arrays to be preserved after execution of the CHAIN statement. It is in the general form:

var[(int[,int])][,var[(int[,int])],...]

When BASIC executes the CHAIN statement all the variables and arrays listed in COMMON statements are preserved.

When BASIC brings in the new program segment, it checks to see that the new segment has corresponding COMMON statements. The lists in the COMMON statements of the new segments must contain the same variable names, data types, and array dimensions in the same order as the lists in the previous segment. You can change the line numbers and the number of items specified in each list but you cannot change the order of the variables and arrays.

Consider these examples:

Program 1

```
10 COMMON A,B,C$
20 COMMON D%(100)
30 COMMON G$(2)
```

Program 2

```
10 COMMON A,B
30 COMMON C$,D%(100),G$(2)
```

Program 3

```
10 COMMON A,B,D%(100)
20 COMMON C$
30 COMMON G$(2)
```

Programs 1 and 2 contain equivalent COMMON statements. However, program 3 does not contain equivalent COMMON statements because D%(100) appears before C$.

If in the new program you do not list the variables and arrays in COMMON statements as in the original, BASIC prints the ?COMMON OUT OF ORDER (?COO) error message and stops program execution.

Consider the following examples:

The file specified by "SEGC1" contains:

```
10 COMMON I(100)      Preserves the array I(100) (in COMMON).
20 DIM J(100)         Dimensions array J(100)  (not in COMMON).
30 PRINT "SEGC1"      Prints identifying message.
40 FOR K=1 TO 100     Assigns the values
50 I(K)=K*2           of the even numbers
60 J(K)=K*2           between 2 and 200 to
70 NEXT K             the elements of each array.
80 CHAIN "SEGC2"      Chains to the next segment.
100 END
```

The file specified by "SEGC2" contains:

```
10 COMMON I(100)      COMMON statement equivalent to original.
20 DIM J(100)         Dimensions new array J(100).
30 PRINT "SEGC2"      Prints identifying message.
40 FOR K=1 TO 100     Adds all the elements of
50 T1=T1+I(K)         I(100) and
60 T2=T2+J(K)         J(100) stored in T1
70 NEXT K             and T2, respectively.
80 PRINT "T1=";T1
90 PRINT "T2=";T2
100 END
```

A run of these programs produces the following output.

```
RUN SEGC1
SEGC1
SEGC2
T1= 10100
T2= 0

READY
```

Note that BASIC preserves I(100) but does not preserve J(100).

It is possible to extend COMMON by placing additional variables and arrays after the existing ones. For example,

Program Segment 1

```
10 COMMON A,B%(100)
20 COMMON C$(5)
```

Program Segment 2

```
10 COMMON A,B%(100)
20 COMMON C$(5),F9(100)
30 COMMON A$(30)
```

Program segment 2 contains the same variables and arrays in COMMON as program 1 but also extends program segment 1's COMMON statements with F9(100) and A$(30).

When program segment 2 chains, BASIC preserves all variables and arrays listed in the extended COMMON statements.

If a program containing COMMON statements chains to a program that has no COMMON statement, BASIC erases all the variables and arrays including those listed in the COMMON statements.

If a program segment containing COMMON statements chains to another program segment containing COMMON, all the variables and arrays in the original COMMON statements should appear in the new statements. But if the new COMMON statements contain some of the variables and arrays (in the correct order) but not all of them, BASIC does not produce an

error message. Instead BASIC preserves all the variables specified in the original COMMON statements. Even though no error message is produced, this situation should be avoided because variables are preserved that do not appear in the new program segment's COMMON statements.

BASIC allows you to list up to 255 variables and arrays in COMMON statements. If a program contains more than 255 variable and array names, BASIC prints the ?TOO MANY ITEMS IN COMMON (?TIC) error message.

BASIC automatically dimensions arrays specified in COMMON statements. If you list an array in a COMMON statement, do not also dimension it with a DIM statement. If an array is in both a COMMON and a DIM statement, BASIC prints the ?ILLEGAL DIMENSION (?IDM) error message and stops program execution.

You cannot specify virtual array files (see Section 6.4) in COMMON statements. An alternative is to pass the file specification in a string variable listed in COMMON and have the new program reopen the virtual array file.

BASIC erases any existing COMMON area when the user types a RUN, OLD or SCR command or when a new program segment has no COMMON statements.

The COMMON statement is meaningless in immediate mode.

NOTE

> The COMMON statement does not provide any communication with programs written in languages other than BASIC (see Section 8.3).

8.2 MERGING PROGRAM SEGMENTS (OVERLAY STATEMENT)

You can use the OVERLAY statement to segment programs as you use the CHAIN statement. The OVERLAY statement allows easier communication between segments than the CHAIN statements. When you use the OVERLAY statement the values of all variables and arrays are preserved and all open files remain open. However, you must ensure that the line numbers of the segments merge correctly, which you need not do with CHAIN.

The OVERLAY statement merges the current program with a program segment stored in a file.

When BASIC reads a line of the program segment from the file, it interweaves the line into the current program. If a line with the same line number already exists, BASIC deletes the existing line and replaces it with the line from the new program segment. During this process all variables and arrays retain their current values, and all open files remain open. When all lines of the program segment in the file are read into memory and merged with the original program lines, BASIC continues execution of the merged program.

NOTE

> The OVERLAY statement is not a standard feature in DIGITAL's BASICs. If you are concerned with transportability of programs and ease of upgrading, do not use the OVERLAY statement. Use only the CHAIN statement for segmenting programs.

To segment a program with the OVERLAY statement, break the program into one main program and several overlay segments.

The total memory required by a program segmented with the OVERLAY statement is the size of the main program plus the size of the largest overlay as illustrated in Figure 8-3.

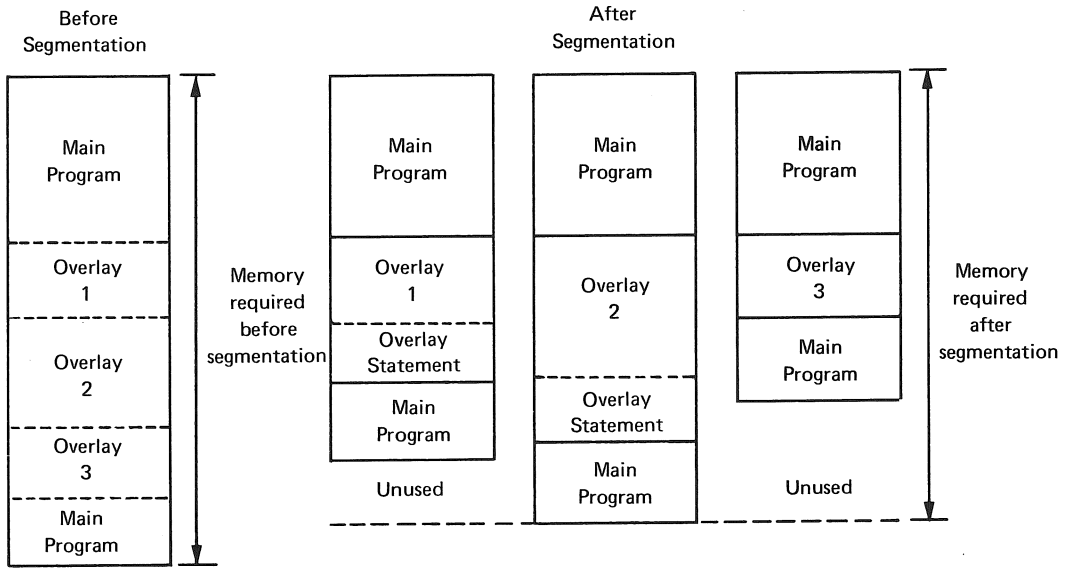


Figure 8-3 Segmenting a Program with the OVERLAY Statement

You must ensure that all line numbers in an overlay segment are repeated in each subsequent segment. Otherwise the parts of the previous segment will still be in memory.

The format of the OVERLAY statement is:

```
OVERLAY string [[LINE expression]]
```

where:

string	is a file specification and can be any string expression.
expression	specifies the line to start execution in the new program.

If you omit LINE and the expression, BASIC starts execution at the next sequential line number after the OVERLAY statement. Note that if you enter the OVERLAY statement on a multi-statement line, BASIC ignores all subsequent statements on the line. If you specify LINE and an expression but the value of the expression in not a valid line number, BASIC prints the ?SYNTAX ERROR (?SYN) error message and stops execution before reading the program segment. If you specify LINE and an expression but the line number specified does not exist in the merged program, BASIC prints the ?UNDEFINED LINE NUMBER (?ULN) error message and stops program execution after merging the new segment.

Consider the following examples:

The file specified by "MAINPR" contains:

```
5 DIM A(100)                                  Dimension array A(100)
10 PRINT "MAIN PROGRAM"                       Print message.
20 FOR I=1 TO 100                             Fill array A with
30 A(I)=I*3                                   every third number
40 NEXT I                                     from 3 to 300.
50 PRINT "STILL IN MAIN PROGRAM"              Print message.
100 PRINT "SEGMENT 1"                         Segment 1 prints message.
110 FOR I=1 TO 100
120 IF A(I)/2<>INT(A(I)/2) THEN 140           Skip all odd values of A
130 T=T+A(I)                                  Add the rest to T.
140 NEXT I
150 PRINT "THE SUM IS";T                      Print result.
170 OVERLAY "OVL"                             Overlay to second segment.
200 GO TO 50                                  End of main
300 END                                       program.
```

The file specified by "OVL" contains:

```
100 PRINT "SEGMENT 2"                         Print message.
110 T=0                                       Reinitialize T.
120 FOR I=1 TO 100                            Add all numbers
130 T=T+A(I)                                  in array.
140 NEXT I
150 PRINT "THE SUM IS";T                      Print result.
160 GO TO 300                                 Go to end.
```

A run of these programs produces:

```
RUNNH MAINPR
MAIN PROGRAM
STILL IN MAIN PROGRAM
SEGMENT 1
THE SUM IS 7650
STILL IN MAIN PROGRAM
SEGMENT 2
THE SUM IS 15150

READY
```

After BASIC executes the OVERLAY statement, the program in memory is:

```
5 DIM A(100)
10 PRINT "MAIN PROGRAM"
20 FOR I=1 TO 100
30 A(I)=I*3
40 NEXT I
50 PRINT "STILL IN MAIN PROGRAM"
100 PRINT "SEGMENT 2"
110 T=0
120 FOR I=1 TO 100
130 T=T+A(I)
140 NEXT I
150 PRINT "THE SUM IS";T
160 GO TO 300
170 OVERLAY "OVL"
200 GO TO 50
300 END
```

BASIC continues execution at the next statement after the OVERLAY statement, which is line 200.

Any user-defined function defined in the main program remains defined after the OVERLAY statement is executed if the line defining function is not replaced by a line in the overlay segment.

The overlay segment in the file should not contain any DIM or DEF statements. BASIC ignores these statements when reading in an overlay segment.

NOTE

Do not specify a compiled file with the OVERLAY statement. BASIC will not be able to merge the programs and will produce a ?LINE TOO LONG (?LTL) or ?TOO LONG TO TRANSLATE (?TLT) error message.

8.3 CALLING A ROUTINE WRITTEN IN ANOTHER LANGUAGE (CALL STATEMENT)

The CALL statement causes BASIC to execute a routine written in another language. The routine, usually written in assembly language, must have been incorporated previously into the BASIC language itself. The procedure for incorporating the routines into BASIC and the interface for the routines are described in your BASIC-11 user's guide.

The format of the CALL statement is:

CALL string[[(list)]]

where:

string — specifies the name of the routine. The string can only be a string constant.

list — is the optional argument list. It can contain constants, variables, expressions, and arrays. An array is listed as a variable name followed by a left parenthesis and a right parenthesis. For example; A%() specifies the integer array A which can have one or two dimensions.

Consider these examples:

```
10 CALL "AND"(A%,B%,C%)
```

Calls the routine AND. The variable list contains three integer variables.

```
20 CALL "ZERO"(A%())
```

Calls the routine ZERO. The argument list contains an integer array.

```
30 CALL "REVERS"(A$,D$,C(9))
```

Calls the routine REVERS. The argument list contains the string variables A$ and D$ and the array C beginning with the ninth element.

