pdp11/DECSYSTEM20

BASIC-PLUS-2
Language Manual

Order No. AA-0153A-TK

digital

DECSYSTEM20/pdp11

BASIC-PLUS-2
Language Manual

Order No. AA-0153A-TK

July 1977

This document describes the elements of the BASIC-PLUS-2 language that are common to the DECSYSTEM-20 and the PDP-11. System dependent information is found in the appropriate user's guide.

This is a new document.

OPERATING SYSTEM AND VERSION:	TOPS-20	V02
	RSTS/E	V06B
	RSX-11M	V03
	IAS	V02
SOFTWARE VERSION:	BASIC-PLUS-2	V01

To order additional copies of this document, contact the Software Distribution Center, Digital Equipment Corporation, Maynard, Massachusetts 01754.

digital equipment corporation · maynard, massachusetts

First Printing, July 1977

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.

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

CONTENTS (CONT.)

CONTENTS (CONT.)

CONTENTS (CONT.)

CONTENTS (CONT.)

FIGURES

TABLES

PREFACE

Because BASIC-PLUS-2 has been implemented on more than one system, this manual describes only the language elements and statement syntax of BASIC-PLUS-2. It does not discuss system-specific information. In order to use BASIC-PLUS-2 on your particular system, you must also read one of the following:

RSX/IAS BASIC-PLUS-2 User's Guide
RSTS/E BASIC-PLUS-2 User's Guide
DECSYSTEM-20 BASIC-PLUS-2 User's Guide

This document is not a tutorial manual. If you are unfamiliar with the BASIC language, you should also read Digital's *Introduction to BASIC.*

In the following manual (*BASIC-PLUS-2 Language Manual*) Chapters 1 through 5 describe the essential parts of a BASIC-PLUS-2 program. Almost any problem can be solved with the statements and features in these chapters. Chapters 6 through 11 provide the advanced features of BASIC-PLUS-2, which allow the BASIC language to be a useful tool for the more experienced programmer. For a quick reference to the BASIC-PLUS-2 language elements, refer to Appendix A.

Throughout the manual, BASIC-PLUS-2 and BASIC are used interchangeably. The User's Guide referred to is the *BASIC-PLUS-2 User's Guide* for your system.

Conventions Used in This Manual

Symbol	Represents
CTRL/X	The CTRL key and another key pressed simultaneously (i.e., CTRL/C).
(RET)	The RETURN key (carriage-return/line feed).
(TAB)	The TAB key (CTRL/I on some terminals).
(SP)	The SPACE bar.
[[ ]]	Special brackets indicating optional information that can be omitted from a command string.
{ }	Braces indicating a choice. Choose one from the enclosed.
Lower-case letters	Lower-case characters in a command string indicate variable information to be supplied by you.
UPPER-CASE LETTERS	Upper-case characters in a command string indicate fixed (literal) information that you must enter as shown.
Examples	On the DECSYSTEM-20, when you list a BASIC-PLUS-2 program (or line in a program) from storage leading zeroes are added to the line numbers (e.g., 10 becomes 00010). All examples were produced on the DECSYSTEM-20. Most of them were stored, recalled, and printed. Therefore, the line numbers in these examples contain leading zeroes.

Conventions Used in This Manual (Cont.)

Symbol	Represents
Contrasting Colors	Red — Where examples contain both user input and computer output, the characters you type are in red; the characters the computer prints are in black. Gray — The text shaded in gray indicates the features available only on the DECSYSTEM-20.

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 John G. Kemeny and Thomas E. Kurtz. It is one of several programming languages used to translate symbolic language programs into machine language. Because the BASIC language is composed of easily understood statements and commands, it is one of the simplest programming languages to learn.

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

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

Some of the special features of BASIC-PLUS-2 are:

Virtual Arrays – Section 9.3
Record File I/O – Section 10.1
Extensive String Support – Sections 2.2.3 and 2.3.3
Full Matrix Package – Section 8.1
Long Variable Names – Section 2.3
IF . . . THEN . . . ELSE – Section 5.1.3
ON ERROR – Section 5.6
Statement Modifiers – Section 6.1
User-Defined Functions – Section 7.6.2
Multi-Statement Lines – Section 1.3.1
Multi-Line Statements – Section 1.3.1

1.2 STRUCTURE OF A BASIC PROGRAM

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

1.2.1 Character Set

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

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

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

BASIC translates characters in the following manner:

1. Letters A through Z – BASIC treats the same alphabetic in upper-case and lower-case as the same character, e.g., I is the same as i.
2. Non-printing characters (e.g., BEL) – BASIC interprets the code during input, prints a warning message, then ignores them during execution.
3. NUL characters – BASIC interprets the code during input, prints a warning message, and ignores them during execution.

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 characters are processed.
3. All null characters are processed.

BASIC also ignores all characters in a REMARK during execution. See Section 1.4.

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 Control/U (^U) are described fully in the User's Guide for your system.

1.2.2 Line Format

The format of a program line is as follows:

line number	keyword	statement	line terminator
00010	PRINT	R = SQR(X^2 + Y^2)	(RET)

Most lines in a BASIC program must begin with a number. (Continuation lines are another matter. See Section 1.3.1.) This number must be a positive integer within the range of 1 to a limit set by your system. Refer to the User's Guide for this information. 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 leading and trailing spaces) have no effect on the number. However, you cannot have embedded spaces within a line number. For example, these numbers are the same to BASIC:

```
00010
10
```

But this number is illegal:

```
0 1 0
```

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

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

can also be typed:

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

Both lines are the same to BASIC.

However, embedded spaces in line numbers, keywords (Section 1.3), or variable names are illegal. For example, BASIC rejects the previous statement if you type it as follows:

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

1.3 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, and operators. These statements divide into two major groups: executable statements and non-executable statements.

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

This is acceptable `10 PRINT A`

This is not `10 PRINTA`

Certain keywords consist of two or more English words. Some keywords allow an optional space between words and some keywords require a space between words. Table 1-1 lists these keywords.

Table 1-1 Keywords and Spaces

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

Statement keywords are reserved, and therefore, cannot be used as variable names (see Section 2.3). Appendix C contains a complete list of reserved keywords.

1.3.1 Single Statement, Multi-Statement, and Continuation Lines

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

A single statement line consists of:

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

This is a single statement line:

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

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

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

The line number labels the first statement in a 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.

The rules for structuring a multi-statement line are:

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

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

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

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

Consider the following example:

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

You can continue any statement including an IMAGE statement (see Section 4.5). However, you cannot continue a comment field (see Section 1.4).

1.4 DOCUMENTING PROCEDURES – THE REM STATEMENT AND THE COMMENT

BASIC allows you to document your methods, insert notes and comments, or leave yourself messages in the source program. This type of documentation is known as a remark or comment. There are two ways of inserting comments within a BASIC source program:

1. With the REM statement
2. With the comment field (!)

The REM statement has the following format:

REM comment

where:

comment is anything you want to write.

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

The REM statement can be 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.

BASIC ignores anything in a line following the keyword REM including a backslash (\). The only character that ends a REM statement is a line terminator. Therefore, a REM statement should be the only statement on the line or the last statement in a multi-statement line.

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

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

```
LISNH
00010 REM SGN FUNCTION EXAMPLE
00020 READ A,B,C
00030 PRINT "SGN(A)=";SGN(A),"SGN(B)=";SGN(B),"SGN(C)=";SGN(C)
00040 GOTO 10
00050 DATA -7.45,.27,0
00060 END
```

Line 40 sends BASIC back to line 10. (See Section 5.1.1 for the GOTO statement.) BASIC ignores the comment on line 10 and continues execution at line 20.

Remember that BASIC prints the remakrs on the terminal only when you list the source program. (See the User's Guide for the LIST Command.)

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

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

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

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

Note that the DATA statement and the IMAGE statement (Sections 3.1.3 and 4.5 respectively) cannot have comment fields. Each must be the only statement on its respective line.

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

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

BASIC does not generate any code for the comment but does continue the statement.

CHAPTER 2
ELEMENTS OF BASIC-PLUS-2

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-PLUS-2.

2.2 CONSTANTS

There are three types of constants in BASIC:

1. Numeric (real numbers, also called floating point numbers)
2. Integer (whole numbers)
3. String (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.

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

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

BASIC accepts numeric constants within a certain range. Refer to your User's Guide for this information.

If you type a numeric constant into the source text that is outside the range of numeric values representable by the hardware, BASIC prints a fatal error message to that effect. 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:

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

where:

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

This method of mathematical shorthand is called E notation or floating point notation. It is BASIC's way of representing scientific notation. To use this format, append the letter E to the number. Then follow the E with an optionally signed constant (see Section 2.2.2). The constant is the exponent. It can be 0 but never blank.

Thus you can type:

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

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

5.2041E-3

BASIC interprets it as .0052041.

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

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 single precision 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

2.2.2 Integer Constants

An integer constant is a whole number (no fractional part) written without a decimal point. 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

In BASIC, you can type integer constants within the range specified by your system. See the User's Guide for this information. If you specify a number outside the range, BASIC prints a fatal error message telling you to 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. Include both the starting and ending delimiters when typing a string constant in a source program. These delimiters must be of the same type (both double quotation marks or both single quotation marks).

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

" DIGITAL " is not the same as "DIGITAL"

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.

```
00010 PRINT "DIGITAL"
00020 END

READY
RUNNH
DIGITAL
```

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

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

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

Here are some examples of string constants:

"This Is a String Constant."
'SO IS THIS.'
"TONY'S TENNIS RACKET"

The following are examples of invalid string constants:

"WRONG TERMINATOR'
'SAME HERE"
"NO TERMINATOR

2.3 VARIABLES

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. Numeric
2. Integer
3. String

All statement keywords are reserved (see Appendix C) and, therefore, cannot be used as variable names. The following sections describe the formation of legal variable names.

2.3.1 Numeric Variables

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

1 letter
29 optional characters

Do not embed spaces between characters. The following are numeric variables:

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

The following are not legal numeric variables:

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

Before program execution, BASIC sets all numeric variables to 0 except for those in virtual array, MAP, or COMMON declarations (Sections 9.3, 10.14, and 11.2.1 respectively). 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. Good programming practice dictates that you initialize all variables at the beginning of the program.

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 space is reserved for the storage of a whole number (no fractional part).

You name an integer variable with a single letter followed by 29 optional characters consisting of letters, digits, or periods and terminate the name with a percent sign (%). Therefore, the maximum length of an integer variable name is 31 characters:

1 letter
29 optional characters
1 percent sign (%)

(No embedded spaces are allowed.) The following are integer variables:

```
ABCDEFG%        C.8%
B%              D6E7%
```

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 (real number) is assigned to an integer variable, BASIC drops the fractional portion of the value. The number is not rounded to the nearest integer; it is truncated. Consider the following example:

```
B% = -5.7
```

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

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

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

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

The following are examples of string variables:

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

These are not string variables:

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

Strings have a value and a length. BASIC initializes all string variables (unless in a virtual array, MAP, or COMMON area) 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 a limit set by your system. See the User's Guide for this information. String variables can also be declared implicitly just by appearing in a program.

Note that a simple numeric variable, an integer variable, and a string variable that begin with the same alphanumeric characters 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, integer, or string variable with one or two subscripts appended to it. The subscripts can be any positive expression type: a constant or a variable (integer or numeric), a letter or symbol, or any combination of these. BASIC converts non-integer expressions to integer by truncating the fraction. The value of the subscript can be 0 to a maximum defined by your system. Refer to the User's Guide for this parameter.

The subscript in a subscripted variable is a pointer to a specific location in a list or table in which a value is stored. (See Section 2.6 for more information on lists and tables.) You designate the pointer with either one or two subscripts enclosed by parentheses. If there are two subscripts, separate them with a comma. The value stored can be numeric, integer, or string data.

To name a subscripted variable, start with a 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:

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)

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(8,6) two subscripts

2.4 FORMING EXPRESSIONS

An expression can be numbers, strings, constants, variables, functions (Section 2.4.6), array references (Section 2.6), or any combination of these, separated by any of the following:

1. Arithmetic operators
2. Relational operators
3. String operators
4. Logical operators

2.4.1 Arithmetic Expressions

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

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

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

A%+B% = an integer expression
G3*M5 = a numeric expression
A$+D$ = a string expression

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

A*B% = a numeric expression

6.83 + 5% = 34.15

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

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

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
**	A**B	Calculate A to the power B

2.4.2 String Expressions

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

Consider the following example:

```
00010 C$ = "GOOD" + "BYE"
00020 PRINT C$
00030 END
```

During execution, BASIC prints the following:

```
GOODBYE
```

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

2.4.3 Relational Expressions

A relational operator is a symbol used to compare the value of one variable or expression to another variable or expression within a BASIC program, thus creating a relational expression. As explained in Section 5.1.3, one of the uses of relational expressions is with the IF-THEN-ELSE statement to create conditional transfers.

If a relational operator is used on the right side of an assignment, its value is -1 if the relation is true, and 0 of the relation is false.

NOTE

It is illegal to compare a numeric or integer expression to a string expression using a relational operator.

Table 2-3 provides examples of relational operators and their meaning.

Table 2-3 Relational Operators

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

2.4.4 Comparing Strings Using Relational Operators

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 B for the ASCII Table.) The comparison is made, character by character, left to right, by ASCII value until BASIC finds a difference in value.

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

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

When BASIC executes line 30, it compares string A$ with B$ to determine if A$ occurs first in alphabetic sequence. In this case, it does, and the program transfers control to line 60 (see Section 5.1 for transferring 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 to be placed in alphabetical order. BASIC compares the first character in each string, A and D. The letter A precedes the letter D in the ASCII table; therefore, string A$ precedes string B$ in alphabetic sequence. If the first two characters are equal, BASIC proceeds to the second two characters, until a difference is found. For example:

```
ABC
AEF
```

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

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

Table 2-4 String Relational Operators

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

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

Consider the following example:

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

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

```
RUNNH
ANTONY
```

2.4.5 Logical Expressions

A logical expression consists of either one operand preceded by a logical operator or two operands separated by a logical operator. Logical expressions are used in statements like the IF-THEN-ELSE statement (Section 4.1.3) where a condition is tested to determine subsequent operations within the program. The operands, in this case, are usually relational expressions. Logical expressions can also be used with integer data. However, logical operations on strings are illegal.

BASIC determined whether the condition is true or false by testing the bit-wise result of the logical expression for non-zero and zero, respectively. (That is, a non-zero result is true, and a zero result is false.) Notice that any non-zero value is assumed to be true. BASIC supplies the value -1 for true when it evaluates a logical or relational expression but accepts any non-zero value when performing a test. Therefore:

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

Both A% and B% are true, e.g., A% = 10 and B% = -11. Logical operators are reserved words. See Appendix C.

Table 2-5 provides a list of logical operators and their meaning.

Table 2-5 Logical Operators

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

Logical expressions are legal wherever numeric expressions are legal in BASIC. However, both operands must be integers.

The following tables are called truth tables. They describe graphically the results of the above logical operations on a bit-by-bit basis. Every possible combination of bits for A% and B% is given.

Table 2-6 Truth Tables

A%	NOT A%
0	1
1	0

A%	B%	A% OR B%
0	0	0
0	1	1
1	0	1
1	1	1

A%	B%	A% AND B%
0	0	0
0	1	0
1	0	0
1	1	1

A%	B%	A% EQV B%
0	0	1
0	1	0
1	0	0
1	1	1

A%	B%	A% XOR B%
0	0	0
0	1	1
1	0	1
1	1	0

A%	B%	A% IMP B%
0	0	1
0	1	1
1	0	0
1	1	1

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

2.4.6 Functions

Library functions are an integral part of the BASIC language. A function can be accessed in a program by stating its name. A sequence of instructions is then performed automatically.

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

```
00010  PRINT SQR(49)
00020  END

READY
RUNNH
 7
```

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

Functions are fully described in Chapter 7. For a complete list of library functions, refer to Appendix A.

2.4.7 Evaluating Expressions

BASIC evaluates expressions according to operator precedence. Each arithmetic, relational, and string 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. Parentheses may be used to alter the order of precedence.

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

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

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

Table 2-7 lists all operators in the order BASIC evaluates them:

Table 2-7 Operator Precedence

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

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

All relational operators are on the same precedence level.

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

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

BASIC evaluates this expression in five ordered steps:

1. 15^2 = 225 Exponentiation (left-most expression)
2. 12^2 = 144 Exponentiation
3. 35*8=280 Multiplication
4. 225+144= 369 Addition
5. 369-280= 89 Subtraction

2.5 ASSIGNING VALUES TO VARIABLES – THE LET STATEMENT

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

```
LET variable(s) = expression
                or
  variable(s) = expression
```

where:

expression can be a string, numeric, or logical expression. (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. BASIC evaluates the expression from left to right and assigns the values from right to left. Here is an example:

```
10  LET A = 482.5
```

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

```
10  A = 482.5
```

BASIC also evaluates any 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, you can assign a value to more than one variable at a time, as in the following example:

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

This statement has the same effect as:

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

BASIC also converts the data type of the value to the data type of the associated variable, i.e., integer or real. In the following example,

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

is the same as

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

BASIC assigns the values from right to left. Refer to Section 2.3 for a description of variables.

Moreover, if you reference an array element as a variable (see Section 2.6) BASIC calculates the value of the subscript before assigning values to the other variables in the list. For example, the following assigns the value 5 to the list element A(2). Then the value of I changes to 5:

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

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 an error message on the terminal.

The following is an example of a string assignment:

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

RUNNH
HELLO
```

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

Basic automatically reserves storage space for arrays with maximum subscripts of 10, i.e., A(10) and A(10,10). If you require a larger amount of reserved space, define the dimensions with the DIM statement (see Section 2.6.1). Conversely, if you do not require the space BASIC supplies by default, save the space for your program by reserving a smaller area with the DIM statement.

When you establish the size of the array, BASIC stores the dimensions in a particular area for future reference. BASIC starts counting elements from 0, not 1; therefore, you have an additional element for a list and an additional row and column for a table.

For example, dimensioning the array A(6) gives you seven storage areas in the list, not six:

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

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

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

Figure 2-1 Array B

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

Remember that it is possible to use the same alphanumerics to name both a simple variable and an array within the same program. But using the same name for two arrays with a different number of subscripts is illegal within the same program, i.e., A(5) and A(3,4).

2.6.1 The DIM Statement

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

The DIM statement has the following format:

DIM subscripted variable(s)

or

DIMENSION subscripted variable(s)

where:

subscripted variable(s) is one or more numeric, integer, or string variables separated with commas (see Section 2.3.4). Subscripts must be numeric or integer constants.

Each subscripted variable name represents a distinct list or table. The subscripts must be constants and can range from 0 to the system defined maximum. Because BASIC automatically reserves storage for arrays with maximum subscripts of 10, i.e., A(10) and B(10,10), use the DIM statement to reserve storage for lists with more than 11 elements and matrices with more than 121 elements.

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

Arrays are stored as if the rightmost subscript varied the fastest. For example:

```
10 DIM A(1,5)
```

provides 12 contiguous areas of storage.

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:

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

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
```

Notice that a numeric or integer variable has a value of 0 until you assign it another value. (A string variable is considered a null string.)

You can also dimension string arrays with the DIM statement:

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

READY
RUNNH
 ? H,E,L,L,O

H
E
L
L
O
```

CHAPTER 3
INPUT AND OUTPUT TO THE TERMINAL

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 source program.
3. The file statements – require that you manipulate files outside the main program. See Chapter 9 for information on file input and output.

3.1.1 The 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 variable(s)

where:

variable(s) is one or a list of 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 space, a question mark (?), and a space. BASIC then waits for you to type one value for each variable requested in the INPUT statement. When there is more than one variable requiring a value, separate each value with a comma. Press a line terminator after you finish typing all the values.

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

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

READY
RUNNH
 ? 5,6,7
```

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

A=5
B=6
C=7

You must supply the same number of values as there are variables in the INPUT request. If you do not type enough data, BASIC lets you know by printing the message "INSUFFICIENT DATA AT LINE n" and another question mark (?) when you press the RETURN key.

```
00010 INPUT A,B
00020 END

READY
RUNNH
 ? 5

? 59  Insufficient data at line 00010 of MAIN PROGRAM
 ? 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.

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

READY
RUNNH
 ? 5,6,7,8

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

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 in answer to the question mark. If you include quotation marks, be sure to type both beginning and ending delimiters. If you forget the end quotation mark, BASIC reads the rest of the line as the entire string. You will also receive an error message.

Including a string constant in an INPUT statement allows you to see the results of your computations. You must separate the string constant from the variable list with a comma or semicolon. For example:

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

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

READY
```

NOTE

The INPUT # statement (see Section 9.3.1) 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
```

The INPUT statement with a string constant but without a variable list has the same effect as the PRINT statement, Section 3.2. For example:

```
10 INPUT "BASIC-PLUS-2"
RUNNH
BASIC-PLUS-2
READY
```

3.1.2 The INPUT LINE and LINPUT Statements

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

INPUT LINE string variable(s)

LINPUT string variable(s)

where:

A space	is required between the keywords INPUT and LINE.
LINPUT	is one word.
All variables	must be string variables in INPUT LINE and LINPUT statements.

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

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

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

READY
```

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

Both INPUT LINE and LINPUT can reference a terminal or a terminal-format file. See Chapter 9 for more information.

3.1.3 READ, DATA, and RESTORE (RESET) 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 variable(s)

where:

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

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

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

The DATA statement has the following format:

DATA constant(s)

where:

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

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

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

A READ statement is not legal without at least one DATA statement. However, you can have more than one DATA statement. Without a READ statement, BASIC ignores the DATA.

```
00010 READ A,B,C,D,E,F
00020 PRINT C
00030 DATA 17,25,30
00040 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. Each list of constants in a DATA statement is local to a program or subprogram (see Section 11.1) and must be referenced by line number, not by order of appearance in a program.

If you build your READ statement with more variables than you include in the data block, you receive a warning message from BASIC.

You will receive error messages if:

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

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

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

READY
RUNNH
8.25

READY
```

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

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

BASIC assigns values as follows:

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

READY
```

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 keyword RESET performs the same function.) 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:

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

The READ statement in line 10 reads the first three values in the DATA statement, 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 statement on line 30 also reads the first three values. BASIC reads these values as though for the first time:

E=6
F=3
G=4

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

E=7
F=9
G=2

The RESTORE statement affects only the program or subprogram (see Chapter 11) in which it is contained. Also, if you have no DATA statements in your program, RESTORE has no effect.

NOTE
The RESTORE # statement (Chapter 9) is used to restore files to their beginning.

3.2 CHECKING OUTPUT – THE PRINT STATEMENT

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

PRINT [[expression(s)]]

where:

expression(s) can be one or more numeric or string elements separated with 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.)

You can include blank lines in your output for readability. Using the PRINT statement without arguments causes a blank line to appear in the output:

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

READY
RUNNH
THIS EXAMPLE LEAVES A BLANK LINE

BETWEEN TWO LINES.

READY
```

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, calculates expressions and prints the results. For example:

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

READY
RUNNH
 100
```

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

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

READY
RUNNH
A + B
```

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:

```
00010 PRINT "WHAT ARE YOUR VALUES OF X,Y,Z"
00020 INPUT X,Y,Z
00030 LET R = SQR(X^2 + Y^2 + Z^2)
00040 PRINT "THE RADIUS VECTOR EQUALS";
00045 PRINT R
00050 END
```

When you run this program, BASIC prints:

```
RUNNH
WHAT ARE YOUR VALUES OF X,Y,Z
 ? 25,40,50
THE RADIUS VECTOR EQUALS 68.73864

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 separates the string from the variable name. Placing a semicolon or a comma after the string makes BASIC print the value 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
```

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 (;).

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

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

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

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

```
00010 A = 5\B = 10
00015 PRINT "FIRST ZONE",,"THIRD ZONE"
00020 PRINT A,,B
00030 END

READY
RUNNH
FIRST ZONE                    THIRD ZONE
 5                             10
```

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 tight packing of the print line. Note that whenever BASIC prints a number it is preceded by a space or a minus sign and followed by a space.

```
00010 PRINT 10;20
00020 END

READY
RUNNH
 10  20
```

BASIC does not print a space before strings:

```
00010 PRINT 10;20
00020 END

READY
RUNNH
 10  20
```

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

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:

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

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

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

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

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

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

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

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

READY
RUNNH
FIRST ZONE                    THIRD ZONE                    FIFTH ZONE
```

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

3.2.2 Output Format for Numbers and Strings

BASIC prints numbers and strings according to a specific format. Strings are printed exactly as you type them in with no leading or trailing spaces. (Quotation marks are not printed unless delimited by another pair.)

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

READY
RUNNH
PRINTING "QUOTATION" MARKS
```

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

```
00010 PRINT -1
00020 PRINT 25;50
00030 END

READY
RUNNH
-1
 25  50
```

Leading and trailing spaces can be added within the quotation marks by using the keyboard space bar.

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

$.01 \leq n < 999999$

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.

3.2.3 The TAB Function

Another method of positioning the terminal printer is to use the TAB function in conjunction with the PRINT statement. (Refer to Chapter 7 for information on functions.)

This function has the following format:

```
PRINT TAB(n);
```

where:

n is an expression indicating the desired printing position. BASIC evaluates the expression and truncates the result to an integer.

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

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

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

The following examples contain several TAB functions in conjunction with one PRINT statement:

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

READY
RUNNH
NAME            ADDRESS         PHONE NO.
```

Without tabs 15 and 30, BASIC would print

```
NAMEADDRESSPHONE NO.
```

Here is an example of printing numbers:

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

READY
RUNNH
 100            29              35
```

Column 0 Column 15 Column 30

Notice that semicolons act as separators in the preceding example.

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

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

READY
RUNNH
 100                                        200
 300
 100                                  200         300
```

The commas move the printer to the next zone, then BASIC executes the TAB function.

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

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

   TAB(n) = SPACE$(n)

   The SPACE$ function allows you to add spaces in a string, see Section 7.3.6.
2. If a PRINT statement is executed with TAB, TAB(n) is a string of spaces whose length is equal to the number of spaces between the last print position and n.

CHAPTER 4
FORMATTED OUTPUT — THE PRINT USING STATEMENT

4.1 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 thus enabling you to create formatted lists, tables, reports, and forms.

The following examples print a series of numbers. One program uses the PRINT statement and the other uses the PRINT USING statement.

PRINT

```
00010 PRINT 1
00020 PRINT 100
00030 PRINT 1000000
00040 PRINT 100.3
00050 PRINT .0123456

READY
RUNNH
 1
 100
 1E+6
 100.3
 0.0123456

READY
```

PRINT USING

```
00010 PRINT USING "#######.##",1
00020 PRINT USING "#######.##",100
00030 PRINT USING "#######.##",1000000
00040 PRINT USING "#######.##",100.3
00050 PRINT USING "#######.##",.0123456

READY
RUNNH
      1.00
    100.00
1000000.00
    100.30
      0.01

READY
```

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

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 sign, asterisks, dollar sign, 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, or the line number of an IMAGE statement. It is called the format string. If it is a string constant, it must be enclosed in double quotation marks, not single quotation marks.

list — contains the items to be printed. Note that a comma or semicolon separating items in the list has no effect on the output. However, a comma or semicolon placed after the last item in the list does inhibit the movement of the terminal printer.

Consider the following example.

```
00010 PRINT USING "HI 'LLLLL YOU WEIGH ###.# LBS.","PAUL",145
00020 END
```

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 ###.# and it corresponds to the second data item, 145. When BASIC prints the line, it prints each data item in the position and format specified by the field. The rest of the format string, namely "HI YOU WEIGH LBS." is the printed message. The output of this example is:

```
RUNNH
HI PAUL   YOU WEIGH 145.0 LBS.
```

The way to write format strings is described throughout this chapter and is summarized in Section 4.4.

4.2 PRINTING NUMBERS WITH PRINT USING

4.2.1 Specifying the Number of Digits

With PRINT USING, you can specify the number of places reserved for digits in a field with a corresponding amount of number signs (#).

For example:

```
00010 PRINT USING "###",123
00020 PRINT USING "#####",12345

READY
RUNNH
123
12345
```

If there are not enough digits to fill the field specified, BASIC prints spaces before the first digit. For example:

```
00010 PRINT USING "#####",1
00020 PRINT USING "#####",10,
00030 PRINT USING "#####",1709
00040 PRINT USING "#####",12345

READY
RUNNH
    1
   10 1709
12345
```

Note that spaces are printed before the number so that the entire field is filled. The number sign indicates where BASIC prints a space. Placing a comma or semicolon after the list item (line 20) causes the next item to be printed on the same line.

BASIC rounds numbers printed with PRINT USING. For example:

```
00010 PRINT USING "###",126.7
00020 PRINT USING "#",5.9
00030 PRINT USING "#",5.4

READY
RUNNH
127
6
5
```

4.2.2 Specifying the Location of the Decimal Point

You can reserve any number of digits on both sides of the decimal point by placing a decimal point in the number sign field. BASIC always prints the digits to the right of the decimal point, even if they are zeros. Consider the following example:

```
00010 PRINT USING "##.###",5.72
00020 PRINT USING "##.###",39.3758
00030 PRINT USING "##.###",26

READY
RUNNH
 5.720
39.376
26.000
```

Note that BASIC prints spaces to the left of the decimal point, as necessary, but prints zeros to the right of the decimal point. Also note that 39.3758 is rounded to 39.376.

If there is more than one number sign to the left of the decimal point, at least one digit is printed to the left of the decimal point. If there is only one # to the left of the decimal point and the number is negative and less than 1, BASIC prints the minus sign to the left of the decimal point instead of a zero.

For example:

```
00010 PRINT USING "###.#",.99
00020 PRINT USING "###.#",.1
00030 PRINT USING "#.#",-.1

READY
RUNNH
  1.0
  0.1
-.1
```

4.2.3 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 ignores the format specified by the field. After BASIC prints the number, it completes the rest of the PRINT USING statement, in the usual manner. Consider the following example:

```
00010 PRINT USING "###.##",256.786
00020 PRINT USING "##.##",256.786

READY
RUNNH
256.79
% 256.786
```

The number at line 10 is printed correctly (with rounding). In line 20, there are only 2 #'s to the left of the decimal point; therefore, the number will not fit. The number is printed in the usual PRINT statement format, with a space before and after it. But in this case, the number is preceded by a percent sign.

Be sure to enter one number sign for every number that will print on the left side of the decimal point. Add one more number sign in case the number is negative. (For another method of reserving a place for the minus sign, see Section 4.2.4.1.)

Field	There are enough places for	But not enough places for
###.##	100.569	%-100.569
.####	.1258	%-.12579

A number can also be larger than its field, because rounding increases the number of places needed. For example:

```
00010 PRINT USING ".###",.999
00020 PRINT USING ".##",.999
00030 PRINT USING "#.##",.999

READY
RUNNH
.999
% 0.999
1.00
```

4.2.4 Printing Numbers With Special Symbols

4.2.4.1 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 number. You must use the trailing minus sign to print a number in an asterisk fill or floating dollar sign field (see Sections 4.2.4.2 and 4.2.4.3).

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

```
00010 PRINT USING "###.##",-10.54
00020 PRINT USING "###.##",10.54

READY
RUNNH
-10.54
 10.54
```

Fields with Trailing Minus Signs

```
00010 PRINT USING "##.##-",-10.54
00020 PRINT USING "##.##-",10.54

READY
RUNNH
10.54-
10.54
```

4.2.4.2 Printing Numbers in Asterisk Fill Fields – To print a number with asterisks (*) filling up any blank spaces before the first digit, start the field with two asterisks. For example:

```
00005 PRINT USING "**##.##",1.2
00010 PRINT USING "**##.##",27.95
00020 PRINT USING "**##.##",107
00030 PRINT USING "**##.##",1007.5

READY
RUNNH
***1.20
**27.95
*107.00
1007.50
```

Note that the asterisks reserve two places as well as cause asterisk fill.

To print a negative number in an asterisk fill field, specify a trailing minus sign in the field. For example:

```
00010 PRINT USING "**##.##-",27.95
00020 PRINT USING "**##.##-",-107
00030 PRINT USING "**##.##-",-1007.5

READY
RUNNH
**27.95
*107.00-
1007.50-
```

If you attempt to print a negative number in an asterisk fill field without a trailing minus sign, BASIC prints a fatal error message.

4.2.4.3 **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 a negative number is desired, end the field with a trailing minus sign. Consider the following example:

```
00010 PRINT USING "$$###.##",77.44
00020 PRINT USING "$$###.##",304.55
00030 PRINT USING "$$###.##",2211.42
00040 PRINT USING "$$###.##-",-125.6
00050 PRINT USING "$$###.##",127.82

READY
RUNNH
 $77.44
$304.55
% 2211.42
$125.60-
$127.82
```

Note that $$ reserves places for the dollar sign and one digit; the $ is always printed. Contrast this with the asterisk fill field where asterisks are printed only if there are leading spaces.

If you attempt to print a negative number in a dollar sign field without a trailing minus sign, BASIC prints a fatal error message.

4.2.4.4 **Printing Numbers With Commas** – To insert commas in a number, place a comma anywhere in the format field, to the left of the decimal point (if present). There is one space allocated in the output field for each comma, regardless of where the comma appears in the format field. BASIC then prints a comma every third digit to the left of the decimal point. Commas, in the format field, to the right of the decimal point, are treated as literals. If there is no digit to be printed to the left of the comma, BASIC does not print the comma. For example:

```
00010 PRINT USING "##,###",10000
00020 PRINT USING "##,###",759
00030 PRINT USING "$$#,###.##",25694.3
00040 PRINT USING "**#,###",7259
00050 PRINT USING "####,#.##",25239
00060 END

READY
RUNNH
10,000
   759
$25,694.30
**7,259
25,239.00
```

4.2.5 **Printing Numbers In E (Exponential) Format**

To print a number in E (exponential) format, place four circumflexes (^^^^ also called up-arrows) at the end of the field. The ^^^^ reserve space for the capital letter E followed by a space or minus sign (which indicates a positive or negative exponent, respectively) and then the 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 left-most place and the exponent is adjusted to compensate.

Consider the following example:

```
00010 PRINT USING "###.##^^^^", 5
00020 PRINT USING "###.##^^^^", 1000

READY
RUNNH
500.00E-02
100.00E+01
```

If you use fewer than 4 carets, the number is not printed in E format but the carets are printed as a literal. If you use more than 4 carets, the number prints in E format but the extra carets are also printed. For example:

```
00010 PRINT USING "###.##^^^", 5
00020 PRINT USING "###.##^^^^^",5

READY
RUNNH
  5.00^^^
500.00E-02^
```

Exponential format cannot be used with asterisk fill, floating dollar sign, or trailing minus formats.

4.2.6 Fields That Exceed BASIC's Accuracy

If a field contains more places than there are digits of accuracy, BASIC prints zeros in all the places following the last significant digit. See the User's Guide for the number of digits of accuracy in your system.

4.3 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, centered format or extended format. String fields start with a single quotation mark ('). The single quotation mark is optionally followed by a contiguous series of uppercase L's, R's, C's, or E's representing left-justified, right-justified, centered, and extended string fields, respectively.

If a string is larger than the string field, BASIC prints as much of the string as fits and ignores the rest. The only exception is extended fields, in which case BASIC prints the entire string.

4.3.1 One-Character String Fields

A string field consisting of only a single quotation mark or a single exclamation point is a 1-character string field. BASIC prints the first character of the string expression corresponding to a 1-character string field and ignores all following characters. For example:

```
00010 PRINT USING "'","ABCDE"

READY
RUNNH
A
```

4.3.2 Printing Strings in Left-Justified Format

If you specify a left-justified field, BASIC prints the string starting at the left-most position. If there are any unused places, BASIC prints spaces after the string. If there are more characters than places, BASIC truncates the string and does not print the excess characters.

A left-justified field is composed of a backslash or a single quotation mark followed by a series of capital L's. For example:

```
00010 PRINT USING "'LLLLLL","ABCD"
00020 PRINT USING "'LLLL","ABC"
00030 PRINT USING "'LLLL","12345678"

READY


ABCD
ABC
12345
```

4.3.3 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 right most place of the field. If there are any unused places before the string, BASIC prints spaces to fill the field.

A right-justified field is composed of a single quotation mark followed by a series of capital R's. For example:

```
00010 PRINT USING "'RRRRRR","ABCD"
00020 PRINT USING "'RRRRRR","A"
00030 PRINT USING "'RRRRRR","XYZ"

READY

  ABCD
     A
   XYZ
```

If there are more characters than places, BASIC left-justifies the string and does not print the excess characters.

4.3.4 Printing Strings In Centered Fields

If you specify a centered field, BASIC prints the string so that the center of the string is in the center of the field. If the string cannot be exactly centered, such as a 2-character string in a 5-character field, BASIC prints the string one character off center to the left.

A centered field is composed of a single quotation mark followed by a series of capital C's. For example:

```
00010 PRINT USING "'CCCCCCC","A"
00020 PRINT USING "'CCCCCCC","AB"
00030 PRINT USING "'CCCCCCC","ABC"
00040 PRINT USING "'CCCCCCC","ABCD"
00050 PRINT USING "'CCCCCCC","ABCDE"

READY

   A
  AB
  ABC
 ABCD
 ABCDE
```

If there are more characters than there are places in the field, BASIC left-justifies the string and does not print the excess characters.

4.3.5 Printing Strings In Extended Fields

The extended field is the only field that automatically prints the entire string. If you specify an extended field, BASIC left-justifies the string as it does for a left-justified field. But, if the string has more characters than there are places in the field, BASIC extends the field and prints the entire string. This extension may cause other items to be misaligned.

An extended field is composed of a single quotation mark followed by a series of capital E's.

Consider the following example that uses extended, left-justified, right-justified, and centered fields.

```
PRUS.B20
Friday, May 20, 1977 17:51:45

00010   PRINT USING "'LLLLLLLLL","THIS TEXT"
00020   PRINT USING "'LLLLLLLLLLLLLL","SHOULD PRINT '
00030   PRINT USING "'LLLLLLLLLLLLLL","AT LEFT MARGIN"
00040   PRINT USING "'RRRR","1,2,3,4"
00050   PRINT USING "'RRRR","1,2,3"
00060   PRINT USING "'RRRR","1,2"
00070   PRINT USING "'RRRR","1"
00080   PRINT USING "'CCCCCCCCC","A"
00090   PRINT USING "'CCCCCCCCC","ABC"
00100   PRINT USING "'CCCCCCCCC","ABCDE"
00110   PRINT USING "'CCCCCCCCC","ABCDEFG"
00120   PRINT USING "'CCCCCCCCC","ABCDEFGHI"
00130   PRINT USING "'LLLLLLLLLLLLLLLLL","YOU ONLY SEE HALF OF THE LINE"
00140   PRINT USING "'E","YOU CAN SEE ALL OF THE LINE WHEN EXTENDED"
00150   END

READY
RUNNH
THIS TEXT
SHOULD PRINT
AT LEFT MARGIN
1,2,3
1,2,3
  1,2
    1
     A
    ABC
   ABCDE
  ABCDEFG
 ABCDEFGHI
YOU ONLY SEE HALF
YOU CAN SEE ALL OF THE LINE WHEN EXTENDED

READY
```

4.4 SUMMARY OF PRINT USING FORMAT

Table 4-1 Format Characters For Numeric Fields

Character	Effect on Format
# number sign	Reserves place for one digit.
. decimal point (period)	Determines location of decimal point.
, comma	Causes a comma to be printed between every third digit starting from the decimal point and proceeding from right to left.

Table 4-1 (Cont.) Format Characters For Numeric Fields

Character	Effect on Format
** two asterisks	Cause leading asterisks to be printed before the first digit instead of spaces. The field formed is called an asterisk fill field. They also reserve places for two digits.
$$ two dollar signs	Cause a dollar sign to be printed before the first digit. The field formed is called a dollar sign field. They reserve places for one dollar sign and one digit.
^^^^ four circumflexes (up-arrows)	Cause number to be printed in E (exponential) format. They also reserve four places for the E notation.
- minus sign	Causes a trailing minus sign to be printed when number is negative. Printing a negative number in an asterisk fill or a dollar sign field requires that the field also have a trailing minus sign.

Note that neither $$ nor ** can be combined with ^^^^

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 **

String fields are composed of a single quotation mark optionally followed by a series of contiguous capital L's, R's, C's, or E's. The effect these characters have on the format is described in Table 4-2.

Table 4-2 Format Characters for String Fields

Character	Effect on Format
' single quotation mark or ! exclamation point (apostrophe)	Starts string field and reserved place or one character.
L upper case L \\ or 2 backslashes	Causes string to be left-justified and reserved place for one character.
R upper case R	Causes string to be right-justified and reserves place for one character.
C upper case C	Causes string to be centered in field and reserves place for one character.
E upper case E	Causes string to be left-justified, expands field, as necessary, to print the entire string and reserves place for one character.

4.5 THE IMAGE STATEMENT

The IMAGE statement has the following format:

:unquoted string
or
IMAGE unquoted string

where:

unquoted string	is the format for printing characters listed in the PRINT USING statement.
:	are interchangeable.
IMAGE	

For example:

```
00010: ##.## ##.##
00020 PRINT USING 10, 12.345,-12.5

READY
RUNNH
 12.35 %-12.5
```

All characters following the colon (or word IMAGE) except the line terminator are considered part of the output image.

The IMAGE statement must be the only statement on a line. No comment fields are allowed. You can, however, continue an image to another line with an ampersand (&) see Section 1.3.1. BASIC continues the line until it finds a line terminator without an ampersand preceding it.

IMAGE is a non-executable statement, therefore, BASIC ignores the IMAGE statement until a PRINT USING statement appears in the program.

You can also use an IMAGE statement with string formats:

```
00010: ++ 'CCCC ++ 'LLLL
00020 INPUT A$
00030 IF A$ = "STOP" GOTO 50
00040 PRINT USING 10, A$,A$
00050 END

READY
RUNNH
 ? ABCDE
 ++ ABCDE ++ ABCDE
```

4.6 PRINT USING STATEMENT ERROR CONDITIONS

There are two types of PRINT USING error conditions, fatal and warning. When a fatal error occurs, BASIC stops executing the program and prints a fatal error message. When a warning is present, BASIC continues to execute the program, although the resulting output may not be in the format intended.

4.6.1 Fatal Error Conditions

A fatal 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 that does not specify a trailing minus.

4.6.2 Warning Conditions

Warning error conditions are:

1. A number does not fit in the field. If a number is larger than the field allows, BASIC prints a percent sign (%) followed by the number in the standard PRINT format.
2. A string does not fit in the field. If a string is larger than any field other than an extended field, BASIC truncates the string and does not print the excess characters.
3. A field contains an illegal combination of characters. If a field contains an illegal combination of characters, the first illegal character and all characters to its right are not recognized as part of the field. They may form another valid field or they may be considered text. If the illegal characters form a new valid field, this field may cause a fatal error condition.

Consider the following examples of illegal combinations of characters in numeric fields.

Illegal Combinations

```
00010 PRINT USING "$$**##.##",5.41,16.30

READY
RUNNH
$5**16.30
```

$$ are combined with **. $$ is a complete field and **##.# forms a second valid field. $5 is printed by $$ and **16.30 is printed by **####.