BASIC automatically allocates room for an undimensioned array when it first encounters a reference to it, but BASIC cannot do this when the

reference to the array first occurs in a CALL statement. In this case, BASIC prints the ?UNDIMENSIONED ARRAY IN CALL (?UAC) error message.

BASIC cannot return a result from a routine to an element of a virtual array file. If the routine returns a result in an argument, do not specify a virtual array element in that position. It is not possible to pass an entire virtual array to a routine for any reason.

CHAPTER 9

BASIC-11 COMMANDS

9.1 KEY COMMANDS

BASIC has a set of key commands that delete characters and lines being typed, stop printout from programs, and interrupt execution of BASIC programs. Table 9-1 describes the key commands.

NOTE

A CTRL (control) key command is typed by pressing the CTRL key while typing the appropriate letter. For example, to type CTRL/C, press the CTRL key and type C.

Table 9-1
BASIC-11 Key Commands

Key	Explanation
CTRL/C	Interrupts execution of a command or program. This is the key to type when your program is running and you want to stop it. BASIC prints a STOP message. CTRL/C also cancels the effect of CTRL/O and CTRL/S. Be sure to see your BASIC-11 user's guide for more information on using CTRL/C on your system.
CTRL/O	Stops output to the terminal but does not interrupt execution of the program. Printing on the terminal resumes after BASIC executes an INPUT statement, the program ends, or after you type another CTRL/O. Typing CTRL/O cancels any message that BASIC is printing. CTRL/O is different than CTRL/S. After you type CTRL/O, BASIC continues executing the program but you will not see anything printed on the terminal. After you type CTRL/S, BASIC waits for you to type CTRL/Q. When you have typed CTRL/Q, BASIC continues to print on the terminal; no information is lost.
CTRL/S	Temporarily interrupts the printing on the terminal. Printing resumes after CTRL/Q or CTRL/C is typed. (See description of CTRL/O.)
CTRL/Q	Continues the printing on the terminal after a CTRL/S has interrupted it.

Table 9-1 (Cont.)
BASIC-11 Key Commands

Key	Explanation
CTRL/U	Deletes the entire line being typed. If you notice a mistake on the line you are typing before you type the RETURN key, type CTRL/U and BASIC ignores the line. CTRL/U can be used when typing in program lines or when responding to the INPUT or LINPUT statement.
RUBOUT	Deletes the last character typed. If you mean to type: 10 FOR I=1 TO 3 but instead type: 10 FOR I=3 TO press the RUBOUT key four times to delete the incorrect characters: 10 FOR I=3 TO ↑↑↑↑ (arrows indicate deleted characters) and then complete the line by typing: 1 TO 3

9.2 LISTING YOUR PROGRAM (LIST AND LISTNH COMMANDS)

Use the LIST command to print the program lines you have entered. To list your entire program, type:

```
LIST
```

BASIC first prints a header line which consists of the program name, date, and the BASIC version number. BASIC then prints your entire program. After BASIC prints your program, it prints the READY message.

You can also list sections of the program, individual lines, groups of lines, or any combination of these. The format of the LIST command is:

```
LIST[line specification, line specification, ...]
```

where each line specification can be:

line number	Lists specified line.
line number-line number	Lists all lines in range specified.
line number-	Lists all line numbers from the specified line to the end.
-line number	Lists all the line numbers from the beginning to the specified line number.

The LISTNH command is the same as the LIST command except that BASIC does not print the header line.

Consider these examples:

LISTNH	Lists the entire program without a header line.
LIST 25	Lists a header line and line 25.
LIST 25,50,100-200,500-	Lists lines 25 and 50, all lines from 100 through 200, and all lines from 500 through the end of the program. Remember a comma is equivalent to "and" and a dash is equivalent to "through".

NOTE

To list a program to a file or to a line printer, use the SAVE command (see Section 9.6.1).

9.3 EXECUTING A PROGRAM (RUN AND RUNNH COMMANDS)

After you have entered your program, you can execute it by typing:

```
RUN
```

When BASIC executes the RUN command, it first prints a header line. Then it scans the program, reserving space in memory for all arrays in DIM or COMMON statements, defining any user-defined functions in DEF statements, and initializing all numeric variables and arrays to 0 and all strings to the null string. Finally BASIC starts executing the program at the lowest numbered line.

The RUNNH command has the same effect as the RUN command except that BASIC does not print the header line.

See Section 9.6.3 for a description of executing a program that is stored in a file.

9.4 DELETING PROGRAM LINES (DEL COMMAND)

In addition to deleting a line by typing the line number followed by the RETURN key, you can delete lines with the DEL command. The format of the DEL command is:

```
DEL line specification[,line specification,...]
```

where the line specification can take the same form that it does in the LIST command (see Section 9.2).

For example:

DEL 10	Deletes line 10.
DEL 100,500	Deletes lines 100 and 500.
DEL 25,100-150,75,275-300	Deletes lines 25 and 75 and all the lines between 100 and 150, inclusively, and between 275 and 300, inclusively. Note that the lines do not have to be specified in order.

9.5 ERASING THE PROGRAM (NEW, SCR, AND CLEAR COMMANDS)

The NEW command erases your program storage area in memory. It deletes all program lines, variables, arrays, and definitions of user-defined functions. It also changes the program name to the one specified.

The format of the NEW command is:

```
NEW program name
```

where program name can contain the letters A through Z and the digits 0 through 9.

If you type:

```
NEW
```

without a program name, BASIC requests the new name by printing:

```
NEW FILE NAME--
```

If you type a name, BASIC uses it for the new program name. However, if you type the RETURN key, BASIC changes the name to the default program name, NONAME.

After BASIC erases the program and changes the program name, it prints the READY message.

For example:

```
NEW MONEY
```

Erases all your program information and changes the program name to MONEY.

Be sure to type the NEW command when you start writing a program. It erases any existing program lines.

The SCR or scratch command does the same thing as the NEW command except that it always changes the program name to NONAME. Use the SCR command instead of the NEW command when you do not care what the program name is. The format of the SCR command is:

```
SCR
```

BASIC prints READY when it is finished erasing the program.

The CLEAR command erases the contents of all variables and arrays but does not erase any program lines. The CLEAR command returns the space used by arrays so that it can be used for program lines. The CLEAR command does not change the program name.

The format of the CLEAR command is:

```
CLEAR
```

BASIC prints the READY message when it is done.

9.6 PROGRAMS IN FILES

Up to this point, all the commands have manipulated the programs in your area of memory (see Section 1.7), but this section describes how to transfer programs between your area of memory and a file. You can store in a file the program that is in your area, and you can transfer a program from a file to your area.

Programs stored in files are in the same format as sequential data files but you use different statements and commands to access program files and sequential data files (see Chapter 6 for a description of data file statements and see your BASIC-11 user's guide for more information on differences between program and data files). The following list describes the statements which access program files.

Statement	Transfers Program
CHAIN	From a file to your area and executes it.
OVERLAY	From a file to your area and merges it with your current program.

The following list describes the commands which access program files.

Command	Transfers Program
SAVE	From your area to a file.
REPLACE	From your area to a file, deleting an existing file, if necessary.
OLD	From a file to your area.
APPEND	From a file to your area and merges it with your current program.
RUN	From a file to your area and executes it.

In addition, you can use the UNSAVE command to delete a program file.

The COMPILE command transfers a program in a special format from your area to a file (see Section 9.10).

9.6.1 Saving the Program in a File (SAVE and REPLACE Commands)

The SAVE command saves the BASIC program that is currently in your area in memory. BASIC transfers the program from your area to the file you specify. BASIC writes to the file the same ASCII characters that it would print on the terminal if you typed LISTNH. The format of the SAVE command is:

SAVE [[file specification]]

If you omit the file specification, BASIC uses the current program name as a default specification.

After BASIC saves the file, it prints the READY message. Once you save a file, you can read it into memory by typing the OLD or APPEND command (see Section 9.6.2). Alternatively, you can run it from the file by typing the RUN file specification command (see Section 9.6.3) or by having BASIC execute the CHAIN or OVERLAY statement (see Sections 8.1 and 8.2).

For example:

SAVE PROG1	Stores the current program in the file specified by PROG1.
SAVE LP:	Stores (lists) the current program on the line printer.

The SAVE command does not delete an existing file with the same file specification. If a file with the same specification exists and if creating a new file would delete the original one, BASIC prints the ?USE REPLACE (?RPL) error message. To save the program, you should either use a different file specification or use the REPLACE command.

The REPLACE command is like the SAVE command except that REPLACE saves a program even if it means deleting an existing file. This difference between the SAVE and REPLACE commands helps to prevent you from inadvertently deleting program files that you have previously saved.

The format of the REPLACE command is:

REPLACE [file specification]

If you omit the file specification, BASIC uses the current program name as the default specification.

For example:

```
REPLACE PROG1
```

Saves the current program in the file specified by PROG1, deleting any existing file with the same file specification.

9.6.2 Restoring a Program from a File (OLD and APPEND Commands)

The OLD command first erases your area in memory and changes the program name (like the NEW command) and then reads in the program from the specified file.

The format of the OLD command is:

OLD [file specification]

If you type only:

```
OLD
```

BASIC requests the file specification by printing:

```
OLD FILE NAME--
```

Then type the file specification. If you type only the RETURN key, BASIC assumes the file specification NONAME (BASIC looks for a program stored in the file specified by NONAME).

For example:

```
OLD PROG1
```

Erases the program currently in memory and reads in the program in the file specified by PROG1.

NOTE

If BASIC cannot find the file you specify in an OLD command, it changes the program name and prints the ?FILE NOT FOUND (?FNF) error message. However, BASIC does not delete the program in your area.

The APPEND command merges the program currently in memory with the program in the specified file. BASIC reads in the specified file and sorts the new program lines into the current program. If one of the new lines has the same line number as an existing line, BASIC deletes the existing line.

The format of the APPEND command is:

```
APPEND [file specification]
```

If you type only:

```
APPEND
```

BASIC requests the file specification by printing:

```
OLD FILE NAME--
```

You then type the file specification. If you just type the RETURN key, BASIC assumes the file specification NONAME.

Entering the APPEND command has the same effect as entering the new lines at the terminal and also as entering an immediate mode OVERLAY statement.

9.6.3 Running a Program from a File

If you want to run a program from a file, specify a file after the RUN command.

The format of the command is:

```
RUN file specification
```

When you specify a file with the RUN command, BASIC erases all program lines and the contents of all variables, changes the program name (equivalent to the NEW command, see Section 9.5) reads in the specified program (equivalent to the OLD command, see Section 9.5) and finally starts the execution of the program.

The RUN file specification command does not print a header line and is equivalent to the RUNNH file specification command.

See Section 9.3 for a description of the RUN and RUNNH commands without a file specification.

For example:

```
RUN SPCWR
```

Erases the program currently in memory, changes the program name to SPCWR, reads in the file specified by SPCWR, and starts execution of the program.

NOTE

If BASIC cannot find the file specified in a RUN command, it changes the program name and prints the ?FILE NOT FOUND (?FNF) error message. However, BASIC does not delete the program in your area. This may happen if you mistype the RUNNH command, such as RUNHN: BASIC interprets this as a RUN HN command where HN is the file specification.

If the file specification begins with NH, BASIC will assume that the NH is part of the RUNNH command. For example, BASIC interprets a RUN NHT command as RUNNH T. To run a program whose file specification begins with NH, use the RUNNH command.

9.6.4 Deleting a Program File (UNSAVE Command)

The UNSAVE command deletes the specified program file. The format of the UNSAVE command is:

```
UNSAVE file specification
```

BASIC deletes the file specified and then prints the READY message. When the file is deleted, it cannot be restored.

For example:

```
UNSAVE MONEY
```

Deletes the program file specified by MONEY.

The UNSAVE command has the same effect as the KILL statement except that the KILL statement deletes data files instead of program files.

9.7 CHANGING THE PROGRAM NAME (RENAME COMMAND)

The RENAME command changes the program name to the one specified but, unlike the NEW command, does not erase the program.