```
00010 PRINT USING "$$**##.## 'LLL",5.41,"ABC"

READY
RUNNH
$5
? 234  Numeric IMAGE specified for a string at line 00010 of MAIN PROGRA
M
```

The same illegal combination appears here, but the next data item is a string. BASIC produces the fatal error message after trying to print the string "ABC in the numeric field **##.##.

```
00010 PRINT USING "##.#^^^^",5.43E09

READY
RUNNH
% 5.43E+9^^^
```

Field has only three not four. The number does not fit in the field ##.#, a % and the number are printed followed by the ^^^.

```
00010 PRINT USING "'LLEEE","VWXYZ"

READY
RUNNH
VWXEEE
```

Two letters can not be combined in one field.
EEE is printed as it is.

Attempting to print characters as text produces errors when the characters form a valid field. For example:

```
00010 PRINT USING "THERE ARE ### # ## NAILS",123,4,16,6
```

is an attempt to print

```
THERE ARE 123 # 4 NAILS
THERE ARE 16 # 6 NAILS
```

but instead produces

```
THERE ARE 123 4 16 NAILSTHERE ARE    6
```

To correctly print characters that form a valid field, use a string field and place the characters as a string constant in the list. For example:

```
00010 A$="THE BALANCE OF ACCOUNT '#### IS $$###.##"
00020 PRINT USING A$,"#",5634,107.56

READY
RUNNH
THE BALANCE OF ACCOUNT #5634 IS  $107.56
```

This is also the only way to print a single or double quotation mark character with the PRINT USING statement.

CHAPTER 5
CONTROL STATEMENTS

5.1 TRANSFERRING CONTROL OF THE PROGRAM

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

5.1.1 Unconditional Transfer – The GOTO Statement

The GOTO statement causes control to be transferred to the statement that it identifies.

The format of the GOTO statement is:

GOTO line number

where:

line number is the next line to be executed.

This line number can be smaller or larger than the line number of the GOTO statement. Therefore, you have the option to skip any number of lines in either direction.

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

```
30 GOTO 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.

Consider the following sample program with a GOTO statement:

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

READY
RUNNH
 2             4
```

In this program, control passes in the following sequence:

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

Notice that line 30 is never executed.

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

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

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

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

```
00010 REM THIS IS AN EXAMPLE OF A GOTO
00020 INPUT A,B
00030 C = A*B
00040 PRINT "C = A*B THE ANSWER IS ";C
00050 GOTO 10
00060 END

READY
RUNNH
 ? 25,2
C = A*B THE ANSWER IS  50
```

At line 50, BASIC transfers control to line 10. (Refer to Section 5.1.3 for the IF-THEN-ELSE statement.) Because line 10 is a non-executable statement (a remark), BASIC ignores it and transfers control to line 20.

NOTE

Before you use the GOTO statement, be sure you know how to stop your program from running in an infinite loop. Refer to your User's Guide for this information.

5.1.2 Multiple Branching – The ON-GOTO Statement

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

The ON-GOTO statement has the following format:

ON numeric expression $\left\{\begin{matrix}\text{GOTO}\\\text{THEN}\end{matrix}\right\}$ line number(s)

where:

numeric expression	is any legal BASIC numeric expression.
GOTO THEN	are interchangeable keywords.
line number(s)	must be separated by commas.

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

The following examples illustrate the ON-GOTO statement:

```
10 ON A+B GOTO 10,20,30,40
20 ON J% GOTO 50,60,40,100,110
```

Notice that the line numbers in the list can be in any order. The numeric expression is evaluated, and if the value of the expression is:

1. control branches to the first line number specified.
2. control branches to the second line number specified.
3. control branches to the third number specified.

Consider this example:

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

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

5.1.3 Conditional Transfer – The IF-THEN-ELSE Statement

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

The format of the IF-THEN-ELSE statement is:

IF conditional expression {THEN / GOTO} line number

or

IF conditional expression THEN statement(s)

or

IF conditional expression {THEN / GOTO} line number ELSE {line number / statement(s)}

IF conditional expression THEN statement(s) ELSE {line number / statement(s)}

where:

conditional expression	can be any expression. It can also be a variable where the value 0 is false and anything else is true.
statement(s)	can be one or more BASIC statements including IF-THEN-ELSE.

The IF-THEN-ELSE statement gives you several choices: you may specify a conditional transfer of control to another statement line, or you may cause another statement to execute without transferring control, depending on the truth of a conditional expression.

If the value of the conditional expression is true, BASIC transfers control to the specified line number (as in the first format) or executes the statements following the THEN (as in the second format). If the relation is not true, the next executable statement following the IF-THEN-ELSE statement is performed. For example:

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

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

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

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

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

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

The following example illustrates the ELSE clause:

```
10 IF A>=90 THEN G$ = "A"&
   ELSE IF A>=80 AND A<90 THEN G$ = "B"&
   ELSE IF A>=70 AND A<80 THEN G$ = "C"&
   ELSE IF A>=60 AND A<70 THEN G$ = "D"&
   ELSE G$ = "F"
```

You can also use string expressions as in this example:

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

If the value of the string variable C$ is equal to the string "OUTPUT", control passes to line 10. See Section 2.4.2 for string expressions.

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

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

For example:

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

If A is equal to 1, BASIC prints:

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

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

If A is not equal to 1, BASIC prints:

```
RUNNH
 ? 5
NOT = 1
```

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

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

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

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

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

3. If the statement following the THEN or ELSE clause is a FOR statement (FOR modifier is not permitted, Section 6.1.5), you must include the corresponding NEXT statement in the same THEN or ELSE clause. For example:

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

5.2 EXECUTION OF LOOPS

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

5.2.1 The FOR and NEXT Statements

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

The FOR statement has the following format:

FOR variable = num expr 1 TO num expr 2 [[STEP num expr 3]]

where:

variable	is a numeric variable known as the loop index.
num expr 1	is the first numeric expression – the initial value of the index.
num expr 2	is the second numeric expression – the maximum value of the index.

STEP and num expr 3 are 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.

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

```
10 T = 3
20 FOR M = 10*T TO 30*T STEP T
30 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 or equal to 90. When the value of M is greater than 90, control passes to the statement following the NEXT statement.

The NEXT statement has the following format:

NEXT numeric variable(s)

where:

numeric variable(s)	must be the same variable named in the corresponding FOR statement. With multiple variables, the last loop variable must be specified first. Refer to Section 5.2.2 for nested loops.

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

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

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

In this program, the initial value of the 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. The following is the resulting output.

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

Notice that when control passes from the loop, the last value of the loop variable is retained. 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 and the step size is positive, 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 decrements 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 TP 7.7 STEP 1.32
```

NOTE

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

```
10 REM THIS IS A POOR PROGRAM
20 GOTO 40
30 FOR I = 1 TO 20
40 PRINT I
50 NEXT I
60 END
```

Line 20 transfers 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:

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

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:

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

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

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

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

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

5.2.2 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:

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

READY
RUNNH
 I             J

 1             1
 1             2
 1             3
 2             1
 2             2
 2             3
```

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

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

READY
RUNNH
 15
```

5.2.3 The Conditional FOR Statement

Another method of creating loops in a program is by using the Conditional FOR statement. The Conditional FOR statement has the following format:

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

where:

variable	is a numeric variable
num expr1	is the first numeric expression which determines the initial value of the index.
STEP num expr2	increments the value of the index variable. This is optional. The default is +1.
WHILE UNTIL	is the condition tested. The expression can be a relational or logical expression.

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

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

```
LIST

5-12.B20
Tuesday, May 24, 1977 15:55:20

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

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

READY
```

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

The following example illustrates the WHILE clause:

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

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

5.2.4 The FOR Statement With Additional Test

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

FOR variable=num expr 1 TO num expr 2 [[{STEP / BY} num expr 3]] {WHILE / UNTIL} conditional expr

where:

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

This type of loop is equivalent to the normal FOR-NEXT statements except for the addition of the conditional test. Each time BASIC encounters the NEXT statement, the loop variable is incremented. After incrementing the loop variable, BASIC checks the index variable to see if the TO expression has been exceeded. If the TO expression has not been exceeded, BASIC checks the conditional expression. The termination test on the conditional expression is the same as on the Conditional FOR statement, Section 5.2.3.

Consider the following example:

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

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

5.2.5 WHILE And UNTIL Statements

The WHILE and UNTIL statements have the following format:

WHILE conditional expression

UNTIL conditional expression

where:

conditional expression — is any numeric, logical, or relational expression.

Like the FOR statement, the WHILE and UNTIL statements require a corresponding NEXT statement. However, the NEXT statement, in this case, does not contain a variable.

The expression is evaluated before each loop iteration. If the expression is true, BASIC executes the statements within the loop. If the expression is false, BASIC executes the statement following the NEXT statement. For example:

```
00010 WHILE A%<10%
00020 LET A% = A%+I%
00030 FOR I% = 1 TO 5
00040 NEXT I%
00045 NEXT
00050 PRINT A%
00060 END
```

As long as A% is less than 10%, BASIC will execute the statements within the loop.

With the UNTIL statement, the loop executes until the expression is true. For example:

```
00010 I = 12
00020 UNTIL I = 0
00030 PRINT I;
00040 I = I-1
00050 NEXT
```

5.3 TIME LIMITS

BASIC provides statements to suspend program execution for a specified amount of time. These statements are SLEEP and WAIT.

5.3.1 The SLEEP Statement

The SLEEP statement has the following format:

SLEEP numeric expression

where:

numeric expression — is the number of seconds to delay further execution of the program or subprogram (Chapter 11).

For example:

```
10 SLEEP 120*10
```

At the end of 1200 seconds (20 minutes) BASIC continues execution.

To awaken a job from a SLEEP state before the specified number of seconds has elapsed, type a line terminator.

5.3.2 The WAIT Statement

The Wait statement has the following format:

WAIT numeric expression

where:

numeric expression	specifies the maximum number of seconds allowed for all future input from the terminal before an error condition is signalled.

For example:

```
10 WAIT 60
```

BASIC will wait 60 seconds for input before issuing an error message.

The WAIT statement is used in conjunction with the INPUT statement so that you can set time limits for responses to your program.

A WAIT statement with a value of 0 or no value, indicates that no WAIT error condition exists no matter how long it takes for a response. Thus, WAIT 0 turns off a previous WAIT.

You must place the WAIT statement before the respective INPUT statement. For example:

```
00010 WAIT 15
00020 INPUT A,B,C
00030 D = A*B/C
00040 PRINT D
```

BASIC waits 15 seconds for a response to the INPUT statement. If no response is typed, BASIC prints an error message.

5.4 STOPPING PROGRAM EXECUTION – THE STOP AND END STATEMENTS

There are three methods of halting program execution:

1. Executing all the statements
2. Using the STOP statement
3. Using the END statement

The first method is shown in the following example. The program executes completely and then BASIC closes all files:

```
10 FOR I = 1 TO 10
20 PRINT I
30 NEXT I
```

The STOP statement has the following format:

STOP

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

STOP AT LINE n

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.

The STOP statement halts execution but it does not close files. To cause BASIC to close files at program termination, use the END statement.

The END statement has the following format:

END

The END statement is optional unless subprograms are in the same program. See Chapter 11 for information on subprograms. If you include an END, it must have the largest line number in the main program. Any reference to an END statement via a GOTO or IF-THEN-ELSE statement terminates program execution and closes all files.

An END statement does not cause BASIC to print a message on the terminal. If a message is desired, use the STOP statement.

If you do not include a STOP or an END statement in a program, the execution of the last statement of the program terminates program execution and closes all files.

The following examples show all three options of ending a program:

```
00010 READ A,B,C
00020 PRINT "A = ";A
00030 PRINT "B = ";B
00040 PRINT "C = ";C
00050 DATA 100,300,450

READY
RUNNH
A =  100
B =  300
C =  450
```

BASIC executes all statements and closes all files.

```
00010 READ A,B,C
00020 PRINT "A = ";A
00030 PRINT "B = ";B
00040 PRINT "C = ";C
00050 DATA 100,300,450
00060 STOP

READY
RUNNH
A =  100
B =  300
C =  450
STOP at line 00060 of MAIN PROGRAM
```

BASIC executes all statements and stops execution at line 60. Files are not closed.

```
00010 READ A,B,C
00020 PRINT "A = ";A
00030 PRINT "B = ";B
00040 PRINT "C = ";C
00050 DATA 100,300,450
00060 END

READY
RUNNH
A =  100
B =  300
C =  450
```

BASIC executes all statements and closes all files.

As you can see, the first and third examples have the same output. END is only necessary when you plan to reference the end of the program with a transfer statement.

5.5 SUBROUTINES

Subroutines are like functions (Section 2.4.6) 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 statement, which transfers control of the program to a subroutine, and the RETURN statement, which returns control from that subroutine back to normal program execution.

In BASIC, you can enter more than one subroutine in the same program. Subroutines are easier to locate (for debugging purposes) if you place them near the end of the program, before any DATA statements, and before the END statement (if present). 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 subroutine calls (one subroutine within another) up to a system defined limit. See your User's Guide.

The following sections describe the building of subroutines with the GOSUB and RETURN statements.

5.5.1 The GOSUB and RETURN Statements

When BASIC begins executing a program, it continues until it encounters a GOSUB statement. The GOSUB statement has the following format:

GOSUB line number

where:

line number — following the keyword GOSUB can be the first line of the subroutine or an entry point within the subroutine.

BASIC transfers control to that line. For example:

```
10 GOSUB 200
```

BASIC stops executing sequentially at line 10 and transfers control to line 200. BASIC executes the subroutine until it encounters a RETURN statement, which causes BASIC to transfer control back to the statement immediately following the calling GOSUB statement. (A subroutine can exit only through a RETURN statement.)

The RETURN statement has the following format:

RETURN

Before transferring control to a subroutine, BASIC internally records the next sequential statement following the GOSUB statement. The RETURN statement is a signal to BASIC to return to the statement previously recorded. In this way, no matter how many subroutines there are or how many times they are called, BASIC always knows where to transfer control. For example:

```
00010 INPUT A,B,C
00020 GOSUB 40
00030 PRINT D
00035 GOTO 70
00040 REM THIS IS A SUBROUTINE
00050 D = A * B - C
00060 RETURN
00070 END
```

Line 20 sends BASIC to line 40; then line 60 returns execution to line 30. The resulting output is:

```
RUNNH
 ? 5,10,15
 35
```

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

```
00010 DIM B(100)
00020 GOSUB 60
00030 GOSUB 60
00040 GOSUB 60
00050 GOTO 110
00060 LET A = 0
00070 FOR I = 1 TO 5
00080 LET A = A+B(I)
00090 NEXT I
00100 RETURN
00110 END
```

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

5.5.2 The ON-GOSUB Statement

The ON GOSUB statement is used to conditionally transfer control to one of several subroutines or to one of several entry points into one or more subroutines. The ON-GOSUB statement has the following format:

ON numeric expression GOSUB line number(s)

where:

numeric expression	is any legal BASIC numeric expression.
line number(s)	a list of line numbers contained in the program, separated by commas.

The ON-GOSUB statement works like the ON-GOTO statement (Section 5.1.2). 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.
2, control passes to the second line number specified.
3, control passes to the third line number specified.

and so on. If the expression is less than 1 or greater than the number of line numbers in the list, BASIC prints an error message to that effect. The following is an example of an ON GOSUB statement:

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

When A+B=1, go to the subroutine on line 200, (first line number in the list)

A+B=2, go to the subroutine on line 300, (second line number in the list)

A+B=3, go to the subroutine on line 120, (third line number in the list)

(A+B)<1, print error message,

(A+B)>4, print error message.

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

5.6 ERROR CHECKING

Normally, BASIC detects errors while executing a program and either terminates execution or prints a warning message. These errors fall into two general categories:

1. Computational errors
2. Input and output errors

However, if you plan ahead, you can prepare alternatives which can save you time in the event of an error. You can build an error-handling routine that is activated when, and if, BASIC finds an error. This routine takes control away from the normal system errors and gives it to your error-handling routine.

5.6.1 ONERROR GOTO and RESUME Statements

The ONERROR GOTO statement provides the means for trapping errors. This statement has the following format:

ONERROR GOTO line number

where:

line number after the keyword GOTO specifies the beginning of the error-handling routine.

This statement tells BASIC that a user error-handling routine exists beginning at the specified line number. The routine analyzes any input-output or computational error the program encounters and tries to recover from it.

If an error occurs before BASIC executes the ONERROR GOTO statement, BASIC proceeds with normal system error handling.

If an error occurs after the ONERROR statement has been executed, program execution is interrupted, and BASIC transfers control to your error routine.

During the execution of the error-handling routine, the variable

ERR	is set to one of the values listed in the error table (Table 5-1).
ERL	is set to the line number where the error occurred.
ERN$	is set to the name of the program or subprogram being executed.

If an error occurs within the user error-handling routine, the system error handler takes over. Within the error-handling routine, the ONERROR GOTO 0 returns error processing to the system.

The ONERROR GOTO statement is local to the program, subprogram (Chapter 11), or function in which it is contained. However, a special keyword can alter this situation:

```
ONERROR GO [[TO]] BACK
```

With this statement, if an error occurs in a function (Chapter 7) or subprogram (Chapter 11), BASIC can check the caller of the function or subprogram for a user error-handling routine. (ONERROR GO [[TO]] BACK does not extend across a CHAIN boundary, Section 11.2.)

You place the RESUME statement at the end of the error-handling routine. The RESUME statement has the following format:

```
RESUME [[line number]]
```

The RESUME acts like the RETURN statement. It returns to the program line that caused the error. A RESUME statement without a line number resumes execution at the beginning of the line where the error occurred. If the line contains multiple statements, BASIC usually resumes execution at the beginning of the line. However, if you have a DEF, FNEND, DIM, FOR, or NEXT statement in a multi-statement line, execution resumes:[1]

1. Right before the DEF statement
2. Right after the FNEND statement
3. Right after the DIM statement
4. Right after the FOR, WHILE, or UNTIL statement. (The loop will not be reinitialized.)
5. Right after the NEXT statement

If you have two or all five of these statements in a multi-statement line, execution resumes at the last one encountered.

If you specify a line number after the RESUME statement, BASIC resumes execution at the beginning of that line. For example, the following illustrates both methods:

```
200 RESUME
200 RESUME 25
```

If the error-handling routine is in a multi-line function or subprogram, then the line number in the RESUME statement must refer to a line within the function or subprogram.

5.6.2 Error Table

The following table lists the values of the variable ERR and the corresponding error messages.

[1] On the DECSYSTEM-20, execution resumes at the beginning of the line.

Table 5-1 Error Table

ERR	Message Printed	Meaning
1	BAD DIRECTORY FOR DEVICE	The directory of the device referenced is in an unreadable format.
2	ILLEGAL FILE NAME	The filename specified is not acceptable. It contains embedded blanks or unacceptable characters.
3	ACCOUNT OR DEVICE IN USE	The specified operation cannot be performed because the file is already open by someone else. This message has a general "file in use" meaning.
4	NO ROOM FOR USER ON DEVICE	Storage space allowed for the current user on the device specified has been used or the device as a whole is too full to accept further data.
5	CAN'T FIND FILE OR ACCOUNT	The file specified or current user account numbers were not found on the device specified. This message has a general "not here" meaning.
6	NOT A VALID DEVICE	Attempt to use an illegal or non-existent device.
7	I/O CHANNEL ALREADY OPEN	An attempt was made to open one of the I/O channels which had already been opened by the program.
8	DEVICE NOT AVAILABLE	The device requested is currently reserved by another user.
9	I/O CHANNEL NOT OPEN	Attempt to perform I/O on one of the channels that has not been previously opened in the program.
10	PROTECTION VIOLATION	The current user is not allowed to perform the requested operation on the specified file. Input may have been requested from an output-only device or vice versa. This message has a general "can't do that" meaning.
11	END OF FILE ON DEVICE	Attempt to perform input beyond the end of a date file.
12	FATAL SYSTEM I/O FAILURE	An I/O error has occurred on the system level. The user has no guarantee that the last operation has been performed.
13	USER DATA ERROR ON DEVICE	One or more characters may have been transmitted incorrectly due to a parity error, bad punch combination on a card, or similar error.
14	DEVICE HUNG OR WRITE LOCKED	Check hardware condition of device requested. Possible causes of this error include a line printer out of paper or high-speed reader being off-line.

Table 5-1 (Cont.) Error Table

ERR	Message Printed	Meaning
15	KEYBOARD WAIT EXHAUSTED	Time requested by WAIT statement has been exhausted with no input received from the specified keyboard.
16	NAME OR ACCOUNT NOW EXISTS	An attempt was made to rename a file with the name of a file which already exists, or an attempt was made by the system manager to insert an account code that is already within the system.
17	TOO MANY OPEN FILES ON UNIT	Only one DECtape output file is permitted per DECtape drive. Only one open file per magtape drive is permitted.
28	PROGRAMMABLE ^C TRAP	ON ERROR-GOTO subroutine was entered through a program trapped by means of control C.
30	DEVICE NOT FILE-STRUCTURED	An attempt is made to access a device, other than a disk as a file-structured device. This error occurs, for example, when the user attempts to gain a directory listing of a non-directory device.
31	ILLEGAL BYTE COUNT FOR I/O	The buffer size specified in the RECORDSIZE option of the OPEN statement does not match the I/O attempted.
43	VIRTUAL ARRAY NOT ON DISK	A non-disk device is open on the channel upon which the virtual array is referenced.
44	MATRIX OR ARRAY TOO BIG	In-core array size is too large.
45	VIRTUAL ARRAY NOT YET OPEN	An attempt was made to use a virtual array before opening the corresponding disk file.
46	ILLEGAL I/O CHANNEL	Attempt was made to open a file on an I/O channel outside the range of legal channel.
47	LINE TOO LONG	Attempt to input a line longer than the buffer.
48	FLOATING POINT ERROR	Floating point overflow or underflow. If no transfer is made to an error handling routine, a 0 is returned as the floating-point value for underflow and the maximum positive number for overflow.
49	ARGUMENT TOO LARGE IN EXP	Value is outside of legal range.
50	DATA FORMAT ERROR	
51	INTEGER ERROR	Attempt to use a number as an integer when that number is outside the allowable integer range. If no transfer is made to an error handling routine, a 0 is returned as the integer value.

Table 5-1 (Cont.) Error Table

ERR	Message Printed	Meaning
52	ILLEGAL NUMBER	Improperly formed input. For example, "1..2" is an improperly formed number.
53	ILLEGAL ARGUMENT IN LOG	Negative or zero argument to log function. Value returned is the argument as passed to the function.
54	IMAGINARY SQUARE ROOTS	Attempt to take square root of a number less than 0. If no transfer is made to an error handling routine, the value returned is the square root of the absolute value of the argument.
55	SUBSCRIPT OUT OF RANGE	Attempt to reference an array element beyond the number of elements created for the array when it was dimensioned.
56	CAN'T INVERT MATRIX	Attempt to invert a singular matrix.
57	OUT OF DATA	A READ requested additional data from an exhausted DATA list.
58	ON STATEMENT OUT OF RANGE	The index value in an ON GOTO or ON GOSUB statement is less than 1 or greater than the number of line numbers in the list.
59	NOT ENOUGH DATA IN RECORD	An INPUT statement did not find enough data in one line to satisfy all the specified variables.
60	INTEGER OVERFLOW, FOR LOOP	The integer index in a FOR loop attempted to go beyond implementation defined limits.
61	DIVISION BY 0	Attempt by the user program to divide some quantity by 0. If no transfer is made to an error handling routine, the largest positive number is returned as the result.

CHAPTER 6
STATEMENT MODIFIERS

6.1 MODIFYING STATEMENTS

Another useful tool for building programs is the statement modifier. In BASIC, the statement modifier qualifies or restricts the execution of a statement; thus allowing you to:

1. Indicate conditional execution of a statement
2. Create an implied loop

An implied loop built with a statement modifier iterates only one statement on a line. In cases where the FOR-NEXT statement loop is extremely simple, the necessity for both the FOR and NEXT statements is eliminated.

BASIC provides five statement modifiers:

1. IF
2. WHILE
3. UNTIL
4. UNLESS
5. FOR

These statement modifiers cannot stand alone; they must be appended to a statement. Most BASIC statements can be modified. There are some, however, that cannot be modified, and some that do not need to be modified. Table 6-1 lists the various statements.

When using statement modifiers with the various forms of the IF statement, the following rules apply:

1. Append statement modifiers to either the THEN clause or the ELSE clause of an IF statement.
2. The statement modifier applies only to the clause it is appended to and not to the statement as a whole.

If you have more than one statement on a line, the modifier applies only to the statement immediately preceding it. You can also append more than one statement modifier to a single statement. In this case, BASIC processes the modifiers from right to left. See Section 6.1.5. Statement modifiers are reserved words. See Appendix C.

6.1.1 The IF Modifier

The IF modifier has the following format:

statement IF condition

where the condition can be any numeric expression.

BASIC tests to see if the condition is true or false. The statement executes only if the condition is true.

For example:

```
10 PRINT X IF X <> 0
```

Table 6-1 Statements

Can Have Modifiers	Cannot Have Modifiers
CALL	DATA
CHAIN	DEF
CHANGE	DIM
CLOSE	END
GOSUB	FNEND
GOTO	FOR
IF-THEN-ELSE	NEXT
INPUT	REM
KILL	SUB
LET	SUBEND
MAT INPUT	UNTIL
Matrix Initialization	WHILE
MAT PRINT	
ON GOSUB	
ON GOTO	
ONERROR	
OPEN	
PRINT	
RANDOMIZE	
RESTORE	
RESUME	
RETURN	
SLEEP	
STOP	

BASIC prints the value X only if X is not equal to 0. The example is the same as using the IF-THEN-ELSE statement:

```
10 IF X <> 0 THEN PRINT X
```

Note that you cannot add an ELSE or a THEN clause to the IF modifier. However, you can use the IF modifier in a THEN or ELSE clause:

```
10 IF A=B THEN PRINT B IF B<100
```

The IF modifier in this example applies only to the statement PRINT B. BASIC prints the value of B when both the following conditions are true.

1. If A is equal to B
2. If B is less then 100

6.1.2 The UNLESS Modifier

The UNLESS modifier has the following format:

statement UNLESS condition

where the statement executes only if the condition is false. For example:

```
10 PRINT A UNLESS A=0
```

BASIC prints the value of A only if A is not equal to 0.

The following examples produce the same results as the UNLESS modifier:

```
10 PRINT A IF NOT A=0
20 IF NOT A=0 THEN PRINT A
30 IF A <> 0 THEN PRINT A
```

The UNLESS modifier simplifies the negation of a logical condition.

6.1.3 The WHILE Modifier

The WHILE modifier has the following format:

statement WHILE condition

where the statement executes repeatedly as long as the condition is true.

For example:

```
00010 Y=2
00020 Y=Y^(2) WHILE Y<1E6
00030 PRINT Y
```

Line 20 executes over and over as long as X^(2) is less than 1E6. When X^(2) is greater than or equal to 1E6, BASIC executes line 30.

The WHILE modifier sets up a loop wherein one statement executes iteratively if the condition is true. There is no formal control variable, i.e., I=1 TO 10, as in a FOR-NEXT loop. Instead, the structure of the loop modifies the values which determine loop termination.

The previous example is equivalent to:

```
00010 Y=2
00020 Y=Y^(2)
00030 IF Y^(2)<1E6 GOTO 40 ELSE GOTO 50
00040 PRINT Y
00050 END
```

Be careful not to create an infinite loop with the WHILE modifier. The following sequence never terminates properly:

```
10 X=X+1 WHILE I<1000
```

I is set to 0 at the beginning of program execution; therefore I is less than 1000. The condition of the WHILE modifier is unrelated to the assignment X=X+1.

Because 0 is always less than 1000, the statement causes an infinite loop.

Consider the following example:

```
00010 READ Z WHILE Z < 10
00020 IF Z >= 10 THEN PRINT "?WHILE ON READ FAILED."
00030 DATA 1,2,3,4,5,6,7,8,9,10
00040 END
```

Line 10 reads the data in line 30 until it reaches 10. Then the WHILE condition is no longer true, and line 20 executes.

```
RUNNH
?WHILE ON READ FAILED.
```

6.1.4 The UNTIL Modifier

The UNTIL modifier has the following format:

statement UNTIL condition

where the statement executes repeatedly as long as the condition is false. For example:

```
00005 X = 40
00010 X = X  + 1 UNTIL X > 795
00020 PRINT X
```

Line 10 executes repeatedly as long as X is less than or equal to 795. The statement continues until the condition becomes true.

The UNTIL modifier is similar to the WHILE modifier in that it does not need a formal control variable to determine loop termination.

The previous example is equivalent to:

```
00005 X = 750
00010 X = X + 5
00015 PRINT X
00020 IF X <=795 GOTO 10
```

Be careful not to create an infinite loop with the UNTIL modifier.

Consider the following example:

```
00010 A=1\B=2\C=3
00020 LET D=C+2*A UNTIL D>=50
00030 IF D>=50 THEN PRINT D
```

Line 20 continues to execute as long as D is less than 50. Once D is greater than or equal to 50, BASIC proceeds to line 30.

6.1.5 The FOR Modifier

The FOR modifier has the following format:

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

or

statement FOR variable = num expr1 TO num expr2 [[STEP num expr3]]

The FOR modifier is used to create an implied loop on a single line.

For example:

```
10 PRINT I, SQR(I) FOR I = 1 TO 10
```

is equal to

```
10 FOR I = 1 TO 10
20 PRINT I, SQR(I)
30 NEXT I
```

By using the FOR modifier for simple loops, you eliminate the need for the FOR-NEXT statement. Notice that the FOR modifier applies only to one statement on the line. Hence, it iterates only one statement. You can have many FOR modifiers in a single program.

The STEP and BY clauses increment the index variable just as they do in the FOR statement. The default is +1.

```
10 PRINT A=B*C FOR I = 1 TO 50 STEP 3
```

If you use the WHILE or UNTIL option, the loop continues as long as the WHILE condition is true; the loop continues as long as the UNTIL condition is false.

The following is an example of a FOR modifier and an IF modifier:

```
10 DIM X(100)
20 PRINT I, X(I) IF X(I) <> 0 FOR I = 1 TO 100
```

With more than one modifier, BASIC reads from right to left. Therefore, the implied loop, I=1 TO 100, executes first, then the IF modifier is tested. Appending more than one modifier to a statement is known as nesting modifiers.

Consider the following examples:

```
10 LET A=A+J FOR J=1 TO 10 IF A+J<10
20 LET B=B-J FOR J=1 TO 10 UNLESS B>=10
30 LET C=C+J*2 FOR J=1 TO 4 WHILE C<10
40 LET D=D-J FOR J=2 TO 10 STEP 2 UNTIL D=-10
50 LET F=I+J FOR I=1 TO 5 FOR J=2 TO 6
60 END
```

In each case, the modifiers are tested from right to left. If the first modifier fails, BASIC continues execution at the next statement of the program (not the next modifier on the same line).

CHAPTER 7
FUNCTIONS

7.1 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 2.4.6). You can use functions which return numeric values in numeric expressions and functions which return string values in string expressions. BASIC provides numeric functions, string functions, conversion functions, date and time functions, and user-defined functions. BASIC-PLUS-2 library functions are reserved words. See Appendix C.

7.2 NUMERIC FUNCTIONS

The BASIC-PLUS-2 numeric functions perform standard mathematical operations.

BASIC provides the following trigonometric functions:

1. SIN – sine
2. COS – cosine
3. TAN – tangent
4. ATN – arctangent

In addition, BASIC has a special function, PI, which returns the value of a transcendental number frequently used as a trigonometric constant.

BASIC also has algebraic functions:

1. SQR – the square root of a number
2. EXP – the value of e, an algebraic constant, raised to any power
3. LOG and LOG10 – the logarithm of a number
4. INT – the integral part of a number
5. ABS – the absolute value of a number
6. FIX – the truncated value of a number

All BASIC numeric functions return real numbers (internally) as opposed to integer values. Note that a numeric argument to a function is converted to integer by truncation.

7.2.1 Trigonometric Functions (SIN, COS, TAN, ATN, and PI)

BASIC provides functions, SIN and COS, to find the sine and cosine of an angle in radians. In addition, you can use the ATN function to find the arctangent of a number, the angle whose tangent is equal to the number. The format of these functions is:

```
SIN(expression)
COS(expression)
TAN(expression)
ATN(expression)
```

The PI function returns a numeric constant, 3.141593. The accuracy of this number depends on your system. Because PI is a transcendental number, the value the PI function returns is only an approximation. The format of the PI function is:

PI

PI can be used in any expression.

Do not include an argument with PI; if you do, BASIC prints an error message.

Consider the following example:

```
00010 REM CONVERT ANGLE (X) TO RADIANS, AND
00020 REM FIND SIN AND COS
00025 PRINT "DEGREES","RADIANS","SINE","COSINE"
00030 INPUT X\GO TO 100 IF X<0
00040 LET Y=X*PI/180
00050 PRINT X,Y,SIN(Y),COS(Y)
00060 GOTO 30
00100 END

READY
RUNNH
DEGREES        RADIANS        SINE           COSINE
 ? 0
 0              0              0              1
 ? 10
 10             0.1745329      0.1736482      0.9848078
 ? 20
 20             0.3490658      0.3420201      0.9396926
 ? 30
 30             0.5235988      0.5            0.8660254
 ? 360
 360            6.283185       0              1
 ? 45
 45             0.7853982      0.7071068      0.7071068
 ? -1
```

Note that in this example, PI is used to convert degrees to radian measure (line 40).

The TAN function returns the tangent of the argument you supply. The TAN function has the following format:

TAN(expression)

where:

expression must be given in radians.

The ATN function returns the value in radians of the angle whose tangent is equal to the argument. The format of the ATN function is:

ATN(expression)

The value BASIC returns is also in radians.

The ATN function returns a value in the range +PI/2 to -PI/2.

The following example tests the ATN function. The program inputs an angle in degrees, converts it to radians, and calculates the tangent of the angle according to this formula:

TAN(X) =SIN(X)/COS(X)

Then the program converts the tangent to an angle using the ATN function and prints the results. The angles returned by the ATN function should be the same as the angles you supply.

```
00100 PRINT "SUPPLY AN ANGLE IN DEGREES"
00110 PRINT "ANGLE","ANGLE","TAN(X)","ATAN(X)","ATAN(X)"
00120 PRINT "(DEGS)","(RADS)",,"(RADS)","(DEGS)"
00130 INPUT X\GO TO 200 IF X<0
00140 Y=X*PI/180
00150 Z=SIN(Y)/COS(Y)
00160 PRINT X,Y,Z,ATN(Z),ATN(Z)*180/PI !COMPUTE ARCTANGENT
00170 PRINT
00180 GOTO 130
00190 GOTO 130
00200 END

READY
RUNNH
SUPPLY AN ANGLE IN DEGREES
ANGLE          ANGLE          TAN(X)         ATAN(X)        ATAN(X)
(DEGS)         (RADS)                        (RADS)         (DEGS)
 ? 0
 0              0              0              0              0

 ? 45
 45             0.7853982      1              0.7853982      45

 ? 10
 10             0.1745329      0.176327       0.1745329      10

 ? -1
```

7.2.2 Algebraic Functions

BASIC has several algebraic functions that you can use in calculations:

SQR	Square root function
EXP	Exponential function
LOG	Logarithm function
LOG10	Common Logarithm function
INT	Integer function
ABS	Absolute Value function
SGN	Sign function
FIX	Fix function

7.2.2.1 **Square Root Function (SQR)** – The SQR function returns the square root of the expression you specify. The format of the SQR function is:

SQR(expression)

If the value of the expression is negative, BASIC prints a warning message and the function returns the square root of the absolute value of the expression.

```
00010 INPUT X\GOTO 100 IF X<0
00020 LET Z=SQR(X)
00030 PRINT Z
00040 GOTO 10
00100 END

READY
RUNNH
 ? 16
 4
 ? 1000
 31.62278
 ? 12345
 111.1081
 ? 25E2
 50
 ? 1970
 44.38468
 ? -1
```

7.2.2.2 **Exponential and Log Functions (EXP, LOG, and LOG10)** – The exponential function, EXP, returns e, an algebraic constant, raised to the power specified by the expression, where e is the base of the natural logarithm system. The value of e is approximately 2.71828. The accuracy of this number is system-dependent.

The format of the exponential function is:

EXP(expression)

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

The format of the LOG function is:

LOG(expression)

EXP and LOG are related functions. Specifically, EXP is the inverse of LOG. The following formula describes their relationship:

LOG(EXP(X)) = X

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

EXP Function

```
00010 INPUT X\GOTO 100 IF X<0
00020 PRINT EXP(X)
00030 GOTO 10
00100 END

READY
RUNNH
 ? 4
 54.59815
 ? 10
 22026.47
 ? 9.42100
 12344.92
 ? 4.60517
 99.99998
 ? 25
 7.20049E+10
 ? -1
```

LOG Function

```
00010 INPUT X\GO TO 100 IF X<0
00020 PRINT LOG(X)
00030 GOTO 10
00100 END

READY
RUNNH
 ? 54.59815
 4
 ? 2206.47
 7.699149
 ? 12344.92
 9.421
 ? 99.99998
 4.60517
 ? 7.20049E+10
 25
 ? -1
```

The LOG10 function returns the common logarithm (base 10) of the specified value. The form of the LOG10 function is:

LOG10(expression)

Programs that require the computation of logarithm (base 10) do not have to use the conversion formula described above. For example:

```
00010 INPUT X
00020 PRINT "X","LOG10(X)"
00030 FOR I = 1 TO 5
00040 PRINT X^I,LOG10(X^I)
00050 NEXT I

READY
RUNNH
 ? 5.732
X               LOG10(X)
 5.732           0.7583062
 32.85582        1.516612
 188.3296        2.274919
 1079.505        3.033225
 6187.724        3.791531
```

If the expression supplied for the LOG or LOG10 function is equal to or less than zero, BASIC prints a message, and the function returns a value of zero.

7.2.2.3 **The Integer Function (INT)** – The integer function returns the value of the greatest integer that is less than or equal to the expression you specify. The format of the integer function is:

INT(expression)

For example:

```
00010 PRINT INT(34.47)
00020 PRINT INT(33000.9)

READY
RUNNH
 34
 33000
```

The INT function always returns the value of the greatest integer that is less than the specified integer; however, when you specify a negative number, INT produces a number whose absolute value is larger. For example:

```
00010 PRINT INT(-23.45)
00020 PRINT INT(-14.7)
00030 PRINT INT(-11)

READY
RUNNH
-24
-15
-11
```

Note that the value returned by INT is a real number.

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

```
00010 PRINT INT(34.67+.5)
00020 PRINT INT(-5.1+.5)

READY
RUNNH
 35
-5
```

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:

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

where P represents the number of places of accuracy and is positive for accuracy to the right of the decimal point and negative for accuracy to an integral power of 10.

Consider the following example, which rounds numbers to the number of decimal places specified (line 150):

```
00050 REM PROGRAM TO ROUND OFF DECIMAL NUMBERS
00100 PRINT "WHAT NUMBER DO YOU WISH TO ROUND OFF";
00110 INPUT N
00115 IF N = -9999 THEN 1000
00120 PRINT "TO HOW MANY PLACES";
00130 INPUT P
00140 PRINT
00150 LET A=INT(N*10^P+.5)/(10^P)
00160 PRINT N;"=";A;"TO";P;"DECIMAL PLACES."
00170 PRINT
00180 GO TO 100
01000 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
```

7.2.2.4 **The Absolute Value Function (ABS)** – 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 is always positive. If the expression is a positive number, the absolute value is equal to that number. If the expression is a negative number, the absolute value is equal to -1 times the number. For example:

```
00010 INPUT X\GO TO 100 IF X=0
00020 X=ABS(X)
00030 PRINT X
00040 GOTO 10
00100 END

READY
```

```
RUNNH
 ? -35.7
 35.7
 ? 2
 2
 ? 25E20
 2.5E+21
 ? 10555567
 1.055557E+7
 ? 10.12345
 10.12345
 ? -44.5556669
 44.55567
 ? 0
```

Note that the ABS function returns a real number even if the argument is an integer.

7.2.2.5 **The SIGN(SGN) And FIX (Fix) Functions** – The sign function determines whether an expression is positive, negative or equal to 0. The format of the SGN function is:

SGN(expression)

If the expression is positive, SGN returns a value of +1. If the expression is negative, SGN returns a value of -1. If the expression is equal to zero, SGN returns a value of zero. For example:

```
00010 A=-7.32
00020 B=.44
00030 C=0
00040 PRINT "A=";A,"B=";B,"C=";C
00050 PRINT "SGN(A)=";SGN(A),
00060 PRINT "SGN(B)=";SGN(B),
00070 PRINT "SGN(C)=";SGN(C)
00080 END

READY
RUNNH
A=-7.32         B= 0.44         C= 0
SGN(A)=-1       SGN(B)= 1       SGN(C)= 0
```

Note that the SGN function returns the values as a real number.

The FIX function has the following format:

FIX(expression)

The FIX function returns the truncated value of the argument you supply as a real number not an integer. For example:

FIX(-.5)=0
FIX(2.6)=2

The FIX function is equivalent to:

SGN(X)*INT(ABS(X))

7.2.3 Random Numbers (RND And RANDOMIZE)

The RND function supplies a series of random numbers to a BASIC program. This function is useful if you want to simulate a situation that involves input of an unknown quantity, i.e., a roll of the dice. When you include the RND function in a program, it produces a predictable sequence of numbers that are seemingly unrelated. Because a computer always produces the same results given the same starting conditions, the RND function does not create a truly random series of numbers. Every time you execute the same program you will receive the same series of random numbers. Therefore, the RND function is known as a pseudorandom number generator.

The RND function has the following format:

RND

The RND function returns a random number between 0 and 1 but never returns the extremes of the range, 0 and 1. (This kind of range is called an open range, or open interval.) For example:

```
00010 PRINT RND,RND,RND,RND
00020 END

READY
RUNNH
 0.1948187       0.7324636       0.6087399       0.3225784
```

The program requests 4 random numbers so BASIC prints 4 numbers in the open range 0 to 1.

The RND function has the same starting location each time you run the same program. However, you can change the starting point by adding the RANDOMIZE statement before the RND function in the program. Each time BASIC executes the RANDOMIZE statement, it starts the RND function at a new unpredictable location in the series. This location is determined by the current time of day according to the computer's clock.

NOTE

You should not include the RANDOMIZE statement until you have debugged your program. If you do, you will not know if changes in the results are caused by changes in the program or changes in the starting location of the random number generator.

The RANDOMIZE statement has the following format:

RANDOM[[IZE]]

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

RND without RANDOMIZE

```
00010 PRINT RND,RND,RND,RND
00020 END

READY
RUNNH
 0.1948187       0.7324636       0.6087399       0.3225784

READY
```

```
RUNNH
 0.1948187      0.7324636      0.6087399      0.3225784

READY
RUNNH
 0.1948187      0.7324636      0.6087399      0.3225784
```

Notice every time the program without RANDOMIZE is run, RND produces the same series of values.

RND with RANDOMIZE

```
00005 RANDOMIZE
00010 PRINT RND,RND,RND,RND
00020 END

READY
RUNNH
 0.9734626      0.08921584     0.9579798      0.1565555

READY
RUNNH
 0.6524263      0.8212112      0.1453525      0.6475236

READY
RUNNH
 0.2330611      0.4360573      0.5879883      0.3611355
```

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

```
00010   FOR I = 1 TO 10
00020   PRINT (6-4) * RND+4,
00030   NEXT I
00040   END

READY
RUNNH
 4.389637       5.464927       5.21748        4.645157       4.216904
 4.376965       4.123446       5.426511       5.275825       4.097507
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.

7.2.4 The MOD Function

The MOD function has the following format:

MOD%(A,B)
or
MOD(A,B)

where:

A,B represent numeric constants you supply.

MOD%(A,B) returns the integer result of A mod B, which is the remainder of A/B.

MOD(A,B) returns the real result of A mod B, which is equal to A-B*INT(A/B).

7.3 STRING FUNCTIONS

BASIC provides string functions that allow you to modify strings. With these functions you can:

1. Determine the length of a string (LEN)
2. Trim off trailing blanks from a string (TRM$)
3. Search for the position of a set of characters within a string (POS, INSTR)
4. Extract a segment from a string (SEG$,MID,LEFT$,RIGHT$)
5. Create a string of a certain length (STRING$)
6. Insert spaces into a string (SPACE$)
7. Alter the contents of a string (EDIT$)

Another group of BASIC string functions allows you to convert strings to numbers and numbers to strings. In particular, you can convert:

1. Character to ASCII code (ASCII)
2. ASCII code to character (CHR$)
3. String representation of a number to a number (VAL)

BASIC's relational operators allow you to concatenate and compare strings (Section 2.4.4), but with string functions you can also analyze the composition of a string. The following sections describe these functions.

The functions LEFT$, RIGHT$, MID, and SEG$ all return the null string if their string argument is null. No further range checking is done in this case.

7.3.1 Finding the Length of a String (LEN)

The LEN function returns an integer equal to the number of characters in the specified string (including trailing blanks). The format of the LEN function is:

LEN(string)

For example:

```
00010 A$="ABCDEFGHIJKLMNOPQRSTUVWXYZ"
00020 PRINT LEN(A$)
00030 END

READY
RUNNH
 26
```

7.3.2 Trimming Trailing Blanks (TRM$)

The TRM$ function returns the specified string with all trailing blanks 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:

```
00010 A$="ABCD      "
00020 B$="EFG"
00030 PRINT "BEFORE TRIMMING:",A$+B$
00040 PRINT "AFTER TRIMMING:",TRM$(A$)+B$
00050 END

READY
RUNNH
BEFORE TRIMMING:              ABCD      EFG
AFTER TRIMMING:               ABCDEFG
```

7.3.3 Finding the Position of a Substring (POS, INSTR)

Use the POS or INSTR function to find the position of a group of characters, a substring, in a string. The form of the functions are:

POS(string1, string2, expression)
or
INSTR(expression,string1,string2)

where:

string1	is the string being searched.
string2	is the substring.
expression	is the character position at which BASIC starts the search.

These functions search for and return the position of the first occurrence of string2 in string1, starting with the character position specified by expression. If the specified substring is found, the character position of the first character of the substring is returned. If the specified substring is not found, the function returns 0.

You can use these functions to map a string of characters to a corresponding integer which can then be used in calculations. This technique is called a table look-up: the table string is string1 and the string to be mapped is string2 in the POS function. Consider the following example which translates month names to numbers.

```
00010 REM PROGRAM TO TRANSLATE MONTH NAMES TO NUMBERS
00020 T$ = "JANFEBMARAPRMAYJUNJULAUGSEPOCTNOVDEC" !TABLE STRING
00030 PRINT "TYPE THE FIRST 3 LETTERS OF A MONTH." !INPUT THE STRING
00040 LINPUT M$
00050 IF M$ = "" THEN 99999 !IF THE STRING IS NULL THEN END
00060 IF LEN(M$) <> 3 GOTO 120 !CHECK IF THE STRING IS 3 CHARACTERS
00070 M = (POS(T$,M$,1)+2)/3
00080 REM CHECK IF MONTH IS SPELLED CORRECTLY
00090 IF M <> INT(M) GOTO 120
00100 PRINT M$;" IS MONTH NUMBER";M !PRINT NUMBER IF IT IS CORRECT
00110 GOTO 30
00120 PRINT "INVALID ENTRY - TRY AGAIN."\GOTO 30
00670
99999 END

READY
```

```
RUNNH
TYPE THE FIRST 3 LETTERS OF A MONTH.
 ? NOV
NOV IS MONTH NUMBER 11
TYPE THE FIRST 3 LETTERS OF A MONTH.
 ? DEC
DEC IS MONTH NUMBER 12
TYPE THE FIRST 3 LETTERS OF A MONTH.
 ? JAN
JAN IS MONTH NUMBER 1
TYPE THE FIRST 3 LETTERS OF A MONTH.
 ? AUD
INVALID ENTRY - TRY AGAIN.
TYPE THE FIRST 3 LETTERS OF A MONTH.
 ? AUG
AUG IS MONTH NUMBER 8
TYPE THE FIRST 3 LETTERS OF A MONTH.
 ?
```

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

1. If string1, the table string, is null, an error is given.
2. If string1 is non-null and string2 (the substring) is null, 1 is returned.
3. If neither 1. nor 2. holds, and if the value of the expression is greater than the length of string1 or less than 1, an error is given.

7.3.4 Extracting a Segment from a String (SEG$)

The SEG$ function is used to extract a segment (substring) from a string. The original string remains 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:

```
00010 PRINT SEG$("ABCDEF",3,5)
00020 END

READY
RUNNH
CDE
```

If expression1 equals expression2, SEG$ returns the character at expression1.

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

1. If expression1 < = 0, an error is given.
2. If expression2 > = the length of the string, an error is given.
3. If expression1 > expression2, an error is given.

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

```
00010 A$ = "ABCDEFG"
00020 C$ = SEG$(A$,1,2) + "XYZ"+ SEG$(A$,6,7)
00030 PRINT C$
00040 END

READY
RUNNH
ABXYZFG
```

Line 20 replaces the characters CDE in the string A$ with XYZ.

Examine line 20:

```
20 C$ = SEG$(A$,1,2)+"XYZ"+SEG$(A$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 6th through 9th characters of the string "ABCDEFGHIJK" with "123456", enter the following program:

```
00010 A$ = "ABCDEFGHIJK"
00020 B$ = "123456"
00030 C$ = SEG$(A$,1,5)+ B$ + SEG$(A$,10,LEN(A$))
00040 PRINT C$
00050 END

READY
RUNNH
ABCDE123456JK
```

7.3.5 The MID Function

The MID function has the following format:

MID(string,expression1%,expression2%)

where:

string	is a string constant or string variable.
expression1%	is a positive integer designating the starting position of the substring.
expression2%	is a positive integer designating the number of characters in the substring.

Starting with the character at expression1%, the MID function returns a substring with a length of expression2%.

For example:

```
00010 ALPHA$="ABCDEFGHIJKLMNOPQRSTUVWXYZ"
00020 PRINT MID(ALPHA$,15%,5%)
00030 PRINT
00040 PRINT MID("ENCYCLOPEDIA",3%,6%)
00050 END

READY
RUN

7-17.B20
Monday, May 23, 1977 16:04:41

OPQRS

CYCLOP
```

The following error conditions apply to the MID function:

1. If expression2 is zero, BASIC returns the null string.
2. If expression2 is less than zero, an error message is given.
3. If expression 2 is greater than zero, then

   MID(string,expression1,expression2) is equivalent to
   SEG$(string,expression1,expression1+expression2-1).

7.3.6 The LEFT$ and RIGHT$ Functions

The LEFT$ function has the following format:

LEFT[[$]] (string, expression)

where:

[[$]]	the dollar sign is optional but preferred.
string	represents the string that contains the substring.
expression	represents an integer constant denoting the character position where the copying should stop.

BASIC returns a substring of the string you specify, from the first character in the string to the character position you specify in the expression. For example:

```
00010 PRINT LEFT$("ABCDEFG",4%)
00020 END

READY
RUNNH
ABCD
```

The RIGHT$ function has the following format:

RIGHT[[$]] (string, expression)

where:

[[$]]	the dollar sign is optional but preferred.
string	represents the string that contains the substring.
expression	represents the character position where the copying begins.

BASIC returns a substring of the string you specify, starting with the character position in the expression up to the last character in the string. For example:

```
00010 PRINT RIGHT$("ABCDEFG",6%)

READY
RUNNH
FG
```

In general, if expression is less than 1 or greater than the length of the string, an error is given. However, these two particular cases each return the null string:

LEFT$(string,0) and
RIGHT$(string, 1+LEN(string))

7.3.7 The STRING$ and SPACE$ Functions

The STRING$ function has the following format:

STRING$(expression1%,expression2%)

where:

expression1%	is a positive integer constant representing the length of the string you want to create.
expression2%	is a positive integer constant representing the decimal ASCII value of the character you want in the string.

BASIC creates a string of length expression1% with characters whose ASCII value is expression2%. For example, to create a string consisting of 10 upper case A's, use the following:

```
00010 PRINT STRING$(10%,65%)

READY
RUNNH
AAAAAAAAAA
```

The SPACE$ function has the following format:

SPACE$(expression%)

where:

expression% is an integer constant representing the number of spaces you want to add to a string.

For example:

```
00010 A$="ABC"+SPACE$(5%)
00020 PRINT A$+"DEF"

READY
RUNNH
ABC     DEF
```

7.3.8 The EDIT$ Function

The EDIT$ function has the following format:

string var = EDIT$(string,expression%)

where:

string var	contains the new string after alterations.
string	is a string constant or string variable representing the original string.
expression%	is one of the integers in the following table, or a sum of the integers.

Table 7-1 EDIT$ Conversions

Expression%	Effect
2%	Discard all spaces and tabs.
4%	Discard excess characters: CR, LF, FF, ESC, RUBOUT, and NULL.
8%	Discard leading spaces and tabs.
16%	Reduce spaces and tabs to one space.
32%	Convert lower-case to upper-case.
64%	Convert [to (and] to).
128%	Discard trailing spaces and tabs.

The EDIT$ converts the source character string according to the decimal value of the integer represented by expression%.

For example:

```
00010 B$="DISCARD ALL SPACES AND TABS."
00020 A$=EDIT$(B$,2%)
00030 PRINT B$
00040 PRINT A$
00045 PRINT
00050 C$="REDUCE       SPACES     AND    TABS       TO  ONE    SPACE."
00060 D$=EDIT$(C$,16%)
00070 PRINT C$
00080 PRINT D$
00090 END

READY
RUN

EDIT$.B20
Monday, May 23, 1977 16:07:58

DISCARD ALL SPACES AND TABS.
DISCARDALLSPACESANDTABS.

REDUCE       SPACES     AND    TABS       TO  ONE    SPACE.
REDUCE SPACES AND TABS TO ONE SPACE.
```

You can also specify the sum of 2 or more integers in the table for a multiple effect. For example:

```
00010 PRINT "TYPE THE INPUT STRING";\INPUT LINE A$
00020 B$=EDIT$(A$,80%)
00030 PRINT "B$ = ";B$
00040 END

READY
RUN

EDIT$1.B20
Monday, May 23, 1977 16:08:39

TYPE THE INPUT STRING ? "THIS IS          MY      [PPN]."
B$ = "THIS IS MY (PPN)."
```

In line 20, the expression%, 80% is a combination of 16% and 64%.

7.4 CONVERSION FUNCTIONS

BASIC provides several string functions to do string to numeric and numeric to string conversions. ASCII and CHR$ allow you to convert a one character string to the character's ASCII number and vice versa. These functions are often useful in analyzing the characters in a string. The VAL and STR$ functions convert a string representation of a number to the number and vice versa. You should use them when you want to input a numeric value as a string or to print a number without the spaces around it.

7.4.1 Character and ASCII Code Conversions (ASCII and CHR$)

The ASCII function has the following format:

ASCII(string)

where:

string is either a string constant, a string variable, or string expression.

The ASCII functions returns the decimal ASCII value of the first character in the string specified. For example, ASCII ("D") is equal to 68, the decimal ASCII value of D.

You can also use a string variable as an argument:

```
00010 A$="DOG"
00020 PRINT ASCII(A$)

READY
RUNNH
 68
```

This program prints the value of the first character in A$.

The ASCII function returns an integer value.

7.4.2 Converting the ASCII Code to a Character

The CHR$ Function – Use the CHR$ function to create strings from ASCII values. The CHR$ function returns a 1-character string having an ASCII value of the specified expression. The format of the function is:

CHR$ (expression)

Only one character is generated at a time.

The expression must be zero or greater, and arguments greater than 255 are treated modulo 256.

Consider the following example:

```
00010 REM THIS PROGRAM WILL RETURN AN INPUT LETTER AND THE 2
00020 REM FOLLOWING IT ALPHABETICALLY
00030 PRINT "ENTER A LETTER A THROUGH Z, ENTER END WHEN FINISHED."
00040 PRINT "LETTER","NEXT LETTER","3RD LETTER"
00050 INPUT X$
00055 IF X$ = "END" GOTO 999
00060 IF X$ < "A" GOTO 190
00070 IF X$ > "Z" GOTO 190
00080 FOR F = ASCII(X$)  TO ASCII(X$)+2 !RETURNS ASCII VALUE OF X$
00090 IF CHR$(F) > "Z" GOTO 150
00100 PRINT CHR$(F),
00110 NEXT F
00120 PRINT
00130 GOTO 50
00150 PRINT "END OF ALPHABET"
00160 GOTO 50
00190 PRINT "ENTRY IS NOT A LETTER A THROUGH Z"
00210 GOTO 30
00999 END

READY
```

```
RUNNH
ENTER A LETTER A THROUGH 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 THROUGH Z
ENTER A LETTER A THROUGH Z, ENTER END WHEN FINISHED.
LETTER          NEXT LETTER    3RD LETTER
 ? END
```

7.4.3 Converting an Integer to RADIX-50 (RAD)

The RAD function has the following format:

RAD(expression%)

where:

expression% represents an integer constant that you supply.

The RAD function converts the integer you specify to RADIX-50. RADIX-50 is a character set similar to the ASCII code. See the User's Guide for more information on RADIX-50.

7.4.4 The CHANGE Statement

The CHANGE statement converts a string of alphanumeric characters into their ASCII decimal values and a list of decimal numbers into a string of alphanumeric characters (see Appendix B for the ASCII Table).

The CHANGE statement has the following format:

CHANGE list TO string variable

or

CHANGE { string variable / string expression } TO list

where:

list is a numeric or integer variable representing a 1- or 2-dimensional array of decimal values.

In the first format, the CHANGE statement converts a list of integers (real numbers are truncated) into a string of characters. The length of the string is determined by the value found in element 0 of the list. For example:

```
00010 FOR I = 0 TO 5
00020 READ A(I)
00030 NEXT I
00040 DATA 5,65,66,67,68,69
00050 CHANGE A TO A$
00060 PRINT A$
00070 END

READY
```

```
RUNNH
ABCDE
```

The CHANGE statement uses the first value in the list (5) to determine the length of the character string. It then converts the next 5 values into their ASCII representations (see Appendix B).

In the second format, the CHANGE statement converts a string of characters into a list of integers. The length of the string determines the value placed in element 0 of the list. For example:

```
00010 DIM A(50)
00020 READ A$
00030 CHANGE A$ TO A
00040 FOR I = 0 TO A(0)
00050 PRINT A(I);
00055 NEXT I
00060 DATA ABCDEFGHIJKLMNOPQRSTUVWXYZ
00070 END

READY
RUNNH
 26  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81
 82  83  84  85  86  87  88  89  90
```

Notice that A(0) is equal to 26 because there are 26 characters in the string.

7.4.5 Numbers and Their String Representation (VAL and STR$)

Two functions VAL and STR$ convert numbers to their string representation and vice versa.

Consider these programs:

String Representations

```
00010 PRINT "25"
00020 PRINT "25+1"

READY
RUNNH
25
25+1
```

Numbers

```
00010 PRINT 25
00020 PRINT 25+1

READY
RUNNH
 25
 26
```

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

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

VAL(string expression)

where:

string expression	may contain the digits 0 through 9, the letter E (for E format numbers) and the symbols "+", "-", and ".", and must be a string representation of a number.

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 have been printed by a PRINT statement, but without a leading or trailing space.

Consider the following example:

```
00005 PRINT "PROGRAM TO CALCULATE 5% INTEREST"
00010 PRINT "TYPE IN AMOUNT";
00020 INPUT M$
00030 IF POS(M$,"$",1)<>1 THEN 1000
00040 A$=SEG$(M$,2,LEN(M$))
00050 M = VAL(A$)
00060 I = .05 * M
00070 I$ = STR$(I)
00080 I$ = SEG$(I$,1,2+POS(I$,".",1))
00090 PRINT "5% INTEREST OF ";M$;
01000 PRINT " IS $";I$
99999 END

READY
RUNNH
PROGRAM TO CALCULATE 5% INTEREST
TYPE IN AMOUNT ? $100.00
5% INTEREST OF $100.00 IS $ 5
```

7.5 DATE AND TIME FUNCTIONS

The following table describes the various date and time functions (24 hour clock) available in BASIC.

Table 7-2 The Date and Time Functions

Function	Meaning	Example

CLK$	returns the current time of day as an 8-character string of the form hh:mm:ss.	`10 PRINT CLK$`
		`RUNNH`
		`16:43:53`

DAT$	returns the current date as dd-mmm-yy.	`10 PRINT DAT$`
		`RUNNH`
		`17-Jun-77`

DATE$(0%)	returns the current date in the form mm/dd/yy.	`10 PRINT DATE$(0%)`
		`RUNNH`
		`6/17/77`

DATE$(n%)	returns a specific day you specify with the integer constant supplied for n%. The formula for n% is n% = the day of the year + (the number of years since 1970*1000). If you only specify the day of the year, the date will be 1970, unless n%=0.	`10 PRINT DATE$(126)`
		`20 PRINT DATE$(6168)`
		`RUNNH`
		`6-May-70`
		`16-Jun-76`

Table 7-2 (Cont.) The Date and Time Functions

Function	Meaning	Example
TIME$(n%)	returns a string corresponding to the time of day n% minutes before midnight, unless n%=0.	10 PRINT TIME$(1) 20 PRINT TIME$(1440) 30 PRINT TIME$(721) RUNNH 23:59 00:00 11:59
TIMES(0%)	returns the current time of day as a character string of the form hh:mm.	10 PRINT TIME$(0%) RUNNH 16:45
TIME(0)	returns the clocktime, in seconds, since midnight, as a floating-point number.	10 PRINT TIME(0) RUNNH 60349
TIME(1%)	returns the CPU time used by the current job, in tenths of seconds.	10 PRINT TIME(1%) RUNNH 28
TIME(2%)	returns the connect time (during which you are logged-in) for the current job, in minutes.	10 PRINT TIME(2%) RUNNH 274

7.6 USER-DEFINED FUNCTIONS – THE DEF STATEMENT

In some programs you may want to execute the same sequence of statements in several places. You can define a sequence of operations as a user-defined function and use this function like you use the functions BASIC provides, such as SIN and SEG$. There are two ways of defining functions:

1. single-line DEF statement
2. multi-line DEF statement

7.6.1 Single-Line DEF

Single-line DEFs have a function name consisting of the letters FN followed by 1 to 29 letters, digits, or periods optionally followed by a % or a $. If the function name ends in a %, then it returns an integer. If the function name ends in a $, then it returns a string. If the function name does not end in either a % or a $, then it returns a floating point number. Therefore, the function name can have a total of 33 characters.

Legal User-Defined Function Names	Illegal User-Defined Function Names
FN	NF1
FNC%	FN A2
FNR.B$	FNA%$

The format of the single-line DEF statement is:

DEF FNa [[(b1,b2,b3,. . .bn)]] =expression

where:

a	is 1 to 29 letters, digits, or periods followed by an optional percent sign (%) or dollar sign ($) to represent an integer or string value.
(b1,b2,b3, . . .bn)	These can be integer, floating point, or string variables.
expression	is evaluated every 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 floating point and the function name is integer or vice versa, then the expression is converted to the type specified by the function name.

After the function has been defined, it can be called, or evaluated. The format for calling the function is:

FNa $\left\{\begin{matrix}\%\\\$\end{matrix}\right\}$ (expression1 [[,expression2, . . . ,expression5]])

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

When the function is evaluated, BASIC substitutes the values for the dummy variables in the DEF statement and then evaluates the expression and returns the result.

Consider the following two programs:

Program #1

```
00010 DEF FNS(A) = A^A
00020 FOR I = 1 TO 5
00030 PRINT I,FNS(I)
00040 NEXT I
00050 END

READY
RUNNH
 1            1
 2            4
 3            27
 4            256
 5            3125
```

Program #2

```
00010 DEF FNS(X) = X^X
00020 FOR I = 1 TO 5
00030 PRINT I,FNS(I)
00040 NEXT I
00050 END

READY
RUNNH
 1            1
 2            4
 3            27
 4            256
 5            3125
```

These two programs produce the same output. The actual names of the arguments in the DEF statement have no significance; they are strictly dummy variables. But the data types of the variables are significant. If the DEF statement specifies a string variable, then the corresponding argument must be a string. If the DEF statement specifies a numeric variable, then the corresponding argument must be numeric. BASIC converts, as necessary, a numeric argument to the type (floating point or integer) specified by the variable in the DEF statement.

The defining expression can contain any constants, variables, BASIC-supplied function, or any other user-defined function except the function you are defining. For example:

```
10 DEF FNA(X) = X^2+3*X+4
20 DEF FNB(X) = FNA(X)/2 + FNA(X)
30 DEF FNC(X) = SQR(X+4)=1
```

You can include any variables in the defining expression. If the expression contains variables that are not in the dummy variable list, they are not dummy variables. That is, when the user-defined function is evaluated, the variables have the value currently assigned to them.

Consider the following example:

```
00010 DEF FNB(A,B)=A+X^2
00020 X = 1                          !ASSIGN VALUE TO X
00030 PRINT FNB(14,87)               !EVALUATE FUNCTION
00040 X = 2                          !CHANGE VALUE OF X
00050 PRINT FNB(25,32)
99999END

READY
RUNNH
 15
 29
```

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:

```
00010 DEF FNA(X) = 4+2   !NOTE THIS FUNCTION ALWAYS RETURNS
00020 LET R = FNA(10)+1  !A VALUE OF 6 NO MATTER WHAT
00030 PRINT R            !THE VALUE OF THE ARGUMENT IS.
00040 END

READY
RUNNH
 7
```

Consider the following example:

```
00001 REM MODULUS ARITHMETIC PROGRAM
00005 REM FIND X MOD M
00010 DEF FNM(X,M)=X-M*INT(X/M)
00020 REM FIND A+B MOD M
00025 DEF FNA(A,B,M)=FNM(A+B,M)
00030 REM FIND A*B MOD M
00035 DEF FNB(A,B,M)=FNM(A*B,M)
00045 PRINT
00050 PRINT "ADDITION AND MULTIPLICATION TABLES MOD M"
00055 PRINT "GIVE ME AN M";\INPUT M
00060 PRINT\PRINT "ADDITION TABLES MOD";M
00065 GOSUB 800
00070 FOR I = 0 TO M-1
00075 PRINT I;" ";
00080 FOR J = 0 TO M-1
00085 PRINT FNA(I,J,M);
00090 NEXT J\PRINT\NEXT I
00100 PRINT\PRINT
00110 PRINT "MULTIPLICATION TABLES MOD";M
00120 GOSUB 800
```

Continued on next page

```
00130 FOR I = 0 TO M-1
00140 PRINT I;" ";
00150 FOR J = 0 TO M-1
00160 PRINT FNB(I,J,M);
00170 NEXT J\PRINT\NEXT I
00180 GOTO 99999
00800 REM SUBROUTINE FOLLOWS
00810 PRINT\PRINT TAB(4);
00820 FOR I = 0 TO M-1
00830 PRINT I;\NEXT I\PRINT
00840 FOR I = 1 TO 3*M+4
00850 PRINT "-";\NEXT I\PRINT
00860 RETURN
99999 END
```

```
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
```

7.6.2 Multi-Line Function Definitions

Some calculations may require more than a one line arithmetic expression as in the single line DEF statement. BASIC's multi-line DEF statement allows you more flexibility in defining complicated function values.

The multi-line DEF statement has the following format:

DEF FNa [[(b1,b2,b3, . . .)]],[[c1,c2,c3. . .]]

where:

a	represents 1 to 29 letters, digits, or periods followed by an optional percent sign (%) or dollar sign ($) to represent an integer or string function value.
(b1,b2,b3, . . .)	represents the dummy argument list.
c1,c2,c3, . . .	represents a list of variables local to the function definition. This list is optional.

Single and multi-line DEF statements are similar in format. However, multi-line DEFs do not have the equal sign expression on the first line. Instead, the function name must appear in a defining position (such as an assignment statement) within the function definition. Otherwise, the function value will be set to 0 or null string.

Multi-line functions can have from zero to any number of parameters.

The variables specified in the function definition (or DEF statement) can be used only within the body of the function. Any variable referred to in the definition that is not a local variable, refers to the variable of the same name outside the body of the DEF. This means that variables in the main program are global as opposed to variables specified in the DEF statement.

The FNEND statement signals the physical and logical end of the function definition. The FNEND statement has the following format:

```
FNEND
```

When BASIC executes the FNEND statement, it returns the function value to the calling statement.

You can also end a function definition with the FNEXIT statement. FNEXIT has the following format:

```
FNEXIT
```

The FNEXIT statement is equivalent to a GOTO n where n is the line number of the FNEND for the current multi-line DEF. The FNEXIT statement is legal only inside a multi-line DEF.

Most statements can be used within the function definition (between the DEF and FNEND statements). However, multi-line DEFs are local to the main program or subprogram (Chapter 11) in which they are contained. No transfers are allowed into or out of a multi-line DEF. If you attempt to transfer into the body of a multi-line DEF, BASIC will execute the next statement following the FNEND statement and issue a warning message.

DATA statements are global throughout the program. Therefore, even though they may reside within a function definition, the main program can still access them and vice versa. DIM statements within a function are local to the function definition if they apply only to local variables.

You can place a multi-line DEF anywhere in a program. The entire body of the multi-line DEF, as well as the single-line DEF, does not produce code in straight-line execution until it is called.

You call a function into action by using its name in a statement expression. With the name, you must include the actual argument list, one with the same number of arguments as in the DEF statement. The actual arguments can be constants, variables, array elements, or expressions. They must be the same data type as the dummy arguments they replace. (Whole arrays are not legal arguments.)

BASIC uses the actual arguments within the function to define the function value. Using dummy arguments in the multi-line DEF statement allows you to use the function definition many times with a different set of actual arguments.

The following example illustrates the use of the multi-line DEF statement:

```
00010 DEF FNX%(A,B),C
00020 IF A>B THEN C = 3.4
00030 REM C WAS INITIALIZED TO ZERO
00040 FNX% = A*B+C
00050 FNEND
00060 PRINT FNX%(3.1,2.3)
00070 END

READY
RUNNH
 10
```

BASIC ignores the function definition, lines 10 through 50, and begins execution at line 60. The PRINT statement calls the function with actual arguments to be substituted in the definition. A is larger than B; therefore, C is set equal to 3.4. At line 40, BASIC calculates the value to be 10.53. Because the function name is integer (%), the value returned to FNX% is 10.

7.6.2.1 Multi-Line DEF* – With the standard multi-line DEF argument passing mechanism, you cannot transfer into and out of a function definition retaining global values for variables. There is another method of writing multi-line DEFs that allows you to transfer from a function with global variables. This method also uses the DEF statement, however, an asterisk (*) has been added to the keyword.

DEF* FNa [[(b1,b2,b3, . . .)]],[[c1,c2,c3, . . .]]

The asterisks (*) tells BASIC that the BASIC-PLUS compatible form of the function definition is being used. This form allows you to include the GOTO, ON-GOTO, GOSUB, and ON-GOSUB statements within the body of the function to transfer outside the function definition. The variables you define during execution of the function are global while the function is still open (before BASIC reaches the FNEND statement).

The following example illustrates the DEF* method of multi-line DEFs:

```
00010 DEF* FNX(A)
00020 IF A<3 GOTO 40
00030 LET A=6\GOTO 100
00040 FNEND
00050 LET A=3
00060 LET C=FNX(4)
00070 LET D=FNX(2)
00080 PRINT A
00090 STOP
00100 PRINT A
00110 GOTO 40

READY
RUNNH
 6
 3
STOP at line 00090 of MAIN PROGRAM
```

Notice that the values printed for A are 6 and 3 rather than 3 and 3. Line 60 calls the function FNX with an actual argument of 4 to replace the dummy argument A. A is greater than 3 (condition tested on line 20); therefore, A is

assigned a new value of 6 (line 30). Then BASIC transfers out of the function definition to line 100 and prints the current value of A within the function, 6. Execution continues at the next line following the function call.

The second function call (on line 70) sends BASIC back to the function definition. This time, because A is less than 3 (2), BASIC transfers to the FNEND statement. Once the function ends, the value of A is no longer global. BASIC prints the value of A outside the function, 3, and then stops.

You can define multi-line functions either the standard (DEF) or BASIC-PLUS Compatible (DEF*) way.[1] However, all DEFs in the same program must be written the same way throughout. Functions defined either way must have argument lists agreeing in both data type and number.

NOTE

Although transfers into and out of a multi-line DEF* (BASIC-PLUS compatible) are permitted, random transfers may produce unpredictable results.

[1] Version 1 of BASIC-PLUS-2 on RSTS/E supports BASIC-PLUS compatible DEF* only for compatibility with RSTS/E BASIC-PLUS. Version 2 will support both.

CHAPTER 8
ARRAYS

8.1 DIMENSIONING AN ARRAY

Operations on arrays occur frequently; therefore, BASIC provides a special set of statements for array computations. These statements each contain the keyword MAT. The MAT statements apply to both lists and matrices, except where noted in the text. If you specify an array without subscripts (MAT A), the default is 2 dimensions.

Although every list has an element 0, and every matrix has a row 0 and a column 0, the MAT statements ignore these.

In a BASIC program, you reserve storage space in one of two ways:

1. Explicitly – with a DIM statement (2.6.1)
2. Implicitly – including an array in a program without the DIM statement

The MAT statements allow you to alter the number of elements in each row and column of an array as long as the total number of elements does not exceed the number previously defined. Changing the size of an array in this way is called redimensioning an array.

In addition to this capability, you can:

1. Perform arithmetic operations
2. Input array elements or entire arrays from the terminal
3. Read array elements from a DATA statement
4. Print array elements or entire arrays

8.2 INITIALIZING AN ARRAY

The MAT statements allow you to assign values to individual array elements. The values can be set to all 0's, all 1's, or 0's with 1's along the main diagonal.

This statement has the following format:

MAT name=value [[(DIM1,[DIM2])]]

where:

name	is an array dimensioned either implicitly or explicitly.
(DIM1,DIM2)	are new dimensions for the array. These dimensions are optional.
value	is one of the following:

VALUE	MEANING
ZER	sets the value of all elements in the array to 0. This condition is true of all arrays (except for those in a virtual array, MAP, or COMMON area, Sections 9.3, 10.1.4, and 11.2.1 respectively) when first created. (Does not set row 0 and column 0.)
CON	sets the value of all elements in the array to 1. (Does not set row 0 or column 0.)

IDN sets the value of all elements in the array to 0 except for those on the diagonal, which are set to 1. This is called an identity matrix. The matrix must be square. (Does not set row 0 or column 0.)

NUL$ sets the value of all elements in a string array to null string. (Does not set row 0 or column 0.)

The first three values apply to numeric and integer arrays, and the fourth applies to string arrays.

If you do not specify new dimensions with the (DIM1,DIM2) option, the existing dimensions remain unchanged.

Consider the following examples:

```
00010 DIM A(10,10), B(15), C(20,20)
00020 MAT A = ZER !SETS ALL ELEMENTS OF A = 0
00030 MAT B = CON(10) !SETS ALL ELEMENTS OF B = 1 AND REDIMENSIONS B
00040 MAT C = IDN(10,10) !CREATES AN IDENTITY MATRIX 10X10
00050 MAT PRINT A;
00060 MAT PRINT B;
00070 MAT PRINT C;
00080 END

READY
RUNNH

 0  0  0  0  0  0  0  0  0  0
 0  0  0  0  0  0  0  0  0  0
 0  0  0  0  0  0  0  0  0  0
 0  0  0  0  0  0  0  0  0  0
 0  0  0  0  0  0  0  0  0  0
 0  0  0  0  0  0  0  0  0  0
 0  0  0  0  0  0  0  0  0  0
 0  0  0  0  0  0  0  0  0  0
 0  0  0  0  0  0  0  0  0  0
 0  0  0  0  0  0  0  0  0  0


 1  1  1  1  1  1  1  1  1  1

 1  0  0  0  0  0  0  0  0  0
 0  1  0  0  0  0  0  0  0  0
 0  0  1  0  0  0  0  0  0  0
 0  0  0  1  0  0  0  0  0  0
 0  0  0  0  1  0  0  0  0  0
 0  0  0  0  0  1  0  0  0  0
 0  0  0  0  0  0  1  0  0  0
 0  0  0  0  0  0  0  1  0  0
 0  0  0  0  0  0  0  0  1  0
 0  0  0  0  0  0  0  0  0  1
```

To create an identity matrix with IDN, the array must be a square matrix.

8.3 MATRIX OPERATIONS

With the MAT statement, you can perform the following operations with arrays:

1. Assignment
2. Addition
3. Subtraction
4. Multiplication
5. Transposition
6. Inversion

Each MAT operation statement begins with the keyword MAT followed by an expression to be evaluated. You can assign the value of one array to another as in the following example:

```
10 MAT A=B
```

This statement sets each entry of array A equal to corresponding entry of array B. A is redimensioned to the size of array B.

You can also add and subtract arrays:

```
10 MAT A=B+C
20 MAT A=B-C
```

The first statement assigns the sum of arrays B and C to array A. The second statement assigns the difference between arrays B and C to array A. B and C can be either lists or matrices; however, they both must have identical dimensions.

The following statement multiplies two arrays:

```
10 MAT A=B*C
```

This statement causes array A to be set equal to the product of arrays B and C. A, B, and C must all be 2-dimensional arrays, and the number of columns in array B must be equal to the number of rows in C. BASIC redimensions A to the number of rows in B and the number of columns in C.

The following statements are illegal in BASIC:

```
10 MAT A=A*A
20 MAT A=A*B
30 MAT A=B*A
```

Components of array A are needed for the calculation of the expressions after they have already been destroyed. These illegal statements cause BASIC to print an error message.

However, this statement

```
10 MAT C=A*A
```

is legal if A is a square matrix.

You can also perform scalar multiplication of a matrix:

```
10 MAT A=(K)*B
```

where each entry in array B is multiplied by the value of K. K is any arithmetic expression and must be enclosed in parentheses.

Array A is redimensioned to array B provided enough space is reserved.

8.4 ARRAY INPUT AND OUTPUT

Elements in an array can be accessed with the following statements:

1. MAT INPUT
2. MAT PRINT
3. MAT READ

8.4.1 MAT INPUT Statement

The MAT INPUT statement enters values for each element of a list or matrix. The MAT INPUT statement has the following format:

MAT INPUT array(s)

where:

array — can be one or several lists or matrices separated by commas.

The keyword MAT INPUT must have a space between the two words.

BASIC reads data from the terminal as with the normal input statement. The question mark (?) signals that BASIC is ready to accept input.

Unlike the INPUT statement, the MAT INPUT statement allows you to enter a variable number of values into an array. You need not supply the same number of elements requested in the MAT INPUT statement; you can include fewer elements but not more than requested.

You can also continue typing data on more than one line with the continuation character, the ampersand (&). You terminate the input stream with a line terminator.

The values you type are entered into successive array elements in row order starting with the first element. If you type a variable number of values, you can determine the number of rows and columns you filled with the two variables NUM and NUM2.

If the array is a list, NUM is set equal to the number of elements you enter. If the array is a matrix, NUM is the number of rows you enter and NUM2 is the number of elements in the last row. By printing these variables, you can see the size of the array.

If you specify more than one array in the MAT INPUT statement, only the last one can have a variable number of elements. You can also redimension an array by specifying a new size in the MAT INPUT statement.

The following is an example of the MAT INPUT statement:

```
00010 DIM A(5)
00020 MAT INPUT A

READY
RUNNH
 ? 1,2,3,4,5
```

You cannot include a string constant within the MAT INPUT statement as you can in the INPUT statement. You can print the results of your input with the MAT PRINT statement.

8.4.2 MAT PRINT Statement

The MAT PRINT statement has the following format:

MAT PRINT array(s)

where:

array(s) without subscripts causes the printing of the entire array. Subscripted array causes the maximum size of the array (defined by the subscript) to be printed. (It does not redimension the array.)

MAT PRINT must have a space between the two words.

If you follow the array with a semicolon(;), the data values print in a packed fashion. If you follow the array with a comma (,), the data values print across the line with one value per print zone. If neither character follows the array, each element prints on a separate line. All but the last array in a list must have a comma or a semicolon, separating it from the next array on the list.

Each row of a matrix starts printing on a new line. You can print one-dimensional arrays (lists) in either row or column format. For example:

```
00010 DIM A(5)
00020 MAT INPUT A
00030 MAT PRINT A
00040 MAT PRINT A,
00050 MAT PRINT A;
00060 END

READY
RUNNH
 ? 5

 5
 0
 0
 0
 0


 5             0             0             0             0

 5   0   0   0   0
```

Notice that only one value was typed in response to the MAT INPUT. The remaining elements retain a value of 0.

When you specify more than one array, BASIC begins printing each array starting on a new line. (BASIC never prints row 0 and column 0 when using the MAT PRINT statement.)

8.4.3 MAT READ Statement

The MAT READ statement reads the values into elements of a 1- or 2-dimensional array from DATA statements. The MAT READ statement has the following format:

MAT READ array(s)

where:

array(s) without subscripts indicates that the entire array is read. Arrays with subscripts cause the array to be redimensioned.

MAT READ must have a space between the two words.

The maximum size of the array cannot exceed the previous dimensions.

BASIC reads the values from DATA statements in the same manner as the READ statement. (Row 0 and column 0 are not affected.) The DATA statement must contain enough data. You cannot input a variable number of elements.

Consider the following example:

```
00010 DIM B(2,2), C(2,2)
00020 MAT READ B
00030 MAT READ C
00040 MAT A=B+C
00050 MAT PRINT A;
00060 MAT PRINT B;
00070 MAT PRINT C;
00080 DATA 1,2,3,4,5,6,7,8

READY
RUNNH

 6   8
 10   12

 1   2
 3   4

 5   6
 7   8
```

8.4.4 MAT Functions TRN, INV, DET

The TRN function has the following format:

MAT array=TRN(array)

The function interchanges the dimensions of an array and renames it. A matrix with m rows and n columns will be renamed and redimensioned to n rows and m columns. For example:

```
00010 DIM B(3,5)
00020 MAT READ B
00030 MAT A=TRN(B)
00040 DATA 1,2,3,4,5
00050 DATA 6,7,8,9,10
00060 DATA 11,12,13,14,15
00070 MAT PRINT B;
00080 MAT PRINT A;

READY
```

```
RUNNH

 1   2   3   4   5
 6   7   8   9   10
 11   12   13   14   15

 1   6   11
 2   7   12
 3   8   13
 4   9   14
 5   10   15
```

Note that MAT A=TRN(A) is illegal.

If you want to find the determinant of a matrix, you must first find the inverse. Use the INV function for this purpose. The INV function is used as in the following example:

```
10 MAT A=INV(B)
```

The INV function allows matrix A to be the inverse of matrix B. (B must be a square matrix.) BASIC redimensions A to be the same size as B.

NOTE

Although matrix inversion does not operate on the elements of row 0 and column 0 of a matrix, BASIC does store intermediate results in these elements of an inverse.

Therefore, the values of the elements in row 0 and column 0 of an inverse matrix may change.

The function DET is available after the inversion. You can then use DET as a variable set equal to the value of the determinant of B. Consequently, you can obtain the determinant of a matrix by inverting the matrix and then noting the value of DET. For example:

```
00010 MAT A = INV(X)\D1=DET
00020 MAT B = INV(A)\D2=DET
00030 IF D1 = D2 THEN PRINT "RELATIONSHIP TRUE"
00040 PRINT D1
```

NOTE

If you specify a list rather than a matrix, BASIC cannot complete the inversion. Therefore, DET is set equal to 0.

CHAPTER 9
WORKING WITH FILES

9.1 FILES

There are three types of files in BASIC:

1. Terminal-Format files
2. Virtual Array files
3. Record files

To distinguish one file from another, you must label it with a file specification. The file specification usually contains the device name, the file name, and the file type. Because each system has its own constraints, refer to your User's Guide for details.

The following sections describe terminal-format and virtual array files. Record files are described in Chapter 10.

9.2 TERMINAL-FORMAT FILES

A terminal-format file is a collection of ASCII characters stored in lines of various lengths. The end of a line is determined by a line terminator, i.e., line feed. BASIC stores these ASCII characters, including spaces and line terminators, exactly as they would appear on the terminal; hence the name terminal-format file.

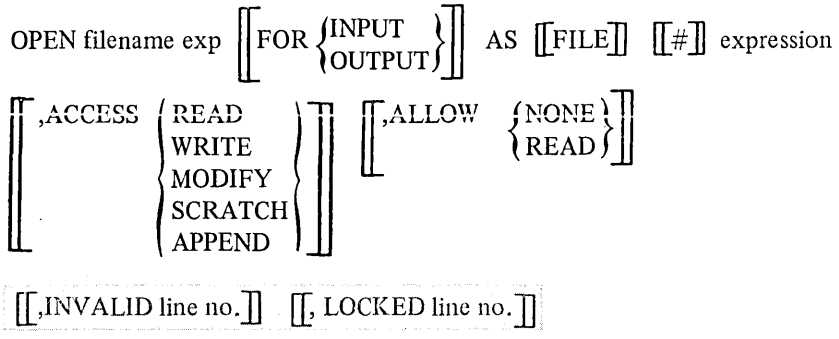
Terminal-format files are sequential access files. Sequential access files are those files that contain information that must be read or written one item after another from the beginning of the file. This means that you cannot retrieve an item from the file without first retrieving all the items preceding it.

BASIC has a file pointer that keeps track of where you are in the file. To add new items to an existing file without overwriting current information, you must read the entire file. This action places the file pointer at the end of the file where you can add data. Section 9.2.1 describes your options.

9.2.1 Opening Terminal-Format Files

Before you can access a terminal-format file, you must open it. The OPEN statement allows you to open a new file, or an existing file, and associate the file with a file number.

The OPEN statement has the following format:



where:

filename exp	is a file specification.
FOR INPUT	specifies an existing file.
FOR OUTPUT	specifies creation of a new file. If a file exists with the same file specification, the existing file is superseded.
expression	is the file number. It can be any numeric or integer expression with a value between 1 and the maximum allowed by your system.

FOR INPUT and FOR OUTPUT are both optional. If you omit this part of the statement from the OPEN statement, BASIC checks first for an existing file (FOR INPUT). If no file exists with the name specified, BASIC opens for the creation of a new file (FOR OUTPUT).

The ACCESS, ALLOW, INVALID, and LOCKED clauses are optional attributes that you can specify (in any order) when opening a file. The ACCESS clause defines both the position of the file pointer and the operations you can perform:

READ	The file pointer is at the beginning of the file. You can only read the file.
WRITE	The file pointer is at the beginning of the file. You can only add data to the file.
MODIFY	This is the default. The file pointer is at the beginning of the file. You can read and write to the file.
SCRATCH	The file pointer is at the beginning of the file. You have complete access to the file; read, write, and truncate the file.
APPEND	The file pointer is at the end of the file. You can only write to the file at this point.

The ALLOW clause defines what you allow other users to do to the file while you are using it.

NONE	is the default. No one can read or write data while you have the file open.
READ	allows others to read the file while you are using it.

Note that you cannot specify an ALLOW clause if you ACCESS SCRATCH.

The INVALID clause specifies the line number of an error-handling routine. This routine takes effect if the OPEN fails, i.e., you specified an illegal file specification. If you do not include INVALID, BASIC takes over the error handling.

The LOCKED clause also specifies the line number of an error-handling routine. This routine takes effect if the file is locked because another user specified ALLOW NONE. If you do not specify LOCKED, BASIC takes over the error handling.

Table 9-1 describes the results of specifying the keywords in the OPEN statement.

TABLE 9-1 OPEN Statement

Access	Initial File Position	I/O Operation
READ	beginning	read only
WRITE	beginning	write only
MODIFY	beginning	read or write
SCRATCH	beginning	read, write, truncate
APPEND	end	write

Consider the following example:

```
00010 OPEN "DATA1" FOR INPUT AS FILE 1,ACCESS APPEND
00015 N=5
00020 OPEN "MONEY" FOR OUTPUT AS FILE N
00030 END
```

Line 10 opens an existing file at the end and associates it with file 1. Line 20 creates a new file specified by MONEY and associates it with file 5. If a file named MONEY already exists, BASIC supersedes it with the new request. When you open a file and associate a file number to it, you use that number when referencing the file, e.g., DATA1 is #1, MONEY is # 5.

To save files for future use, you must close them. See Section 9.2.2.

9.2.2 Closing Terminal-Format Files

All programs that open files should close them before terminating execution. Most systems do not save files unless they are closed. An existing file with the same file specification may not be superseded until the new file is closed. Refer to your User's Guide to determine what happens on your system.

BASIC closes all files:

1. when executing a CHAIN statement Section 11.2
2. when executing an END statement
3. after executing the highest numbered line in the program.

Note that BASIC does not close files after executing a STOP statement.

A more specific way to close files is with the CLOSE statement. The CLOSE statement closes the files you specify and disassociates them from their file numbers. After you close a file, you cannot access it without reopening it.

Unlike the first three methods, the CLOSE statement allows you to specify which files you want closed.

The CLOSE statement has the following format:

CLOSE [[[#]] expression(s)]]

where:

expression(s) specifies one or more file numbers, separated by commas.

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

The following examples illustrate the CLOSE statement:

```
00010 CLOSE #1 !CLOSES FILE ASSOCIATED WITH FILE 1
00020 B=4
00030 CLOSE 2,B,6+1 !CLOSES FILE NUMBERS 2,4,7
00040 CLOSE !CLOSES ALL FILES
```

9.2.3 Reading Data From A Terminal-Format File

The INPUT # statement reads data stored in a terminal-format file and assigns a value to each variable listed.

The INPUT # statement has the following format:

INPUT # expression, variable(s)

where:

expression	is the file number of the terminal-format file. If the value of the expression is zero, data are input from the terminal rather than a file. The comma is required.
variable(s)	is one or more variable names separated with commas.

The INPUT # statement acts very much as the INPUT statement described in Section 3.1.1. However, the INPUT # statement requests data from a terminal-format file rather than from you. In order to INPUT # from a file, you must OPEN for ACCESS READ, MODIFY, or SCRATCH. If you OPEN with ACCESS APPEND, you must RESTORE # the file before you can read it. See Section 9.2.5.

Consider the following example:

```
00010 OPEN "NAMES" AS FILE #2, ACCESS READ !OPENS EXISTING FILE
00020 INPUT #2,A$,B !READS A STRING AND A NUMERIC FROM FILE
00030 PRINT A$,B !PRINTS RESULTS ON TERMINAL
00040 GOTO 20
00050 END

READY
RUNNH
SARAH           187.2
TONY            117.45
LORRAINE        200
JAY             89

? 11  End of file found on INPUT at line 00020 of MAIN PROGRAM
```

If this example had been written with an INPUT statement, BASIC would have stopped and printed a question mark to request data from you. Instead, the INPUT # read the data into the program from a previously stored terminal-format file.

BASIC starts reading data from the beginning of the file. If the line of data in the file contains more data than there are variables in the INPUT # statement, BASIC ignores the excess data. However, if there is not enough data on the line, BASIC looks for more data on the next line of the file. If you try to INPUT # from a new file or a file OPENED with either ACCESS APPEND or ACCESS WRITE, BASIC prints an error message.

9.2.3.1 The INPUT LINE # and LINPUT # Statements – The INPUT LINE # and LINPUT # statements have the following format:

INPUT LINE # expression, variable(s)

LINPUT # expression, variable(s)

where:

expression	is the file number of the file where the data resides. If the number is zero, BASIC inputs data from the terminal. The comma is required.
variable(s)	one or more string variables separated by commas.

The INPUT LINE # statement reads a string of characters from a terminal-format file into each respective string variable in the list. All characters on the input line including commas, quotation marks, and the line terminator are assigned to the string variable.

The LINPUT # statement also reads an entire line of data into the program; however, it does not include the line terminator.

The following example illustrates both statements:

```
00010 OPEN "TEST" AS #1
00020 PRINT #1,"DATA, FOR A PROGRAM."
00030 PRINT #1, "SECOND LINE"
00035 RESTORE #1
00040 INPUT LINE #1, A$
00050 PRINT A$
00060 LINPUT #1, A$
00070 PRINT A$
00075 CLOSE #1
00080 END

READY
RUNNH
DATA, FOR A PROGRAM.

SECOND LINE
```

Line 10 opens a file named TEST and associates it with FILE 1. Lines 20 and 30 write data into the file. The INPUT LINE # statement requests a line of data from the program. BASIC reads the entire line, including the line terminator, into the program. If an INPUT # statement had been used, BASIC would have read only "DATA" into the string variable A$.

The LINPUT # statement on line 60 requests another line of data. This time BASIC reads all characters into the program except for the line terminator.

9.2.4 Writing To A Terminal-Format File

The PRINT # statement writes data into the specified terminal-format file. The PRINT # statement has the following format:

PRINT # expression, list

where:

expression	is the file number of the terminal-format file. If the value of the expression is zero, BASIC prints the data on the terminal.
list	contains the items you want printed. The items can be any numeric, integer, or string expressions. Separate the items with commas or semicolons. The resulting ouptut format is the same as the simple PRINT statement.

If there are no items in the list, BASIC prints a blank line to the file. To PRINT # to a file, you must OPEN with ACCESS WRITE, MODIFY, SCRATCH or APPEND.

The PRINT # expression USING statement prints formatted data to a file (see Section 4.4).

Consider the following example that creates a terminal-format file from data stored in DATA statements:

```
00010 OPEN "NAMES" FOR OUTPUT AS FILE 1
00020 READ A$, A !READ DATA FROM PROGRAM
00030 IF A$="" THEN 100 !CHECK FOR LAST ITEM
00040 PRINT #1, A$;",";A !PRINT TWO ITEMS
00050 GOTO 20
00060 DATA "SARAH",187.2,"TONY",117.45
00070 DATA "LORRAINE",200,"JAY",89
00080 DATA "",0
00090 CLOSE #1
00100 END
```

After you run this program, the file NAMES contains the following:

```
SARAH       187.2
TONY        117.45
LORRAINE    200
JAY          89
```

9.2.5 Restoring A Terminal-Format File

The RESTORE # statement resets the specified terminal-format file to its beginning from the current position of the file.

The RESTORE # statement has the following format:

RESTORE # expression

where:

expression	is the file number of the terminal-format file.

After printing into a file, you can bring the file pointer back to the beginning with the RESTORE # statement.

```
00010 OPEN "NEW" AS FILE 1
00020 PRINT #1, 65;",";80;",";95
00030 RESTORE #1
00040 INPUT #1,A,B,C
00050 PRINT A,B,C
00060 CLOSE #1
00070 END

READY
RUNNH
 65            80            95
```

9.2.6 Checking for the End of a Terminal-Format File

The IFEND # statement has the following format:

IFEND # expression THEN { statement / line number }
,or
IFEND # expression GO TO line number

where:

expression — is the file number associated with the file.

With the IFEND # statement you check for the end of the file. If the file pointer is at the end, you can transfer control to another line of the program or execute a statement.

9.2.7 The IFMORE Statement

The IFMORE statement has the following format:

IFMORE #expression THEN { statement / line number }

or

IFMORE #expression GOTO line number

where:

file exp — is a file number of a terminal format file,

statement — is any legal BASIC statement except DATA, DEF, DIM, END, FNEND, IMAGE, NEXT, REM, SUB, SUBEND,-UNLESS, UNTIL, WHILE.

line number — is any valid line number in the program.

IFMORE tests whether the file pointer is at the end of the file specified. If NOT at the end, BASIC executes the statement or goes to the line number specified.

9.2.8 The NODATA Statement

The NODATA statement has the following format:

NODATA [[#]] expression, line number

where:

[[#]] expression, is a terminal-format file number.

line number — is any valid line number in the program.

When you specify NODATA with a file number:

```
NODATA #6, 45
```

BASIC checks for the end of the file, i.e., no data. If at the end of the file, BASIC transfers control to the specified line number.

If you do not specify a file number, (e.g., NODATA 110) BASIC tests to see if all the DATA for that program or subprogram has been exhausted. If there is no data left, control passes to the line specified in the NODATA statement.

9.2.9 Changing Margins

The MARGIN statement allows you to modify the margin setting of a terminal-format file or the margin setting of your terminal. The MARGIN statement has the following format:

MARGIN [[#]] expression, num exp

where:

file exp	is the file number of the terminal-format file. If you do not specify this argument, your terminal margin is changed.
num exp	is the numeric expression that determines the margin. (If it is a real number, it is truncated.)

The default margin is the current terminal width for the terminal.

Consider these examples:

```
00010 MARGIN 5
00020 PRINT "#"; FOR I = 1 TO 10
00030 END

READY
RUNNH
#####
#####
```

This example changes the terminal width to 5. To change back to the default, type:

```
10 MARGIN 0
```

The following example changes the margin of a terminal-format file:

```
10 MARGIN #4, 132
```

9.2.10 Setting Page Size

Normally, output to a terminal-format file and to a terminal are not divided into pages. The PAGE statement allows you to set a page size of any positive number of lines.

The PAGE statement has the following format:

PAGE [[#]] expression, num exp

where:

file exp	is the file number of a terminal-format file. If omitted, the PAGE setting affects the terminal.
num exp	is any numeric expression. It is truncated before the page size is set.

The page size remains in effect until:

1. The page size is set again with the PAGE statement.
2. Execution ends.
3. The file is closed.

At the end of program execution, the terminal is reset to its mode at program entry.

When a PAGE statement is executed, BASIC ends the current output line (if necessary) outputs a form-feed, and starts counting lines beginning with the next line of output. As soon as a new page is necessary, a form-feed is output.

9.3 VIRTUAL ARRAY FILES

A virtual array file, like a terminal-format file, is information stored on a system device (disk). Once you open a virtual array file, the similarity with terminal-format files ends. There is no need for the INPUT #, INPUT LINE #, LINPUT # PRINT #, or RESTORE # statements with virtual array files. You access elements in a virtual array exactly as you access elements in an array in memory. (See Sections 2.6 and 2.6.1.) In fact, you can use virtual arrays just as you would regular arrays, see Chapter 8.

Virtual array files are random access files. You can read or write any element in the file no matter where it is located. The last element in a virtual array can be accessed as quickly as the first. Contrast this with a terminal-format file where you must read the entire file to get to the last element.

When BASIC stores data in a virtual array file, it does not convert them to ASCII characters but rather stores them in the internal binary representation. Consequently, there is no loss of precision caused by data conversion.

You must define storage space for a virtual array file just as you do for a regular array. The DIM # statement (Section 9.3.1) allows you to set parameters for the file. Unlike arrays in memory, you must specify the maximum character length of strings in a virtual array file. Strings longer than the maximum are truncated. Strings shorter than the maximum are padded with trailing nulls.

9.3.1 Dimensioning A Virtual Array File

To use a virtual array file, you must first define its size with a DIM # statement. The DIM # statement has the following format:

DIM # num constant, array(s) [[=number]]

where:

num constant	is the file number associated with the virtual array file.
array(s)	is one or more 1- or 2-dimensional arrays separated by commas.
=number	is the maximum length of a string array if any are specified. The default is 16 characters.

For example:

```
10 DIM #2, A(15,20),B(50),C$(18)=10
```

The DIM # statement establishes the number of subscripts allowed for each virtual array, and the maximum values for each. In addition, the DIM # statement allocates all space for the virtual arrays associated with a particular file number. Storage allocation always starts at the beginning of the file. Therefore:

```
100 DIM #1, A(100),B(100)

10 DIM #1, A(100)
20 DIM #2, B(100)
```

do not perform the same function. Line 100 allocates 202 elements on file number one. While lines 10 and 20 allocate only 101 elements on file number 1. Each element has two names in this case.

When you specify a virtual array of strings, you should indicate the maximum length of each string. If no maximum is specified, the default is 16.

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

9.3.2 Opening and Closing Virtual Array Files

To open a virtual array file, you use the OPEN statement with a few variations. The virtual array OPEN statement has the following format:

OPEN filename exp [[FOR {INPUT / OUTPUT}]] AS [[FILE]] [[#]] expression

, [[ORGANIZATION]] VIRTUAL

[[,ACCESS {READ / MODIFY / WRITE}]] [[,ALLOW {NONE / READ / MODIFY[1] / WRITE[1]}]]

[[,INVALID line no.]] [[,LOCKED line no.]]

where:

filename exp	is a file specification.
FOR INPUT	specifies an existing file.
FOR OUTPUT	specifies the creation of a new file.
expression	is the file number. It can be any numeric or integer expression with a value between 1 and the system maximum.
ORGANIZATION VIRTUAL	specifies a virtual array file. The keyword ORGANIZATION is optional.
ACCESS READ	allows read only.
ACCESS WRITE	allows write only.
ACCESS MODIFY	allows read and write operations. This is the default.
ALLOW NONE	No simultaneous access.
ALLOW READ	allows others to read while you have the file. This is the default.

[1] This feature is not available on the DECSYSTEM-20 until Version 2.

ALLOW MODIFY[1]	allows others to read and write while you have the file.
ALLOW WRITE[1]	allows others to write while you have the file.
INVALID line no.	specifies a line in the program where your error-handling routine resides.
LOCKED line no.	specifies a line in the program where your error-handling routine resides.

The ORGANIZATION clause determines the file to be a virtual array as opposed to a terminal-format file. The ORGANIZATION clause is not allowed in a terminal-format OPEN statement. The rest of the attribute clauses may be specified in any order.

Consider these two program lines:

```
10 DIM #2, F(100,20)
20 OPEN "VARAY" FOR OUTPUT AS FILE #2, VIRTUAL
```

This program opens a virtual array file as FILE 2 and allocates 1,111 elements of storage space, i.e., 100 x 10 plus the 0 elements.

As an example of a use of virtual array files, consider the problem of an information retrieval system for a small organization. Assume there are 1000 employees each needing a 255-character record containing the name, home address, home phone, work station and phone extension. If this information is maintained in a terminal-format file, it would take a long time to locate the information for any employee and it would be impossible to update. Alternatively, these records can be maintained in a virtual array file. In this case some index is needed to associate a particular employee with a record.

In the following example, an index file containing badge numbers is used to find the record in the master file. The employee's badge number is in the same position in the index file as the record is in the master file. It is much faster to search through the index file because the data elements are much shorter, and less time is spent reading data from the file. This example program prints the employee's name based on the badge number.