The format of the RENAME command is:

```
RENAME [program name]
```

If you just type:

```
RENAME
```

BASIC requests the new name by printing:

```
NEW FILE NAME--
```

You then type the new program name. If you type only the RETURN key, BASIC changes the program name to NONAME.

9.8 EDITING A LINE (SUB COMMAND)

You can change a program line that you have already typed without retyping the whole line by using the SUB command. The SUB command substitutes the specified characters on the specified line with new characters.

The format of the SUB command is:

SUB line number xstring1xstring2[x[integer]]

where

line number	specifies the line to be edited.
x	can be any character to delimit the strings. X must not appear in the strings. The character @ is a good choice for a delimiting character because it is not used in BASIC programs except in string constants.
string1	is the old series of characters to be deleted. Do not delimit the string with quotation marks.
string2	is the new series of characters to be inserted. Do not delimit the string with quotation marks.
integer	specifies the occurrence of string1 in the line. If the integer is omitted, the first occurrence is substituted. You do not have to type the percent sign after the integer.

After the changes are made, the new line is printed.

For example, if the current line 10 is:

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

READY
```

then type the SUB command:

```
SUB 10 @A=@A+@
10 LET A=B*C

READY
```

BASIC makes the correction to the line and then prints out the new line.

If the current line is

```
LISTNH 100
100 F9=A*SIN(X)+F9

READY
```

and you want to change the second occurrence of F9 to I%, type the SUB command:

```
SUB 100@F9@I%@2
100 F9=A*SIN(X)+I%

READY
```

SUB can be used to change the line number. For example if the current line 20 is:

```
LISTNH 20
20 PRINT A,B,C,D

READY
```

and you wish to change the line number to 100, type the following SUB command:

```
SUB 20 @20@100@
100 PRINT A,B,C,D

READY
```

BASIC makes the correction and then prints the new line. Note that BASIC does not delete the old line but merely copies it.

However, you cannot use the SUB command to delete the entire line number (change a program line to an immediate mode statement). If you try to do this, BASIC prints the ?SUBSTITUTE ERROR (?SUB) message and does not execute the command. This message is also produced if you enter the SUB command in the wrong format (such as omitting the delimiting character ending string1).

BASIC uses the first character after the line number on the SUB command as the delimiting character, but it ignores all spaces and tabs until it finds a character. Consequently you cannot use space nor tab as a delimiting character. Neither can you use a digit as the delimiting character, because BASIC will consider it part of the line number.

If BASIC cannot find the old string you specify, it reprints the line with no changes.

NOTE

You must type string1 exactly as BASIC lists it; consequently, it is useful to list a line before entering the SUB command.

9.9 RESEQUENCING A PROGRAM (RESEQ COMMAND)

You can use the RESEQ command to resequence your program or sections of your program. If you have inserted so many lines in your program that your line numbers are incremented by 1 and you need to insert a new line, you should resequence the program.

When BASIC resequences a program, it changes the line numbers specified and it also changes any references to the changed line numbers (i.e. in GO TO, ON GO TO, IF THEN, IF GO TO, GOSUB, and ON GOSUB statements).

You can use the RESEQ command to resequence your entire program or a segment of the program. You specify the new line number at which you want the segment (or the entire program) to start, the range of line numbers that you want resequenced (the old line numbers), and the increment to be used between each line number. You do not have to specify all the values: BASIC uses a default value when you omit one.

The format of the RESEQ command is:

RESEQ [new],[old1][-old2] [,increment]

where

new	specifies the starting new line number.
old1	specifies the lowest existing line number to be resequenced.
old2	specifies the highest existing line number to be resequenced.
increment	specifies the increment to be used between each line.

If you do not specify new, the new starting line number, BASIC uses the highest existing line number less than old1.

For example if you type

```
RESEQ,105-200,10
```

and the highest line number under 105 is 100, BASIC starts the first new line at 100 plus the increment 10, or 110.

If you do not specify the old1, BASIC starts resequencing at the beginning of the program.

If you do not specify old2, BASIC resequences the program until its end.

If you do not specify the increment, BASIC uses an increment of 10.

If you type only:

```
RESEQ
```

BASIC resequences the entire program, giving the first line the line number 10 and incrementing each line by 10.

For example, if your current program is:

```
LISTNH
10 PRINT "CHECK BALANCING PROGRAM"
13 PRINT "LAST BALANCE";
14 INPUT B
15 PRINT "NEXT CHECK";
20 INPUT C
21 IF C=0 THEN 135
25 B=B-C
30 GO TO 15
135 PRINT "NEXT DEPOSIT";
140 INPUT D
141 IF D=0 THEN 1000
145 B=B+D
150 GO TO 135
1000 PRINT "SERVICE CHARGE";
1001 INPUT S
1002 B=B-S
1005 PRINT "BALANCE IS";B
1100 END

READY
```

The command

```
RESEQ 200,135-150,20
```

resequences lines 135, 140, 141, 145, and 150 to line numbers 200, 220, 240, 260, and 280 respectively. Instead of line 21 being IF C=0 THEN 135, it is changed to IF C=0 THEN 200.

The program now is:

```
LISTNH
10 PRINT "CHECK BALANCING PROGRAM"
13 PRINT "LAST BALANCE";
14 INPUT B
15 PRINT "NEXT CHECK";
20 INPUT C
21 IF C=0 THEN 200
25 B=B-C
30 GO TO 15
200 PRINT "NEXT DEPOSIT";
220 INPUT D
240 IF D=0 THEN 1000
260 B=B+D
280 GO TO 200
1000 PRINT "SERVICE CHARGE";
1001 INPUT S
1002 B=B-S
1005 PRINT "BALANCE IS";B
1010 END

READY
```

Note that the old line 150 (now is line 280) is GO TO 200 instead of the original GO TO 135.

If you type:

```
RESEQ 100,,50
```

BASIC resequences the entire program with the new starting line of 100 and an increment of 50.

If you specify a combination of line numbers that would change the order of lines, BASIC prints the ?RESEQUENCE ARGUMENT ERROR (?RES) message and ignores the command. For example, if in the original example in this section you type:

```
RES 100,10-30
```

BASIC would try to resequence lines 10, 13, 14, 15, 20, 21, 25, and 30 as lines 100, 110, 120, 130, 140, 150, 160, and 170, respectively. But because there is already a line 135, this would change the order of the program lines. Consequently, BASIC would print the error message.

9.10 SAVING A COMPILED PROGRAM (COMPILE COMMAND)

BASIC does not store the program exactly the way you type it but instead compresses or compiles each line. This allows you to fit larger programs in your area than you could if BASIC did not compile each line. Whenever you list or save your program, BASIC actually translates the program from the form in which it is stored to the form that you entered.

The COMPILE command saves a copy of the internal image that BASIC uses to store programs. Once you have saved this image, BASIC can read it into memory much faster than it can read a file saved by the SAVE command.

The format of the COMPILE command is:

```
COMPILE [file specification]
```

If you omit the file specification, BASIC uses the current program name as the default file specification.

The compiled file can be read in by the OLD or RUN command or by the CHAIN statement, but it cannot be read by the APPEND command or the OVERLAY statement.

NOTE

> The file created by the COMPILE command should only be used on the same system on which it was created. If you want to use a compiled file on another version of BASIC, you must restore the compiled file with an OLD command on the system that it was created and then store a new file with the SAVE command. The SAVE command creates files that can be used by other versions of BASIC.

9.11 CHECKING THE LENGTH OF A PROGRAM (LENGTH COMMAND)

Use the LENGTH command to find the amount of storage required by the BASIC program in memory. This information is useful in determining the minimum area in which a specific program can run. The form of the command is:

```
LENGTH
```

The LENGTH command causes BASIC to produce this message:

```
mmm WORDS USED, nnn FREE
```

where:

mmm	is the number of words currently occupied by your program.
nnn	is the number of words remaining free in your area in memory.

The number of words in use includes memory currently needed by the BASIC program itself, arrays, string variables, and file buffers. To determine the size of the program alone, enter the LENGTH command immediately after an OLD or CLEAR command. Arrays are created after the RUN command, and file buffers are created when the OPEN statement is executed.

The memory required for string variables and string arrays varies with the current values of the strings; consequently, the LENGTH command returns the current memory requirements, which may be smaller than the maximum memory requirements.

BASIC prints several error messages when the program exceeds the amount of memory available: ?ARRAY TOO LARGE (?ATL), ?BUFFER STORAGE OVERFLOW (?BSO), ?PROGRAM TOO BIG (?PTB), and ?STRING STORAGE OVERFLOW (?SSO). To reduce program size, follow one or more of the following procedures:

1. Eliminate or reduce unnecessary items such as REM statements, long printed messages, and optional keywords such as LET.

2. Make maximum use of multiple statement lines.

3. Make efficient use of program loops, subroutines, user-defined functions, and multiple branching.

4. Split up large programs into several smaller programs by using the CHAIN statement.

5. Reduce the size of arrays in memory to the size required (DIM statement).

6. Reduce the number of variables and arrays in a program by reusing them when their contents are no longer needed, instead of creating new variables or arrays.

7. Reduce the number of simultaneously open files by opening a file just before it is needed and closing it immediately after the last use.

8. Substitute virtual array files for large arrays in memory.

After deleting program lines, store the program with the SAVE command and restore it with the OLD command to further minimize program memory requirements.

APPENDIX A

ERROR MESSAGES

When BASIC finds an error in a program line, immediate mode statement, or command, it prints an error message. This signals that an error has occurred and that you should correct the error. Error messages are an aid in debugging your program (see Section 1.9).

When BASIC detects an error in a program line it prints the error message in the format:

```
?message AT LINE xxxxx
```

where xxxxx is the line number of the statement containing the error.

When BASIC detects an error in an immediate mode statement or command, it prints the error message in the format:

```
?message
```

Most of the error messages are fatal error messages; that is, after BASIC prints the error message, it interrupts execution of the program line, immediate mode statement or command and then prints the READY message.

Certain arithmetic and input errors are nonfatal. When BASIC finds a nonfatal arithmetic error, it substitutes a default value for the operation causing the error and continues execution. When BASIC finds a nonfatal input error, it prints a message and continues execution. After certain nonfatal input errors, BASIC requests more input.

BASIC detects most errors when it is executing a program line, immediate mode statement, or command. BASIC does, however, detect these errors when you are typing in your program.

```
?LINE TOO LONG (?LTL)
?LINE TOO LONG TO TRANSLATE (?TLT)
?PROGRAM TOO BIG (?PTB)
```

Some versions of BASIC use an abbreviated 3-letter error message. These messages and their corresponding long forms are listed in Table A-1. See your BASIC-11 user's guide for a description of the error messages your system prints.

Table A-2 lists the long form of the error messages and provides a description of each. (The abbreviated forms are listed in parentheses.) You can assume an error message is fatal unless the first word of the description is "nonfatal."

Table A-1
Abbreviated Error Messages

Abbreviated Form	Long Form
?ARG	?ARGUMENT ERROR
?ATL	?ARRAY TOO LARGE
?BDR	?BAD DATA READ
?BLG	?BAD LOG
?BRT	?BAD DATA - RETYPE FROM ERROR
?BSO	?BUFFER STORAGE OVERFLOW
?CAO	?CHANNEL ALREADY OPEN
?CCP	?CHECKSUM ERROR IN COMPILE PROGRAM
?CDF	?CANNOT DELETE FILE
?CIE	?CHANNEL I/O ERROR
?CNO	?CHANNEL NOT OPEN
?COO	?COMMON OUT OF ORDER
?CVO	?CONTROL VARIABLE OUT OF RANGE
?DV0	?DIVISION BY ZERO
?ECC	?ERROR CLOSING CHANNEL
?EER	?EXPONENTATION ERROR
?EII	?EXCESS INPUT IGNORED
?ENL	?END NOT LAST
?ETC	?EXPRESSION TOO COMPLEX
?FAD	?FUNCTION ALREADY DEFINED
?FAE	?FILE ALREADY EXISTS
?FNF	?FILE NOT FOUND
?FOV	?FLOATING OVERFLOW
?FPV	?FILE PRIVILEGE VIOLATION
?FSV	?NESTED FOR STATEMENTS WITH SAME CONTROL VARIABLE
?FUN	?FLOATING UNDERFLOW
?FWN	?FOR WITHOUT NEXT
?ICN	?ILLEGAL CHANNEL NUMBER
?IDF	?ILLEGAL DEF
?IDM	?ILLEGAL DIM
?IDT	?ILLEGAL DATA TYPE
?IEF	?ILLEGAL END OF FILE IN COMPILE PROGRAM
?IFL	?ILLEGAL FILE LENGTH
?IFS	?ILLEGAL FILE SPECIFICATION
?IID	?ILLEGAL I/O DIRECTION
?IIM	?ILLEGAL IN IMMEDIATE MODE
?INS	?INCONSISTENT NUMBER OF SUBSCRIPTS
?IOV	?INTEGER OVERFLOW
?IRS	?ILLEGAL RECORD SIZE
?ISE	?INPUT STRING ERROR
?ISL	?ILLEGAL STRING LENGTH
?LTL	?LINE TOO LONG
?MSP	?MISSING SUBPROGRAM
?NER	?NOT ENOUGH ROOM
?NGS	?NEGATIVE SQUARE ROOT
?NSM	?NUMBERS AND STRING MIXED
?NVD	?NOT A VALID DEVICE
?NWF	?NEXT WITHOUT FOR
?OOD	?OUT OF DATA
?PRU	?PRINT USING ERROR
?PTB	?PROGRAM TOO BIG
?RES	?RESEQ ARGUMENT ERROR
?RPL	?USE REPLACE
?RWG	?RETURN WITHOUT GOSUB

Table A-1 (Cont.)
Abbreviated Error Messages

Abbreviated Form	Long Form
?SOB	?SUBSCRIPT OUT OF BOUNDS
?SSO	?STRING STORAGE OVERFLOW
?STL	?STRING TOO LONG
?SUB	?SUBSTITUTE ERROR
?SYN	?SYNTAX ERROR
?TIC	?TOO MANY ITEMS IN COMMON
?TLT	?LINE TOO LONG TO TRANSLATE
?TMG	?TOO MANY GOSUB'S
?UAC	?UNDIMENSIONED ARRAY IN CALL
?UFN	?UNDEFINED FUNCTION
?ULN	?UNDEFINED LINE NUMBER
?VCU	?VIRTUAL ARRAY CHANNEL ALREADY IN USE

Table A-2
BASIC-11 Error Messages

?ARGUMENT ERROR (?ARG)

Arguments in a function do not match, in number, range, or type, the arguments defined for the function. Ensure that there are the correct number of arguments, that their values are in the correct range, and that they are the correct type.

?ARRAYS TOO LARGE (?ATL)

Not enough memory is available for the arrays specified in the DIM statements. Reduce the size of the arrays or reduce the size of the program (see Section 9.11).

?BAD DATA READ (?BDR)

Data item input from a DATA statement or from a file is the wrong data type. Ensure that the DATA statement or the file contains the same data type as specified in the READ or INPUT # statement.

?BAD DATA - RETYPE FROM ERROR (?BRT)

Nonfatal. Item entered in response to an INPUT or INPUT #0 statement is the wrong data type. Retype item and program will continue.

?BAD LOG (?BLG)

Nonfatal. Expression in LOG or LOG10 function is 0 or negative. The function returns 0 and BASIC continues execution of the program.

?BUFFER STORAGE OVERFLOW (?BSO)

Not enough room available for file buffer in your area. Reduce program size (see Section 9.11).

Table A-2 (Cont.)
BASIC-11 Error Messages

?CANNOT DELETE FILE (?CDF)

The file specified in a KILL statement or UNSAVE command cannot be deleted. See your BASIC-11 user's guide for more information.

?CHANNEL ALREADY OPEN (?CAO)

OPEN statement specifies a channel that is already associated with an open file. Ensure that OPEN statements specify correct channel numbers and that files that should be closed are closed.

?CHANNEL I/O ERROR (?CIE)

Accessing data in a file produces an error. Ensure that your peripheral devices and their storage media are working correctly. One possible cause is that the file accessed has 0 length.

?CHANNEL NOT OPEN (?CNO)

A PRINT #, PRINT # USING, INPUT #, IF END #, or CLOSE statement, or a reference to a virtual array file specifies a channel number not associated with an open file. Check that the OPEN statement has been executed and that it specifies the same channel number as the program line with the error.

?CHECKSUM ERROR IN COMPILED PROGRAM (?CCP)

File produced by the COMPILE command contains a format error. Use a copy of the program created by a SAVE or REPLACE command.

?COMMON OUT OF ORDER (?COO)

Variables and arrays in a COMMON statement are not listed in the same order as those in a previous segment. Ensure that all segments have equivalent COMMON statements.

?CONTROL VARIABLE OUT OF RANGE (?CVO)

Expression in an ON GOTO or ON GOSUB statement is 0 or negative or has a value greater than the number of line numbers listed. Ensure that expression has a value in the correct range.

?DIVISION BY ZERO (?DV0)

Nonfatal. An expression includes a division by 0. BASIC substitutes a value of 0 for that operation and continues execution of the program.

?END NOT LAST (?ENL)

END statement is not the highest numbered program line. This error message is printed when the END statement is executed. Ensure that there is only one END statement in program and that it has the highest line number.

?ERROR CLOSING CHANNEL (?ECC)

Closing a channel produces an error. Ensure that your peripheral devices and their storage media are working correctly.

Table A-2 (Cont.)
BASIC-11 Error Messages

?EXCESS INPUT IGNORED (?EII)

Nonfatal. There are more data items than required by an INPUT or INPUT #0 statement. BASIC ignores the excess items and continues execution of the program. Ensure that data items did not contain an unintended comma (e.g., 1,430 instead of 1.430).

?EXPONENTIATION ERROR (?ERR)

Nonfatal. An expression includes the operation of raising a negative value to a nonintegral power (e.g., (-1)^.5). This would produce a complex number, which can not be represented in BASIC. This message is also produced when a negative value is raised to an integral value that has an absolute value greater than 255 (e.g. (-1)^256). In both cases, BASIC substitutes a value of 0 for the operation and continues execution.

?EXPRESSION TOO COMPLEX (?ETC)

An expression is too complex for BASIC to evaluate in the area it uses for calculations (called the stack). This condition is usually caused by including user-defined functions or nested functions in an expression. The degree of complexity that causes this error varies according to the amount of space available in the stack at the time. Breaking the statement up into several statements containing simpler expressions may eliminate the error.

?FILE ALREADY EXISTS (?FAE)

Creating a file would cause an existing file to be deleted and this deletion is not allowed. See your BASIC-11 user's guide for more information.

?FILE NOT FOUND (?FNF)

BASIC cannot find the specified file. Ensure that the file specification was typed correctly and that the file exists.

?FILE PRIVILEGE VIOLATION (?FPV)

This operation includes a restricted file operation. See your BASIC-11 user's guide for more information.

?FLOATING OVERFLOW (?FOV)

Nonfatal. The absolute value of the result of a computation is greater than the largest number that can be stored by BASIC (approximately 10^38). BASIC substitutes a value of 0 for the operation and continues execution of the program.

?FLOATING UNDERFLOW (?FUN)

Nonfatal. The absolute value of the result of a computation is smaller than the smallest number that BASIC can store (approximately 10^(-38)). BASIC substitutes a value of 0 for operation and continues execution of the program.

Table A-2 (Cont.)
BASIC-11 Error Messages

?FOR WITHOUT NEXT (?FWN)

The program contains a FOR statement without a corresponding NEXT statement to terminate the loop. Ensure that each loop in the program is terminated with a NEXT statement.

?FUNCTION ALREADY DEFINED (?FAD)

The user-defined function is previously defined. Ensure that each function is defined only once and has a unique name.

?ILLEGAL CHANNEL NUMBER (?ICN)

The channel specified is not in the range allowed or the IF END statement specifies a file on a terminal. See your BASIC-11 user's guide for information about the range of valid channel numbers.

?ILLEGAL DEF (?IDF)

There is an error in the DEF statement. Check the format and data types in the argument list and defining expression.

?ILLEGAL DIM (?IDM)

A subscript in a DIM or COMMON statement is not an integer, an array is dimensioned more than once, or an array has more than two dimensions. Ensure that an array specification is in the correct format and appears only once in the COMMON and DIM statements in the program.

?ILLEGAL END OF FILE IN COMPILED PROGRAM (?IEF)

File produced by the COMPILE command contains a format error. Use a copy of the program created by a SAVE or REPLACE command.

?ILLEGAL FILE LENGTH (?IFL)

The FILE LENGTH specified in an OPEN statement is invalid. See your BASIC-11 user's guide for information on the valid range of FILE LENGTH.

?ILLEGAL FILE SPECIFICATION (?IFS)

The file specification is invalid. See your BASIC-11 user's guide for information on the format of a file specification.

?ILLEGAL IN IMMEDIATE MODE (?IIM)

The INPUT or INPUT # statement cannot be entered in immediate mode. Enter the statement in a program line (followed with a STOP statement) and execute the statement with an immediate mode GO TO statement.

Table A-2 (Cont.)
BASIC-11 Error Messages

?ILLEGAL I/O DIRECTION (?IID)

Statement attempts to write to an input file or read an output file. Ensure that the channel number specified specifies the correct file. If the statement assigns a value to an element of a virtual array file, ensure that the file's OPEN statement does not specify "FOR INPUT."

?ILLEGAL RECORD SIZE (?IRS)

The RECORDSIZE specified in an OPEN statement is invalid. See your BASIC-11 user's guide for information on the valid range for RECORDSIZE.

?INCONSISTENT NUMBER OF SUBSCRIPTS (?INS)

The array is dimensioned with one subscript and referenced by two, or vice versa. Ensure that the DIM statement and array references are consistent.

?INPUT STRING ERROR (?ISE)

Nonfatal. A string entered in response to an INPUT statement begins with a quotation mark but is not terminated by the appropriate end quotation mark. BASIC assigns to the string all the characters between the initial quote and the line terminator and continues execution of the program.

?INTEGER OVERFLOW (?IOV)

An integer variable is assigned a value greater than 32767 or less than -32768 or an integer expression produces a result which exceeds this range. Change the variable or expression to a floating point format.

?LINE TOO LONG (?LTL)

The line entered is longer than BASIC allows; the line is ignored. If this message occurs when BASIC is reading a program from a file, BASIC stops reading the file. A possible cause is that you entered a line near the maximum size with no spaces, but when you save the program, BASIC adds spaces making the line too long. Split the line into several smaller lines.

?LINE TOO LONG TO TRANSLATE (?TLT)

Lines are translated as they are entered; the line just entered exceeds the area reserved for translating. The line is ignored. If this message is produced while BASIC is reading a program from a file, BASIC stops reading the file. Split the line into several smaller lines.

?MISSING SUBPROGRAM (?MSP)

The CALL statement specifies a nonexistent routine name. Ensure that the name is typed correctly (it must consist of upper case letters).

Table A-2 (Cont.)
BASIC-11 Error Messages

?NEGATIVE SQUARE ROOT (?NGS)

Nonfatal. The expression in the SQR (square root) function has a negative value. The function returns a value of 0. BASIC continues execution of the program.

?NESTED FOR STATEMENTS WITH SAME CONTROL VARIABLE (?FSV)

A FOR statement specifies the same control variable as that specified by a FOR NEXT loop that the FOR statement is inside. Change one of the control variables to a different variable name (in both the FOR and the corresponding NEXT statement).

?NEXT WITHOUT FOR (?NWF)

A NEXT statement is without a corresponding FOR statement. Ensure that each loop starts with a FOR statement and ends with a NEXT statement which specifies the same variable. This error message is also produced if control is transferred into the middle of a loop. FOR NEXT loops should only be entered by executing the FOR statement.

?NOT A VALID DEVICE (?NVD)

File specification contains an invalid device. See your BASIC-11 user's guide for information about file specifications.

?NOT ENOUGH ROOM (?NER)

There is not enough room for the file. See your BASIC-11 user's guide for more information.

?NUMBERS AND STRINGS (?NSM)

String and numeric values appear in the same expression or they are set equal to each other; for example, A$=2. Change either the data type of the variable (e.g., A=2) or the expression (e.g., A$="2") so that they are consistent.

?OUT OF DATA (?OOD)

The data list is exhausted and a READ statement requests additional data or the end of a file is reached and the INPUT # statement requests additional data. Ensure that there is sufficient data or test for the end-of-file condition with the IF END statement.

?PRINT USING ERROR (?PRU)

There is an error in the PRINT USING statement caused when the format specification is not a valid string, or is null, or does not contain one valid field. The error is also caused when an attempt is made to print a numeric value in a string field, a string value in a numeric field, or a negative number in a floating asterisk or floating dollar sign field that does not also specify a trailing minus sign. The message is also printed if the items in the list are not separated by commas or semicolons.

Table A-2 (Cont.)
BASIC-11 Error Messages

?PROGRAM TOO BIG (?PTB)

The line just entered causes the program to exceed the user area in memory; the line is ignored. Reduce program size (see Section 9.11). If this error occurs when BASIC is reading a program from a file, BASIC stops reading the file.

?RESEQUENCE ERROR (?RES)

Resequencing the program would cause lines to overlap or existing lines to be deleted, or would create an illegal line number. Reenter the command with different arguments.

?RETURN WITHOUT GOSUB (?RWG)

A RETURN is encountered before execution of a GOSUB statement. Do not transfer control to a subroutine except by executing a GOSUB or an ON GOSUB statement.

?STRING STORAGE OVERFLOW (?SSO)

Not enough memory is available to store all the strings used in the program. Reduce program size (see Section 9.11).

?STRING TOO LONG (?STL)

The maximum length of a string in a BASIC statement is 255 characters. Split string into several smaller strings.

?SUBSCRIPT OUT OF BOUNDS (?SOB)

The subscript computed is less than zero or is outside the bounds defined in the DIM statement. Ensure that expression specifying the subscript is in the correct range.

?SUBSTITUTE ERROR (?SUB)

There was no separator between the strings in the SUB command or the command would create an immediate mode statement. Retype SUB command.

?SYNTAX ERROR (?SYN)

BASIC has encountered an unrecognizable element. Common examples of syntax errors are misspelled commands, unmatched parentheses, and other typographical errors. This message can also be produced by attempting to read in a program from a file containing illegal characters, in which case BASIC stops reading the file. Retype program line or ensure that file contains a valid BASIC program.

?TOO MANY GOSUBS (?TMG)

More than 20 GOSUBS have been executed without a corresponding RETURN statement. Change the program logic so that less GOSUB statements are executed.

Table A-2 (Cont.)
BASIC-11 Error Messages

?TOO MANY ITEMS IN COMMON (?TIC)

There are more than 255 variable and array names in COMMON (A, A(100), A%, A%(10, 10), A$, and A$(5) are all considered different names). Reduce the number of items in COMMON by converting individual variables to elements of an array or by passing fewer items to the next program segment.

?UNDEFINED FUNCTIONS (?UFN)

A user-defined function has been used and not defined. Define the function. A function is defined only after the RUN command or CHAIN statement is executed.

?UNDEFINED LINE NUMBER (?ULN)

The line number specified in an IF, GO TO, GOSUB, ON GO TO, ON GOSUB, or CHAIN statement does not exist anywhere in the program. Ensure that the line number specified exists in the program.

?UNDIMENSIONED ARRAY IN CALL (?UAC)

The first reference to an undimensioned array appears in a CALL statement. Dimension the array with the DIM statement.

?USE REPLACE

Saving the program would have caused an existing file to be deleted. Use either a different file specification or the REPLACE command.

?VIRTUAL ARRAY CHANNEL ALREADY IN USE (?VCU)

The DIM # statement specifies a channel number which has already appeared in a DIM # statement. Specify another channel number.

Using BASIC functions improperly causes error messages to be printed. Table A-3 lists the functions and describes under which conditions BASIC functions produce errors.

Table A-3
Error Conditions in Functions

All functions

The argument used is the wrong type. For example, the argument is numeric and the function expects a string expression. This condition produces ?ARGUMENT ERROR (?ARG).

All functions

The wrong number of arguments is used in a function, or the wrong character is used to separate them. For example, PRINT SIN (X,Y) produces a syntax error because the SIN function has only one argument. This condition produces ?SYNTAX ERROR (?SYN).

ASC(string)

String is not a 1-character string. This condition produces ?ARGUMENT ERROR (?ARG).

BIN(string)

Character other than blank, 0, or 1 in string or value is greater than 2^16. This condition produces ?ARGUMENT ERROR (?ARG).

CHR$(expr)

Expression is not in the range 0 to 32767. This condition produces ?ARGUMENT ERROR (?ARG).

EXP(expr)

Value of expression is greater than 87. This condition produces ?EXPONENTIATION ERROR (?EER).

FNletter

The function FNletter is not defined (function cannot be defined by an immediate mode statement). This condition produces ?UNDEFINED FUNCTION (?UFN).

LOG(expr)

Expression is negative or 0. The function returns a value of 0. This condition produces ?BAD LOG (?BLG).

LOG10(expr)

Expression is negative or 0. The function returns a value of 0. This condition produces ?BAD LOG (?BLG).

OCT(string)

Character other than blank or digits 0 through 7 appears in string, or value is greater than 2^16. These conditions produce ARGUMENT ERROR (?ARG).

Table A-3 (Cont.)
Error Conditions in Functions

PI

An argument is included. This condition produces ?SYNTAX ERROR (?SYN).

SEG$(string,expr1,expr2)

No additional error conditions.

SQR(expr)

Expression is negative. The function returns a value of 0. This condition produces ?NEGATIVE SQUARE ROOT (?NGS).

TAB

Expression is not in the range 0 to 32767. This condition produces ?ARGUMENT ERROR (?ARG).

VAL(string)

String is not a numeric constant. This condition produces ?ARGUMENT ERROR (?ARG).

APPENDIX B

SUMMARY OF BASIC-11 STATEMENTS, FUNCTIONS, AND COMMANDS

B.1 SUMMARY OF BASIC-11 STATEMENTS

Table B-1 lists the BASIC-11 statements and provides a brief description of each. For more information see the section of this manual specified on the right.

Table B-1
Summary of Statements

	Section
CALL "routine name" [(argument list)]	3.3
Calls assembly language routines from a BASIC program.	
CHAIN string [LINE expression]	8.1
Terminates execution of the program, loads the program specified by string, and begins execution at the lowest line number or at the line number specified by expression. The string is a file specification.	
CLOSE [[#]expr1,[#]expr2,[#]expr3, ...]	6.2.2
Closes the file(s) associated with the channel number(s) and virtual file channel number(s) specified. If no channel number is specified, closes all open files.	
COMMON list	8.1.1
Preserves values and names of specified variables and arrays when the CHAIN statement is executed. Both string and arithmetic variables and arrays can be passed. The statement also dimensions the specified arrays. List is in the general format: var1[(expr[,expr])][,var2[(expr[,expr])],...]	

Table B-1 (Cont.)
Summary of Statements

	Section
DATA list	3.1.3
Used in conjunction with READ to input listed data into an executing program. Can contain any mixture of strings and numbers. Items must be separated by commas.	
DEF FNletter[{$ / %}](var1[,var2,...,var5])=expression	5.4
Defines a user function. Letter may be any single letter A through Z.	
DIM list	
Reserves space in memory for arrays according to the subscript(s) specified after the variable name. List is in the general format: var1(expr[,expr])[,var2(expr[,expr]),...]	
DIM #integer1,variable(integer2[,integer3])[=integer4]	6.4.1
Dimensions the virtual array file associated with the channel number specified by integer1. Integer4 specifies the string size for string virtual arrays.	
END	4.3
Optional. Placed at the physical end of the program to terminate execution.	
FOR var=expr1 TO expr2 [STEP expr3]	4.2.1
Sets up a loop to be executed the specified number of times.	
GOSUB line number	4.4.1
Unconditionally transfers control to specified line of subroutine.	
GO TO line number	4.1.1
Unconditionally transfers control to specified line number.	
IF relational expression {THEN statement / THEN line number / GO TO line number}	4.1.3
Conditionally executes the specified statement or transfers control to specified line number. When the condition is not true and a statement is specified, execution continues at the next sequential line. When the condition is not true and a line number is specified, execution continues at the next sequential statement. The expressions and the relational operator must all be string or all be numeric.	

Table B-1 (Cont.)
Summary of Statements

	Section
IF END #expr {THEN statement / THEN line number / GO TO line number}	6.3.3
Tests for end-of-file condition of input sequential file associated with channel number specified by expression.	
INPUT [#expr,]variable1[,variable2,...]	3.1.1 and 6.3.1
Inputs data from your terminal or from the file associated with the channel number specified by expression. Variables may be arithmetic or string.	
KILL string	6.6
Deletes file specified by string.	
[LET] variable=expression	2.5
Assigns value of expression to the specified variable. Variable and expression must be of the same type, either numeric or string.	
LINPUT [#expr,]string var1[,string var2,...]	3.1.2 and 6.3.1
Inputs string data from the terminal or from the file associated with channel number specified by expression. Variables can only be string variables.	
NAME string1 TO string2	6.5
Renames file specified by string1 to name specified by string2.	
NEXT variable	4.2.1
Placed at end of FOR loop to return control to FOR statement.	
ON expression GOSUB line number1[,line number2,line number3,...]	4.4.2
Conditionally transfers control to subroutine at one line number specified in list. Value of expression determines the line number to which control is transferred.	
ON expression GO TO line number1[,line number2,line number3,...]	4.1.2
Conditionally transfers control to one line number in the list. Value of expression determines the line number to which control is transferred.	
ON expression THEN line number1[,line number2,...]	
Equivalent to ON GO TO.	

Table B-1 (Cont.)
Summary of Statements

Section

OPEN string [{FOR INPUT / FOR OUTPUT}] AS FILE [#]expr [DOUBLE BUF] [,RECORDSIZE expr] [,MODE expr] [,FILESIZE expr] 6.2.1

Opens a file specified by string for input or output as specified (assumes input if neither specified) and associates file with the channel number specified by exprl. String is a file specification.

OVERLAY string [LINE expression] 7.2

Overlays or merges the program currently in memory with the program in the file specified by string, and when overlay is completed, transfers control to either the next sequential BASIC line number or the line number specified by expression. String is a file specification.

PRINT [#expr,][list] 3.2 and 6.3.2

Prints items in list on the terminal or to the file associated with channel number specified by expression. List can consist of string and arithmetic expressions and the TAB function. Items can be separated by either commas or semicolons.

PRINT [#expr,]USING string, list 7.4

Prints items in list on the terminal or to the file associated with channel number specified by expr in the format determined by string. List can consist of string and arithmetic expressions. Items can be separated by either commas or semicolons.

RANDOMIZE 5.2.3

Causes the random number generator (RND function) to produce different random numbers.

READ variable1[,variable2,...] 3.1.3

Assigns values listed in DATA statements to specified variables. Variables may be string or numeric.

REM comment 1.6

No effect on execution of program. Contains explanatory comments about the BASIC program.

RESET[#expr]

Equivalent to RESTORE.

RESTORE[#expr] 3.1.3 and 6.3.4

Resets either the data pointer or, when specified, the input file associated with the specified channel number to the beginning.

Table B-1 (Cont.)
Summary of Statements

	Section
RETURN	4.4.1
Terminates a subroutine and returns control to the statement following the last executed GOSUB statement.	
STOP	4.3
Terminates execution of the program. Placed at logical end(s) of the program.	

B.2 SUMMARY OF BASIC-11 FUNCTIONS

ARITHMETIC FUNCTIONS

Table B-2 lists the BASIC-11 arithmetic functions and provides a description of each. For more information about a function, see the section of this manual specified on the right.

Table B-2
Summary of Arithmetic Functions

	Section
ABS(expr)	5.2.2.4
Returns the absolute value of the expression.	
ATN(expr)	5.2.1
Returns the arctangent of the expression as an angle in radians in the range + or - pi/2.	
COS(expr)	5.2.1
Returns the cosine of the angle specified by the expression in radians.	
EXP(expr)	5.2.2.2
Returns the value of e raised to the expression power where e is (approximately) 2.71828.	
INT(expr)	5.2.2.3
Returns the greatest integer less than or equal to the expression.	
LOG(expr)	5.2.2.2
Returns the natural logarithm of the expression.	

Table B-2 (Cont.)
Summary of Arithmetic Functions

	Section
LOG10(expr)	5.2.2.2
Returns the base 10 logarithm of the expression.	
PI	5.2.1
Returns the value of pi (3.141593).	
RND[(expr)]	5.2.3
Returns a random number between 0 and 1.	
SGN(expr)	5.2.2.5
Returns a value indicating the sign of expression.	
SIN(expr)	5.2.1
Returns the sine of the angle specified by expression in radians.	
SQR(expr)	5.2.2.1
Returns the square root of the expression.	
TAB(expr)	3.2.3
Causes the terminal type head to tab to column number specified by the expression (valid only in PRINT statements).	

Table B-3 lists the BASIC-11 string functions and provides a description of each. For more information about a function, see the section in this manual specified on the right.

Table B-3
Summary of String Functions

	Section
ASC(string)	5.3.2.1
Returns as a decimal number the 8-bit internal code (ASCII value) for the 1-character string expression.	
BIN(string)	5.3.2.3
Converts a string expression containing a binary number to a decimal value. Blanks are ignored.	
CHR$(expr)	5.3.2.1
Generates a 1-character string whose ASCII value is the low-order 8 bits of the integer value of the expression.	

Table B-3 (Cont.)
Summary of String Functions

	Section
CLK$	5.5
Returns the time as a string in the form hh:mm:ss (for example 12:30:15).	
DAT$	5.5
Returns the date as a string in the form dd-mon-yr (for example 07-FEB-75).	
LEN(string)	5.3.1.1
Returns the number of characters in the string.	
OCT(string)	5.3.2.3
Converts a string expression containing an octal number to a decimal value. Blanks are ignored.	
POS(string1,string2,expr)	5.3.1.3
Searches for and returns the position of the first occurrence of string2 in string1. The search starts at the character position specified by expression.	
SEG$(string,expr1,expr2)	5.3.1.4
Returns the string of characters in position specified by expression1 through the position specified by expression2.	
STR$(expr)	5.3.2.2
Returns the string which represents the numeric value of the expression.	
TRM$(string)	5.3.1.2
Returns string without trailing blanks.	
VAL(string)	5.3.2.2
Returns the value of the decimal number contained in the string.	

B.3 SUMMARY OF BASIC-11 COMMANDS

Table B-4 lists the BASIC-11 commands and provides a description of each. For more information about a command, see the section of this manual specified on the right.

Table B-4
Summary of Commands

	Section
APPEND [file specification]	9.6.2
Merges the program in your area in memory with the program specified by the file specification.	
CLEAR	9.5
Initializes all variables to 0 and all string variables to nulls and deletes arrays.	
COMPILE [file specification]	9.10
Saves a compiled version of the program.	
DEL line specification[,line specification,...]	9.5
Deletes specified lines.	
LENGTH	9.11
Prints on your terminal the size of the program in memory and the size of the remaining free memory.	
LIST[NH][line specification1,line specification2,...]	9.2
Prints on the terminal the specified line(s) of the program currently in memory. NH suppresses the printing of the header line.	
NEW [program name]	9.5
Erases your storage area and sets the current program name to the one specified.	
OLD [file specification]	9.6.2
Erases your storage area and inputs the program from the specified file.	
RENAME program name	9.7
Changes the current program name to the one specified.	
REPLACE [file specification]	9.6.1
Replaces the specified file with the current program.	
RESEQ[[new line number],[old line number1][-old line number2],[increment]]	9.9
Resequences program as specified.	
RUN[NH]	9.3
Executes the program in memory. NH suppresses the printing of the header line.	

Table B-4 (Cont.)
Summary of Commands

	Section
RUN[NH]file specification	9.6.3
Erases your storage area, inputs the program from the specified file, and then executes the program. Does not print header line in any case.	
SAVE [file specification]	9.6.1
Outputs the program in memory to the specified file.	
SCR	9.5
Erases your storage area and changes the program name to NONAME.	
SUB line numberxstringlxstring2[xinteger]	9.8
Substitutes the integer occurrence of string1 with string2 on line specified. x is a delimiter and can be any character such as @.	
UNSAVE file specification	9.6.4
Deletes specified file.	
Key Commands	
CTRL/C	9.1
Interrupts execution of a command or program and causes BASIC to print the READY message. See your BASIC-11 user's guide for more information about CTRL/C.	
CTRL/O	9.1
Causes all further terminal output to be discarded. If an INPUT statement is encountered, CTRL/O is retyped, or the program is terminated, printing resumes.	
CTRL/Q	9.1
Continues output to the terminal; cancels effect of CTRL/S.	
CTRL/S	9.1
Temporarily suspends all output to terminal until CTRL/Q is typed; allows alphanumeric display terminals to be read or photographed before data is moved off screen.	
CTRL/U	9.1
Deletes the entire current input line (provided the RETURN key has not been typed).	
RUBOUT	9.1
Deletes the last character typed.	

APPENDIX C

ASCII CHARACTER SET

The following table shows, with the corresponding octal and decimal codes, the 128-character ASCII (American Standard Code for Information Interchange) character set. These codes are used to store ASCII data in files and to store them internally.

The BASIC user can convert an ASCII value to the corresponding string character with the CHR$ function and can convert a string character to the corresponding ASCII value with the ASC function (see Section 5.3.2.1).

BASIC also uses the ASCII values of the characters in string comparisons (see Section 2.4.3).

The octal code is provided for reference. BASIC does not support octal numbers except through the OCT function (see Section 5.3.2.3).

ASCII characters are stored internally and in files in eight bits. The eighth (high order) bit is normally 0.

Table C-1
ASCII Character Set

ASCII Decimal Code	ASCII 7-Bit Octal Code	Character
0	000	NUL (CTRL/@)
1	001	SOH (CTRL/A)
2	002	STX (CTRL/B)
3	003	ETX (CTRL/C)
4	004	EOT (CTRL/D)
5	005	ENQ (CTRL/E)
6	006	ACK (CTRL/F)
7	007	BEL (CTRL/G)
8	010	BS (CTRL/H)
9	011	HT (CTRL/I or TAB)
10	012	NL (NEW LINE or LINE FEED)
11	013	VT (Vertical TAB)
12	014	FF (Form Feed)
13	015	RT (Return)
14	016	SO (CTRL/N)
15	017	SI (CTRL/O)
16	020	DLE (CTRL/P)
17	021	DC1 (CTRL/Q)
18	022	DC2 (CTRL/R)

Table C-1 (Cont.)
ASCII Character Set

ASCII Decimal Code	ASCII 7-Bit Octal Code	Character
19	023	DC3 (CTRL/S)
20	024	DC4 (CTRL/T)
21	025	NAK (CTRL/U)
22	026	SYN (CTRL/V)
23	027	ETB (CTRL/W)
24	030	CAN (CTRL/X)
25	031	EM (CTRL/Y)
26	032	SUB (CTRL/Z)
27	033	ESC (ESCAPE)
28	034	FS (CTRL/\)
29	035	GS (CTRL/])
30	036	RS (CTRL/^)
31	037	US (CTRL/)
32	040	SP (space bar)
33	041	!
34	042	"
35	043	#
36	044	$
37	045	%
38	046	&
39	047	'
40	050	(
41	051	)
42	052	*
43	053	+
44	054	,
45	055	-
46	056	.
47	057	/
48	060	0
49	061	1
50	062	2
51	063	3
52	064	4
53	065	5
54	066	6
55	067	7
56	070	8
57	071	9
58	072	:
59	073	;
60	074	<
61	075	=
62	076	>
63	077	?
64	100	@
65	101	A
66	102	B
67	103	C
68	104	D
69	105	E
70	106	F
71	107	G
72	110	H

Table C-1 (Cont.)
ASCII Character Set

ASCII Decimal Code	ASCII 7-Bit Octal Code	Character
73	111	I
74	112	J
75	113	K
76	114	L
77	115	M
78	116	N
79	117	O
80	120	P
81	121	Q
82	122	R
83	123	S
84	124	T
85	125	U
86	126	V
87	127	W
88	130	X
89	131	Y
90	132	Z
91	133	[
92	134	\
93	135	]
94	136	^
95	137	—
96	140	`
97	141	a
98	142	b
99	143	c
100	144	d
101	145	e
102	146	f
103	147	g
104	150	h
105	151	i
106	152	j
107	153	k
108	154	l
109	155	m
110	156	n
111	157	o
112	160	p
113	161	q
114	162	r
115	163	s
116	164	t
117	165	u
118	166	v
119	167	w
120	170	x
121	171	y
122	172	z
123	173	{
124	174	|
125	175	}
126	176	~
127	177	RUBOUT

INDEX

Abbreviated error messages, A-2 to A-3
ABS function, 5-8
Absolute value function, 5-8
Accessing data in sequential files, 6-4
Accuracy, digits of, 7-8
Addition, 2-8
Algebraic functions, 5-1
Alphabetical order, comparing strings in, 2-11 to 2-12
Alphanumeric strings, 2-6
Ampersand, string concatenation operator, 2-10
APPEND command, 9-6
Area in memory, program storage, 1-7
Arithmetic
  expressions, 2-8 to 2-10
  functions, 2-13, 5-1 to 5-11
  functions, summary, B-5, B-6
  operator precedence, 2-10
  operators, 2-8 to 2-10
  relational expressions, 2-11
  relational operators, 2-11
Arrays, 2-7, 2-14 to 2-18
  compared to virtual array files, 6-8
  dimensioning, 2-16 to 2-18
  dimensioning virtual, 6-9
  first element in, 2-15
  initializing, 9-4
  numeric, 2-17
  reserving space for, 2-16
  storage order of, 2-16
  string, 2-18
ASC function, 5-17
Ascending order, execution of statements in, 1-3, 1-7, 4-1
ASCII, C-1
  character set, 1-2, C-1 to C-3
  code conversions, 5-17
  values, C-1 to C-3
  values in comparing strings, 2-12
Assembly language routines, 8-9
Assigning values to variables, 2-13
Asterisk
  fill in PRINT USING statement, 7-6
  multiplication operator, 2-8
ATN function, 5-2
Backslash, 1-5, 1-6
Base a log, 5-5
Base e log, 5-5
Base 10 log, 5-6
BASIC, x, 1-1
  character set, 1-2
  program structure, 1-1
BASIC-11, x, 1-1
Beginning of arrays, 2-15
BIN function, 5-20
Binary functions, 5-20
Blank lines, printing, 3-8
Block I/O, 6-8
Braces, ix
Brackets, square, ix
Branch, computed, 4-3
Branching, 4-1 to 4-3
Branching, multiple, 4-3
C in PRINT USING statement, 7-10
Calculating expressions in the PRINT statement, 3-8
Calculator, using BASIC as a, 1-9
CALL statement, 8-9
Calling
  routines, 8-9
  user-defined functions, 5-22
Centered format, 7-10
Centering strings, 7-10
Chain, preserving variables through, 8-3
CHAIN statement, 8-1 to 8-6
Chaining to compiled programs, 8-3
Changing a program line, 1-7, 9-9
Changing the program name, 9-8
Channel number, file, 6-2 to 6-5, 6-7
Characters
  ASCII, 1-2, C-1
  BASIC, 1-2
  conversion, 5-17
  nonprinting, 1-2
Checking
  for the end of input file, 6-7
  the length of a program, 9-13
CHR$ function, 5-17
Circumflex, exponentiation operator, 2-8
CLEAR command, 9-4
CLK$ function, 5-26
CLOSE statement, 6-3

Closing files, 6-3, 6-10
Closing virtual arrays, 6-10
Comma in PRINT statement, 3-9 to 3-12
Commands, 1-8, 9-1 to 9-14
  APPEND, 9-6
  CLEAR, 9-4
  COMPILE, 9-13
  DEL, 9-3
  key, 9-1
  LIST, 1-8, 9-2
  LISTNH, 9-2
  NEW, 9-4
  OLD, 9-6
  RENAME, 9-8
  REPLACE, 9-5
  RESEQ, 9-10 to 9-13
  RUN, 1-8, 9-3
  RUN file specification, 9-7
  RUNNH, 9-3
  SAVE, 9-5
  SCR, 9-4
  SUB, 9-9
  summary of, B-8 to B-9
  summary of key, B-9
  UNSAVE, 9-8
Commas in PRINT USING statement, 7-7
Comment, 1-6
COMMON statement, 8-3 to 8-6
  order of, 8-4
Communicating data between program segments, 8-4, 8-6, 8-9
Comparing strings, 2-12 to 2-13
COMPILE command, 9-13
Compiled programs
  chaining to, 8-3
  efficiency of, 9-13
Computed
  branch, 4-3
  GO TO statement, 4-3
  GOSUB statement, 4-17
Concatenation, string, 2-10
Conditional transfer, 4-3 to 4-6
Constants, 2-1 to 2-4
  integer, 2-1, 2-3
  numeric, 2-1
  string, 2-1, 2-3
Control
  shifting, 4-1
  statements, 4-1 to 4-17
  variable in ON GO TO statement, 4-3
Conventions, documentation, ix to x
Conversions
  ASCII code, 5-17
  character, 5-17
  functions, 5-16 to 5-21
Copying segments from a string, 5-15
COS function, 5-2
Cosine function, 5-2
Counter in loops, 4-8
Creating files, 6-2
CTRL keys
  CTRL/C, 9-1
  CTRL/O, 9-1
  CTRL/Q, 9-1
  CTRL/S, 9-1
  CTRL/U, 9-2

DAT$ function, 5-26
Data
  excess, 3-2
  files, 6-1 to 6-11
  items, format of, 3-5
  pointer, restoring the, 3-7
  reading, 3-5 to 3-7
  storing, 6-1, 6-6 to 6-7
  supplying, 3-1 to 3-7
DATA statement, 3-5 to 3-7
Date function, 5-26
Decimal point
  in numeric constants, 2-1
  in PRINT USING statement, 7-3
DEF statement, 5-21
Defining functions, 5-21
  in immediate mode, 5-25
DEL command, 9-3
Deleting
  data files, 6-11
  program files, 9-8
  program lines, 9-3
Digits in PRINT USING statement, number of, 7-3
Digits of accuracy, 7-8
DIM statement, 2-16 to 2-18
DIM # statement, 6-9
Dimensioning
  arrays, 2-16 to 2-18
  virtual arrays, 6-9
Direct I/O, 6-8
Division, 2-8
Documentation conventions, ix to x
Documenting procedures, 1-6
Dollar signs
  in PRINT USING statement, 7-6
  in string function names, 5-12
  in string names, 2-6
  in user-defined function name, 5-21
Double quotation marks, 2-3
Dummy variables, 5-22

E format, printing numbers in, 7-8
E in PRINT USING statement, 7-11
E notation, 2-2, 7-8
E, an algebraic constant, 5-4
Editing a program line, 9-9
Efficiency of compiled programs, 9-13
Ellipsis, ix
End of file, 6-7
END statement, 4-12
Entering BASIC programs, 1-7
Entering data, 3-1
Equal sign
  in LET statement, 2-13
  relational operator, 2-11
Erasing programs, 9-4
Error conditions
  in functions, A-11 to A-12
  in PRINT USING statement, 7-3, 7-15 to 7-18
Error messages, A-1 to A-12
  abbreviated, A-2 to A-3
  fatal, A-1
  nonfatal, A-1
  summary, A-3 to A-11
Evaluating expressions, 2-9
Excess data, 3-2
Executing a program, 9-3
Execution
  of loops, 4-6
  stopping program, 4-12 to 4-13
EXP function, 5-4
Exponential function, 5-4
Exponentiation, 2-8
Expr, x
Expression, ix
Expressions, 2-8 to 2-13
  arithmetic, 2-8 to 2-10
  arithmetic relational, 2-11
  evaluating, 2-9 to 2-10
  floating point, 2-8 to 2-9
  in the PRINT statement, 3-8
  integer, 2-9
  mixed mode, 2-9
  relational, 2-11 to 2-13
  string, 2-10
Extended string fields, 7-11
Extracting a segment from a string, 5-16

Fatal error messages, A-1
Fields
  centered, 7-10
  extended, 7-11
  format of numeric, 7-13
  format of string, 7-14
  numeric, 7-3
Fields (cont.),
  one-character string, 7-9
  string, 7-8
Files, 6-1 to 6-10, 9-4 to 9-8
  channel number, 6-2 to 6-5, 6-7
  closing, 6-3
  control statements, 6-1 to 6-4
  data, 6-1 to 6-10
  deleting, 6-11, 9-8
  program, 6-1, 8-1, 9-4 to 9-8
  renaming, 6-10
  resetting, 6-8
  restoring program, 9-6
  running programs from, 9-7
  sequential, 6-1
  specification, x, 6-2
  using, sequential, 6-4
  writing, 6-5
Finding
  a square root, 5-4
  the length of a string, 5-12
  the position of a substring, 5-13
First element in arrays, 2-15
Floating point
  expressions, 2-9
  format, 2-3
  numbers, 1-10, 2-5
FN function, 5-21
FOR INPUT in OPEN statement, 6-2
FOR NEXT loops, 4-7 to 4-12
  in immediate mode, 1-10
  STEP value in, 4-9 to 4-10
  terminating condition of, 4-8
FOR OUTPUT in OPEN statement, 6-2
FOR statement, 4-7 to 4-12
Format
  centered, 7-10
  data items, 3-5
  error messages, A-1
  floating point, 2-3
  line, 1-3
  numeric fields, 7-13
  numeric output, 3-12
  output, 3-12
  string fields, 7-14
  strings, left-justified, 7-9
  strings, right-justified, 7-9
  strings with PRINT, 3-12
Formatted output, 3-9 to 3-12, 7-1 to 7-18
Functions, 2-12 to 2-13, 5-1 to 5-26
  ABS, 5-8
  algebraic, 5-1, 5-4 to 5-9
  ASC, 5-17
  ATN, 5-2
  BIN, 5-20
  CHR$, 5-17
  CLK$, 5-26

Functions (cont.),
conversion, 5-16
COS, 5-2
DAT$, 5-26
error conditions in, A-11
EXP, 5-4
FN, 5-21
INT, 5-6
integer, 5-6
LOG, 5-4
LOG1Ø, 5-6
numeric, 5-1 to 5-11
OCT, 5-20
POS, 5-13
RND, 5-9
SEG$, 5-15
SGN, 5-9
SIN, 5-2
SQR, 5-4
STR$, 5-18
string, 5-12 to 5-21
summary of arithmetic, B-5
summary of string, B-6
TAB, 3-13
trigonometric, 5-1 to 5-3
TRM$, 5-13
types of, 5-1
user-defined, 5-21 to 5-23
VAL, 5-18

GO TO statement, 4-1 to 4-3
computed, 4-3
GOSUB statement, 4-14
Greater than or equal to relational operator, 2-11
Greater than relational operator, 2-11

Halting program execution, 4-12
Header lines, 1-8, 9-2, 9-3
Highest line number in program, 4-12

IF END # statement, 6-7
IF THEN statement, 2-11, 4-3 to 4-6
Immediate mode statements, 1-9 to 1-10
defining a function in, 5-25
Index values in FOR NEXT loop, 4-10
Infinite loop, 4-3
Initializing
arrays, 2-5, 9-4
program storage, 9-4
Initializing (cont.),
variables, 2-5, 9-4
your area, 9-4
Input
entering, 3-1
leading spaces in string, 3-3
string, 3-4
INPUT statement, 3-1 to 3-3
INPUT # statement, 6-4
INPUT #Ø statement, 3-4
Int, x
INT function, 5-6
Integer, x, 1-10, 2-3, 2-5
constants, 1-10, 2-3
constants, range of, 2-3
expression, 2-9
function, 5-6
random, 5-11
variables, 2-5
Items
in capital letters, ix
in lower-case letters, ix

Key commands, 9-1, B-9
CTRL/C, 9-1
CTRL/O, 9-1
CTRL/Q, 9-1
CTRL/S, 9-1
CTRL/U, 9-2
RUBOUT, 9-2
Keywords, ix, 1-3, 1-4
KILL statement, 6-11

L in PRINT USING statement, 7-9
Leading spaces in string input, 3-3
Left angle bracket relational operator, 2-11
Left-justified format strings, 7-9
Length of a string, 5-12
Less than or equal to relational operator, 2-11
Less than relational operator, 2-11
LET statement, 2-13
Letters
lower-case, 1-2
upper-case, 1-2
Line
format, 1-3
multi-statement, 1-6
number, x, 1-3
single statement, 1-5
terminator, 1-3
LINPUT statement, 3-4
LINPUT # statement, 6-5

LIST command, 1-8, 9-2
Listing programs, 9-2
LISTNH command, 9-2
Lists, 2-7, 2-14
LOG function, 5-4
LOGlØ function, 5-6
Logarithm functions, 5-4
Logic, program, 4-1
Loops
  execution of, 4-6
  FOR NEXT, 4-7
  infinite, 4-3
  nested, 4-11
  overlapping, 4-11
  STEP value in FOR NEXT, 4-10
Lower-case letters, 1-2

Mathematical functions, 2-13, 5-1 to 5-11
Matrices, 2-14
Memory
  program storage area in, 1-7
  saved by reducing program size, 9-14
  saved by segmenting programs with CHAIN, 8-1
  saved by segmenting programs with OVERLAY, 8-7
Merging program segments, 8-6
Messages
  error, A-1
  fatal, error, A-1
  nonfatal, error, A-1
  READY, 1-8
  STOP, 4-13
Minus sign
  subtraction operator, 2-8
  trailing, 7-5
  unary, 2-8
Mixed mode expressions, 2-9
Multi-statement line, 1-6
Multiple
  branching, 4-3
  GOSUB statement, 4-17
Multiplication, 2-8

NAME statement, 6-10
Nested
  loops, 4-11
  parentheses, 2-9
NEW command, 9-4
NEXT statement, 4-7
NONAME program name, 9-4
Nonfatal error messages, A-1
Nonprinting characters, 1-2
Not equal to relational operator, 2-11
Notations, numeric, 2-2
Number
  of digits in PRINT USING statement, 7-3
  of subscripts, 2-16
  signs in PRINT USING statement, 7-3
Numbers
  floating point, 1-10
  output format of, 3-12
  random, 5-9
  real, 1-10
  rounding off of, 5-7
  string representation of, 5-18
  truncating, 5-6
  whole, 5-6
Numeric
  arrays, 2-17
  constants, 1-10
  constants, range of, 2-2
  field, 7-3
  field, format of, 7-13
  functions, 5-1
  output format, 3-12
  variables, 2-5

OCT function, 5-20
Octal
  ASCII code, C-1
  function, 5-20
OLD command, 9-6
One-character string fields, 7-9
One-dimensional arrays, 2-14
ON GO TO statement, 4-3
ON GOSUB statement, 4-17
ON THEN statements, 4-3
OPEN statement, 6-2
Opening a file, 6-2
Operators
  arithmetic, 2-8
  arithmetic relational, 2-11
  equal sign relational, 2-11
  greater than or equal to relational, 2-11
  greater than relational, 2-11
  left angle bracket relational, 2-11
  less than or equal to relational, 2-11
  less than relational, 2-11
  not equal to relational, 2-11
  precedence of arithmetic, 2-10
  right angle bracket relational, 2-11

Operators (cont.),
  string, 2-10
  string relational, 2-12
Order of
  array storage, 2-16
  common statements, 8-4
  entering program lines, 1-7
  evaluating expressions, 2-10
Output, 3-8
  format of numbers, 3-12
  format of strings, 3-12
  formatted, 3-9, 3-13, 7-1 to 7-18
Overlapping loops, 4-11
OVERLAY statement, 8-6
Overlaying programs, 8-6

Parenthesis, nested, 2-9
Percent signs
  in integer constants, 2-3
  in integer variable names, 2-6
  in PRINT USING statement, 7-4
  in user-defined function name, 5-21
PI function, 5-2
Plus sign
  addition operator, 2-8
  string concatenation operator, 2-10
  unary, 2-8
POS function, 5-13
Position of a substring, 5-13
Precedence of arithmetic operators, 2-10
Preserving variables through CHAIN, 8-3
PRINT statement, 3-8 to 3-15
  compared to PRINT USING, 9-1
  separators in, 3-9
PRINT # statement, 6-5
Printing, 3-8 to 3-15, 7-1 to 7-18
  data to a file, 6-5 to 6-7, 7-11
  quotation marks, 2-4, 7-18
  zones, 3-9
PRINT USING statement, 7-1 to 7-18
  compared to PRINT, 7-1
  format of, 7-2, 7-11 to 7-15
  error conditions in, 7-15 to 7-18
  printing numbers, 7-2 to 7-8
  printing strings, 7-8 to 7-10
Program lines
  deleting, 9-3
  editing, 9-9
  format of, 1-5
  order of entering, 1-7
  spaces in, 1-4
Programs
  changing the name of, 9-8
  checking the size of, 9-13 to 9-14
  deleting, 9-8
  in files, 6-1, 8-1, 9-4 to 9-8
  initializing, 9-4
  listing of, 9-2
  logic of, 4-1
  merging, 8-6
  overlaying, 8-6
  reducing the size of, 9-14
  resequencing, 9-10 to 9-13
  restoring, 9-6
  running, 1-8, 9-3, 9-7
  saving compiled, 9-13
  segmentation, 8-1 to 8-10
  stopping, 4-12 to 4-13
  storage area in memory, 1-7
  structure of, x
  termination of, 4-12 to 4-13

Question mark in INPUT statement, 3-1
Quotation marks, 2-3 to 2-4
  in string input, 3-3
  printing, 2-4
  printing with PRINT USING, 7-18

R in PRINT USING statement, 7-10
Radians, 5-2
Random
  access files, 6-8
  integers, 5-11
  numbers, 5-9 to 5-11
RANDOMIZE statement, 5-9 to 5-13
Range of
  integer constants, 2-3
  numeric constants, 2-2
  subscripts, 2-7
READ statement, 3-5
Reading
  data from a file, 6-4 to 6-5
  data values, 3-5
  program files, 9-6
READY message, 1-8

Real numbers, 1-10
Red ink, x
Reducing program size, 9-14
Relational
  arithmetic expressions, 2-11
  expression, 2-11, 4-3
  operators, 2-11
REM statement, 1-6
  transferring control to, 4-2
RENAME command, 9-8
Renaming
  files, 6-10
  programs, 9-8
REPLACE command, 9-5
Replacing a segment of a program line, 9-9
Representation of numbers, string, 5-18
RESEQ command, 9-10 to 9-13
Resequencing a program, 9-10 to 9-13
Reserving
  digits in PRINT USING statement, 7-3
  space for arrays, 2-16
Resetting a file, 6-8
RESTORE statement, 3-7
RESTORE # statement, 6-8
Restoring
  data pointer, 3-7
  files, 6-8
  program files, 9-6
Results, printing the, 3-8
Retrieving program files, 9-6
RETURN
  key, 3-2
  statement, 4-14
Right angle bracket relational operator, 2-11
Right-justified format strings, 7-9
RND function, 5-9
Rounding off numbers, 5-7
Routines, assembly language, 8-9
RUBOUT key, 1-7, 9-2
RUN command, 1-8, 9-3, 9-7
  with file specification, 9-7
RUNNH command, 9-3
Running a program from a file, 9-7

SAVE command, 9-5
Saving
  compiled programs, 9-13
  programs, 9-5
SCR command, 9-4
Search, string, 5-13
SEG$ function, 5-15
Segmentation function, 5-15
Segmenting programs, 8-1 to 8-10
  with CALL statement, 8-9 to 8-10
  with CHAIN statement, 8-1 to 8-6
  with OVERLAY statement, 8-6 to 8-9
Semicolon in PRINT statement, 3-9
Separators in PRINT statement, 3-9
Sequence of statement execution, 4-1
Sequential files, 6-1, 6-4 to 6-8
  compared to virtual array files, 6-8
SGN function, 5-9
Shifting control, 4-1 to 4-17
Sign function, 5-9
Signed 2's complement integer, 5-20, 5-21
Simple numeric variables, 2-5
SIN function, 5-2
Sine function, 5-2
Single quotation marks, 2-3
  in PRINT USING statement, 7-9
Single statement line, 1-5
Size, reducing program, 9-14
Slash, division operator, 2-8
Spaces in program lines, 1-4
Special characters, 1-2
Special symbols, ix
  in PRINT USING statement, 7-5
Specification, file, 6-2
SQR function, 5-4
Square brackets, ix
Square root function, 5-4
Statements
  CALL, 8-9 to 8-10
  CHAIN, 8-1 to 8-6
  CLOSE, 6-3
  COMMON, 8-3 to 8-6
  control, 4-1 to 4-17
  DATA, 3-5 to 3-7
  DEF, 5-21 to 5-25
  DIM, 2-16 to 2-18
  DIM #, 6-9
  END, 4-12
  execution sequence of, 4-1
  file control, 6-1
  FOR statement, 4-7 to 4-12
  GOSUB, 4-14 to 4-16
  GO TO, 4-1 to 4-2
  IF END, 6-7 to 6-8
  IF THEN, 2-11, 4-3 to 4-6
  immediate mode, 1-9 to 1-10
  in IF THEN statement, 4-4
  INPUT, 3-1 to 3-3
  INPUT #Ø, 3-4
  KILL, 6-11
  LET, 2-13 to 2-14
  LINPUT, 3-4 to 3-5

Statements (cont.),
LINPUT #, 6-5
NAME, 6-10
NEXT, 4-7 to 4-12
non-executable, 1-4
ON GOSUB, 4-17
ON GOTO, 4-3
OPEN, 6-2
OVERLAY, 8-6 to 8-9
PRINT, 3-8 to 3-15
PRINT USING, 7-1 to 7-18
RANDOMIZE, 5-9
READ, 3-5 to 3-7
REM, 1-6
RESTORE, 3-7
RESTORE #, 6-8
RETURN, 4-14 to 4-16
STOP, 4-12 to 4-13
summary of, B-1 to B-5
STEP value in FOR NEXT loops, 4-10
STOP
message, 4-13
statement, 4-12 to 4-13
Stopping program execution, 4-12
Storage order of arrays, 2-16
Storing data, 6-1, 6-5
STR$ function, 5-18
String, x, 1-3, 2-3 to 2-6, 2-10
alphanumeric, 2-6
arrays, 2-17
centering, 7-10
concatenation, 2-10
constants, 1-3, 2-1, 2-3 to 2-4
copying segments from a, 5-15
expressions, 2-10
extracting segments from a, 5-16
fields, 7-8 to 7-14
finding the length of a, 5-12
functions, 2-13, 5-12
input, 3-3, 3-4 to 3-5
input with leading spaces, 3-3
input with trailing spaces, 3-3
left-justified format, 7-9
operators, 2-10
output format, 3-12
relational operators, 2-11 to 2-12
representation of numbers, 5-18
right-justified format, 7-9
search, 5-13
summary of functions, B-6 to B-7
variables, 2-5, 2-6
Structure of a BASIC program, 1-1 to 1-2
SUB command, 9-9
Subroutines, 4-13
Subscripted variables, 2-7
Subscripts, 2-7
number of, 2-16
range of, 2-7
Substring, finding the position of a, 5-13
Subtraction, 2-8
Summaries
arithmetic functions, B-5 to B-6
commands, B-8 to B-9
error message, A-3 to A-10
key commands, B-9
statements, B-1 to B-5
string functions, B-6
Supplying data, 3-1, 3-5 to 3-7
Symbols, special, ix
asterisk multiplication operator, 2-8
backslash, 1-5, 1-6
braces, v
circumflex exponentiation operator, 2-8
dollar sign, in string names, 2-6
double quotation marks, 2-3
equal sign in LET statement, 2-13
equal sign relational operator, 2-11
minus sign, unary, 2-8
plus sign, unary, 2-8
single quotation marks, 2-3
slash division operator, 2-8
square brackets, v

TAB function, 3-13
Tables, 2-7
Tangent function, 5-2
Terminating
condition of FOR NEXT loops, 4-8
the program, 4-12
Time functions, 5-26
Trailing spaces
in string comparison, 2-12
in string input, 3-3
trimming, 5-13
Transcendental number, 5-2
Transfers
conditional, 4-3 to 4-6
data between program segments, 8-4
to a REM statement, 4-2
unconditional, 4-1 to 4-2
Trigonometric functions, 5-1
Trimming trailing blanks on strings, 5-13
TRM$ function, 5-13
Truncating numbers, 5-6
Two-dimensional arrays, 2-14

Type-in, user, x
Types of functions, 5-1

Unary
  minus sign, 2-8
  plus sign, 2-8
Unconditional transfer, 4-1
UNSAVE command, 9-8
Upper-case letters, 1-2
User-defined function, 5-21 to 5-25
Using sequential files, 6-4

VAL function, 5-18
Values, ASCII, C-1 to C-3
Var, x
Variable, x
Variables, 2-4 to 2-8
  assigning values to, 2-13
  dummy, 5-22
  initializing, 9-4
  integer, 2-5 to 2-6
  numeric, 2-5
  preserving through CHAIN, 8-3
Variables (Cont.)
  simple numeric, 2-5
  string, 2-5, 2-6 to 2-7
  subscripted, 2-7 to 2-8, 2-14
Virtual array files, 6-1, 6-8 to 6-10
  closing, 6-10
  compared to arrays in memory, 6-8
  compared to sequential files, 6-8
  dimensioning, 6-9

Whole numbers, 2-3, 5-6
Writing files, 6-5

Zone, printing, 3-9 to 3-12

BASIC-11
Language Reference Manual
DEC-11-LIBBB-A-D

READER'S COMMENTS

NOTE: This form is for document comments only. Problems with software should be reported on a Software Performance Report (SPR) form.

Did you find errors in this manual? If so, specify by page.

_____
_____
_____
_____
_____
_____

Did you find this manual understandable, usable, and well-organized? Please make suggestions for improvement.

_____
_____
_____
_____
_____
_____

Is there sufficient documentation on associated system programs required for use of the software described in this manual? If not, what material is missing and where should it be placed?

_____
_____
_____
_____
_____
_____

Please indicate the type of user/reader that you most nearly represent.

- ☐ Assembly language programmer
- ☐ Higher-level language programmer
- ☐ Occasional programmer (experienced)
- ☐ User with little programming experience
- ☐ Student programmer
- ☐ Non-programmer interested in computer concepts and capabilities

Name _____ Date _____

Organization _____

Street _____

City _____ State _____ Zip Code or Country _____

If you require a written reply, please check here. ☐

Please cut along this line.

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

RT/C SOFTWARE PUBLICATIONS ML 5-5/E45
DIGITAL EQUIPMENT CORPORATION
146 MAIN STREET
MAYNARD, MASSACHUSETTS 01754

Do Not Tear - Fold Here

Cut Along Dotted Line

BASIC-11
Language Reference Manual
DEC-11-LIBBB-A-D

READER'S COMMENTS

NOTE: This form is for document comments only. Problems with software should be reported on a Software Performance Report (SPR) form.

Did you find errors in this manual? If so, specify by page.

Did you find this manual understandable, usable, and well-organized? Please make suggestions for improvement.

Is there sufficient documentation on associated system programs required for use of the software described in this manual? If not, what material is missing and where should it be placed?

Please indicate the type of user/reader that you most nearly represent.

- ☐ Assembly language programmer
- ☐ Higher-level language programmer
- ☐ Occasional programmer (experienced)
- ☐ User with little programming experience
- ☐ Student programmer
- ☐ Non-programmer interested in computer concepts and capabilities

Name _____ Date _____

Organization _____

Street _____

City _____ State _____ Zip Code or Country _____

If you require a written reply, please check here. ☐

Please cut along this line.

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

RT/C SOFTWARE PUBLICATIONS ML 5-5/E45
DIGITAL EQUIPMENT CORPORATION
146 MAIN STREET
MAYNARD, MASSACHUSETTS 01754

Do Not Tear - Fold Here

Cut Along Dotted Line

digital

digital equipment corporation

Printed in U.S.A.