```
00010 DIM #1,B%(1000)
00020 DIM #2,B$(1000)=255
00030 OPEN "BADGE" AS FILE 1,VIRTUAL
00040 OPEN "MASTER" AS FILE 2,VIRTUAL
00050 PRINT "WHAT IS THE BADGE NUMBER";
00060 INPUT N
00070 FOR I%=1 TO 1000
00080 IF B%(I%)=N THEN 200
00090 NEXT I%
00100 PRINT "NO SUCH EMPLOYEE"
00110 GOTO 99000
00200 PRINT "NAME IS"; SEG$(B$(I%),10%,30%)
99000 CLOSE #1,#2
99999 END
```

To close a virtual array file, use the CLOSE statement described in Section 9.2.2.

9.4 FILE RENAMING AND DELETION

The following sections describe the process of renaming a file and deleting a file from storage.

[1] This feature is not available on the DECSYSTEM-20 until Version 2.

9.4.1 The NAME-AS Statement

The NAME-AS statement has the following format:

NAME string 1 AS string 2

where:

string 1 is the file specification of the file to be renamed.

string 2 is the new file specification.

For example:

```
10 NAME "MONEY" AS "ACCNTS"
```

This statement changes the file named MONEY to ACCNTS.

The NAME-AS statement does not alter the contents of the file. It renames the first file specified to that of the second file without changing the file number. If you use the NAME-AS statement on an open file, the new name does not take effect until the file is closed. See your User's Guide for system dependent file specifications.

9.4.2 The KILL Statement

The KILL statement has the following format:

KILL string expression

where:

string expression is the file specification of the file you want deleted from storage.

After you delete a file, you cannot open it or access it in any way. For example:

```
10 KILL "DATA"
```

deletes the file DATA from storage.

CHAPTER 10
RECORD I/O

10.1 RECORD FILES

In addition to the two files discussed in Chapter 9, terminal-format and virtual arrays, BASIC provides another method for storing information, the record file. A BASIC record file is a collection of related data stored in the form of records. You determine the size and content of the records and the structure and access properties of the file.

Programs can write records into a file and subsequently retrieve them. Each record is treated as a separate unit. All input and output is performed on a record-by-record basis via a buffer between the file and the program.

In order to write BASIC programs that deal with record files, you need to establish the following:

1. File organization
2. Access method
3. Record format
4. Record mapping
5. File operations
6. Record operations

You also have the option of dynamically mapping I/O buffers.

These topics are explained in the following sections. For further information on using record files, see the *BASIC-PLUS-2 User's Guide* for your system.

10.1.1 File Organization

The manner in which BASIC stores and retrieves records in a file is determined by the structure of the file. In BASIC, the structure of a file is known as the organization. When you create the file, you specify its organization. The organization, in turn, determines the operations and access methods that you can use on the file. The three organizations you can specify are:

1. Sequential
2. Relative
3. Indexed

A Sequential file contains records that are stored in series. The order in which the records occur in the file is always the order in which they are written to the file. To read a particular record in the file, for example the 15th record, a program must open the file and successfully read the first 14 records before accessing the desired record.

Consequently, records can be added only to the end of a Sequential file because the location of each record is fixed in relation to the record preceding and succeeding it. Sequential files are allowed on disk or magnetic tape.

A Relative file contains records that are stored in numbered locations. BASIC structures the file into a series of record positions with each position capable of containing a single record. The number associated with a position represents its location relative to the beginning of the file. Thus, record number 1 occupies the first record position; record number 2 occupies the second record position, and so forth.

Access to a record can be made sequentially or randomly by record number. Relative files are allowed only on disk.

An **Indexed file** contains records that are stored according to a table. BASIC sets part of the file aside as an index in order to locate these records. Each record is retrieved based on the contents of a field, called a key, in the record.

When you create an Indexed file, you must specify which field in the record is to be used as the key. Access to a record can be made sequentially or randomly by reference to the key. Indexed files are allowed only on disk.

10.1.2 Access Methods

The methods that you use to store or retrieve records in a file are called access methods. The access method allowed on a particular file is determined by the file's organization. BASIC allows you to specify one of two access methods:

1. Sequential
2. Random

Sequential access indicates that records are accessed in serial order. Random access indicates that records are accessed by record number.

Table 10-1 shows the relationship between file organization and access methods.

Table 10-1 Access Methods

File Organization	Access Methods	
	Sequential	**Random**
Sequential	yes	no
Relative	yes	yes
Indexed	yes	yes

10.1.3 Record Format

A BASIC program must specify the format of records within a file. The format of a record determines how a record physically appears in a file on a storage medium. BASIC allows you to specify one of three formats:

1. Fixed Length
2. Variable Length
3. Stream

Fixed-length record format refers to records that are all equal in size. Each record occupies an identical amount of space in the file.

Variable-length record format refers to records that are not necessarily equal in length.

A **stream format** file contains a series of contiguous ASCII characters. In this case, a record is defined as a set of characters delimited by a form feed, vertical tab, or line feed.

The record format you select is restricted by the file organization. Sequential files support all three formats. However, Relative and Indexed files permit only fixed and variable length record formats.

Note that specifying variable-length format for Relative files does not save space on the disk. Space is allocated for the maximum record size for each record position. Records that are smaller than the maximum use only part of the space available.

Table 10-2 shows the relationship between file organization and record format.

Table 10-2 Record Formats

File Organization	Record Format		
	Fixed	Variable	Stream
Sequential	yes	yes	yes
Relative	yes	yes	no
Indexed	yes	yes	no

10.1.4 Record Mapping

To access records in a file, you must establish a buffer for input and output. You can name the buffer and describe the characteristics of the records in a particular file with the MAP statement. The MAP statement specifies that certain variables are contained in the buffer.

The MAP statement has the following format:

MAP (name) [[{ALIGNED / UNALIGNED}]] element(s)

where:

(name)	is the name you give to the buffer. The length of the name is system dependent. Parentheses are optional.
ALIGNED UNALIGNED	is optional. These keywords refer to the way data is placed in the buffer. ALIGNED is the default. For more information, refer to the User's Guide.
element(s)	is a list of elements, separated by commas, defining the characteristics of the record. Each element represents a field in the record.

A legal element in a MAP statement can be any numeric, integer, or string variable, an entire array, or a FILL. (FILL, FILL%, FILL$, FILL$=n, FILL(n), FILL$(n)=m)

FILL acts as a space holder allowing you to mask parts of a record or hold space for future use. FILL$=m is a string of m characters and FILL$(n)=m is n strings of m characters.

The length of a string variable is specified by the syntax

A$=n

where:

A$ is the string variable.

n is the number of characters in the string. n must be a constant.

For example:

```
10 MAP (BUFF1) NAME$=25,SS%,FILL,AGE%
```

This statement sets up a buffer area named BUFF1 and describes four data fields:

1. A string field containing up to 25 characters
2. An integer field

3. A place holder
4. Another integer field

When specifying a string data field, you should define the number of characters in the field. The default is 16 characters. Strings in a string field are a fixed length. BASIC stores them left-justified and padded with blanks. For example:

```
00010 MAP (TEST) B$=7
00020 OPEN "FILE" AS FILE 1, SEQUENTIAL, ACCESS APPEND, MAP TEST
00030 B$='ABC'
00040 C%=LEN(B$)
00045 PRINT C%
00050 PUT #1
00060 CLOSE #1
00070 END

READY
RUNNH
 7
```

Although the value assigned to B$ is 3 characters long, the length of the string field contains those 3 characters plus 4 trailing blanks. Its length is 7.

NOTE

Variables that are parameters to subprograms are all passed by reference. Therefore, if an actual parameter is a variable that is MAPed, execution of a GET inside the subprogram will change the value of the dummy parameter.

The following rules apply to MAP statements in a BASIC program:

1. All MAP statements must appear before the OPEN statement in a program and before their variables are referenced.
2. You can have multiple maps with the same name. The largest buffer (longest element list) must be specified first. The first map sets up storage allocation.
3. The same variable can appear on the element lists of different map statements with the same map name. In this case, the variable must occur in the same position in each map. For example:

```
100 MAP (BUFF1) A,B,C
200 MAP (BUFF1) A,Q,C
```

4. If you specify an array in a MAP statement, you must dimension it in that statement. If the same array occurs in two different MAPs, the dimensions must be the same.
5. The length of a string field should be defined; otherwise, the default is 16 characters.
6. MAPs are local to the MAIN program or subprogram in which they are defined.

The MAP statement is referenced in the OPEN statement when you create a new file or access an existing file.

10.2 FILE OPERATIONS

When dealing with record files, you are either working with the file as a whole or working with an individual record in the file. The following sections describe how to

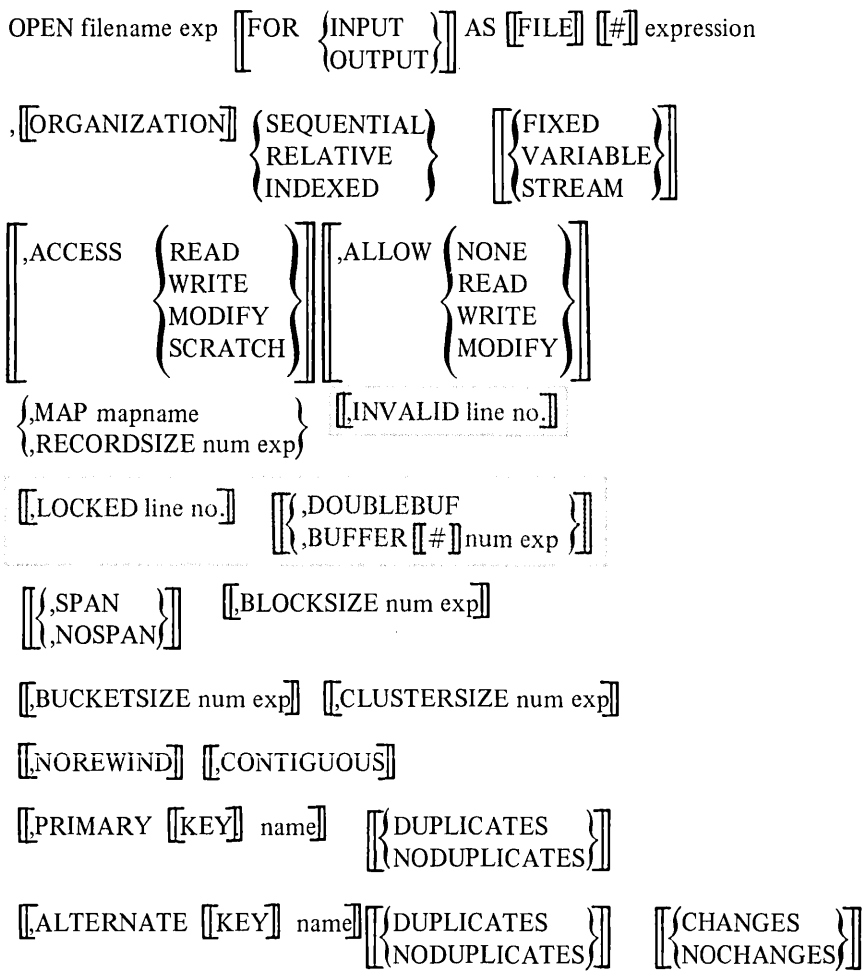
1. Create a new file – OPEN statement
2. Access an existing file – OPEN statement
3. Close a file – CLOSE statement
4. Return the file pointer to the beginning – RESTORE statement
5. Truncate an entire file – SCRATCH statement

10.2.1 Creating and Accessing a File

The OPEN statement enables you to create a new file or access an existing file. With this statement you can define, explicitly, all the important aspects of each data transfer operation including the structure of the file and its file sharing capabilities. You can also include the specification of error returns.

The syntax of the OPEN statement includes keywords that describe attributes of the file. These attributes are followed, in general, by a name, numeric expression, or line number, and you separate them with commas.

The following is the general syntax of the OPEN statement for a record file:



where:

filename exp	is a system dependent file specification.
FOR INPUT	requires that the specified file exist. If the file does not exist, an error results. This error causes the OPEN to return to the line specified by the INVALID clause.
FOR OUTPUT	creates a new file with the name you specify.

If you leave the FOR clause out entirely, BASIC searches for an existing file of the specified name. If the search fails, BASIC creates a new file.

AS [[FILE]] [[#]] expression
associates the file with a file number. File number 0 (user's terminal) is illegal.

,[[ORGANIZATION]] SEQUENTIAL
arranges the records in the file by order of input, i.e., in serial order.

,[[ORGANIZATION]] RELATIVE
arranges records by numbered position in the file.

,[[ORGANIZATION]] INDEXED
arranges records so that they can be accessed by reference to a keyed index.

[[FIXED]]
specifies that the records are a fixed length.

[[VARIABLE]]
specifies variable length records in the file. Note that if records are variable length, the buffer is padded with 0's (nulls) after a GET of a record that is smaller than the buffer. This format is the default for all 3 organizations.

[[STREAM]]
specifies ASCII stream records. (Sequential files only.)

[[,ACCESS READ]]
WRITE
MODIFY
SCRATCH
APPEND
specifies the operations that the current user can perform on the file.

READ
allows read only.

WRITE
allows write only.

MODIFY
allows read, write, delete, and update operations. This is the default for Sequential, Relative and Indexed files.

SCRATCH
allows full access; read, write, delete, update and truncate. (Note that a file cannot be accessed by multiple users if it is open with ACCESS SCRATCH.)

APPEND
allows write access at end of file.

[[,ALLOW NONE]]
READ
WRITE
MODIFY

defines what you allow other users to do to the file while you are using it.

NONE
specifies a protected file. This is the default for Sequential files.

READ
allows read only. This is the default for Relative and Indexed files.

WRITE
allows write only.

MODIFY
allows read and write access.

,MAP mapname
references a MAP statement. The map buffer you reference defines the buffer used to store the file's data temporarily. The MAP can also be used to define the record size.

,RECORDSIZE num exp
defines the maximum size of records (in characters) in the file. RECORDSIZE must be specified when no MAP clause is specified.

If you specify both the MAP clause and the RECORDSIZE clause in the same OPEN statement, the RECORDSIZE overrides the MAP even if the former is smaller. In this case, the RECORDSIZE is the size of the record, and the MAP is just a place to store it.

[[,INVALID line no.]]
specifies the line number of an error-handling routine. Control transfers to this line if the OPEN fails due to an illegal file specification.

[[,LOCKED line no.]]
also specifies the line number of an error-handling routine. Control transfers to this line if the OPEN fails because the file is locked (protected).

[[,DOUBLEBUF
,BUFFER [[#]] num exp]]
signifies the number of buffers used during file operations. The default is 2. DOUBLEBUF has no effect. It exists for compatibility with another version of BASIC.

[[,SPAN, NOSPAN]]
signifies that records are allowed to cross block boundaries. The default is SPAN.

[[,BUCKETSIZE num exp
,BLOCKSIZE num exp
,CLUSTERSIZE num exp
,NOREWIND
,CONTIGUOUS]]
These attributes are system dependent. Refer to BASIC-PLUS-2 User's Guide for more information.

[[,PRIMARY [[KEY]] name]]
is required for an Indexed file. It defines the name of the Primary index key. The size and location of this key is specified in the MAP statement. The name is one of the elements in the list. The key must be a string. Duplicates are allowed but CHANGES are not.

[[,ALTERNATE [[KEY]] name]]
allows you to optionally define the names of one to 254 Alternate index keys. Alternate keys must also be strings.

[[{NODUPLICATES}]]
[[{DUPLICATES }]]
NODUPLICATES is the default. If DUPLICATES is specified for a given key of reference, the file can contain more than one record with the same value for that key.

[[CHANGES]]
[[NOCHANGES]]
NOCHANGES is the default. If CHANGES is specified for a given key of reference, the value of the field for that key in a given record can be changed. Alternate keys may have changes but the primary key may not. Note that the combination CHANGES and NODUPLICATES is illegal.

The ORGANIZATION clause must be the first attribute specified. The severity of the error is system dependent. Refer to the User's Guide. The other attributes may be specified in any order.

The following sections describe the OPEN statement as it applies to each file organization.

10.2.1.1 Opening a Sequential File – The following syntax is used when opening an existing file or creating a new Sequential file:

OPEN filename exp [[FOR {INPUT / OUTPUT}]] AS [[FILE]] [[#]] expression

,[[ORGANIZATION]] SEQUENTIAL [[{FIXED / VARIABLE / STREAM}]]

[[,ACCESS {READ / WRITE / MODIFY / SCRATCH / APPEND}]] [[,ALLOW {NONE / READ / WRITE / MODIFY}]]

{,MAP mapname / ,RECORDSIZE num exp} [[,INVALID line no.]] [[,LOCKED line no.]]

[[{,DOUBLEBUF / ,BUFFER [[#]] num exp}]] [[,CLUSTERSIZE num exp]]

[[,BLOCKSIZE num exp]] [[{,SPAN / ,NOSPAN}]] [[,NOREWIND]]

[[,CONTIGUOUS]]

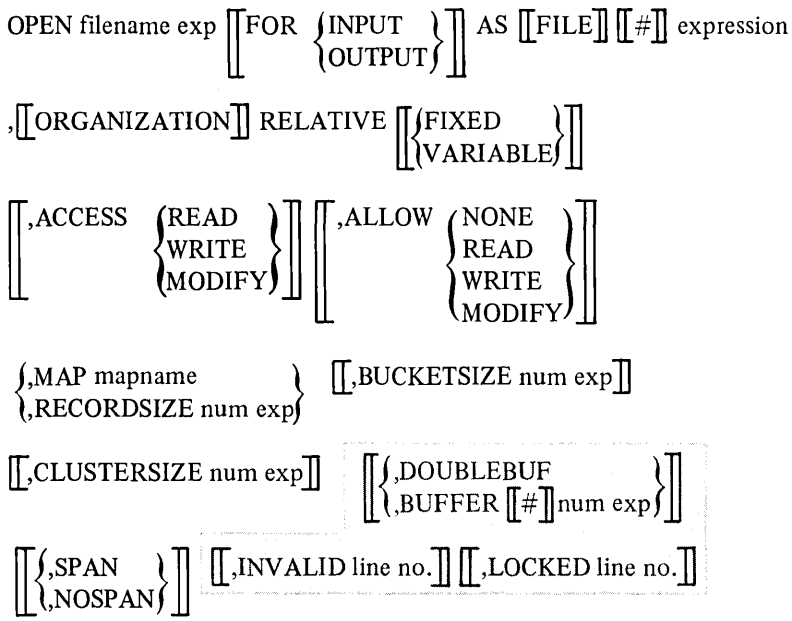
The following example opens a Sequential file:

```
20 OPEN "CASE" AS FILE #5 &
   ,ORGANIZATION SEQUENTIAL STREAM &
   ,ACCESS APPEND, ALLOW NONE, MAP MAP1 &
   ,INVALID 120, LOCKED 90
```

This statement opens an existing file named CASE, positions the file pointer at the end of the file, and associates the file with file number 5.

The SCRATCH statement allows you to truncate the entire file. This statement is only valid for a file OPENed with ACCESS SCRATCH. See Section 10.2.4.

10.2.1.2 Opening a Relative File – The following syntax opens a Relative file:



The following example opens a Relative file:

```
110 OPEN "FOO" FOR OUTPUT AS FILE #1 &
    ,ORGANIZATION RELATIVE FIXED, ACCESS MODIFY, &
    ALLOW READ, MAP TEST, LOCKED 230
```

This statement creates a new file name FOO and associates it with file number 1. Each record in the file has a fixed length. The user of the file has read and write access capabilities, while other people can only read records. BUCKETSIZE is system dependent. Refer to the *BASIC-PLUS-2 User's Guide*.

10.2.1.3 Opening an Indexed File – The following syntax opens an Indexed file:

OPEN filename exp [[FOR {INPUT / OUTPUT}]] AS [[FILE]][[#]] expression

,[[ORGANIZATION]] INDEXED [[{FIXED / VARIABLE}]]

[[,ACCESS {READ / WRITE / MODIFY}]] [[,ALLOW {NONE / READ / WRITE / MODIFY}]]

{,MAP mapname / ,RECORDSIZE num exp} [[,BUCKETSIZE num exp]] [[, LOCKED line no.]]

[[,CLUSTERSIZE num exp]] [[{,DOUBLEBUF / ,BUFFER [[#]] num exp}]]

,PRIMARY [[KEY]] name [[{DUPLICATES / NODUPLICATES}]] [[, INVALID line no.]]

[[,ALTERNATE [[KEY]] name]][[{DUPLICATES / NODUPLICATES}]] [[{CHANGES / NOCHANGES}]]

The PRIMARY key is mandatory for an Indexed file. The following example illustrates the opening of an Indexed file:

```
50 OPEN "ACCOUNT" FOR INPUT AS FILE #4 &
   ,ORGANIZATION INDEXED VARIABLE &
   ,ACCESS MODIFY, ALLOW NONE &
   ,PRIMARY B$, ALTERNATE WAGES$ &
   ,MAP BUFF1
```

This statement opens an existing file named ACCOUNT and associates the file with file number 4. The records are variable length. The primary index key is in the data field B$, and there is one alternate key named WAGES$. Note specifying CHANGES with NODUPLICATES is illegal.

10.2.2 Closing Files

Record files, as well as terminal-format and virtual array files, should be closed when no longer needed. The CLOSE statement, described in Section 9.2.2, is also valid for record files.

The CLOSE statement has the following format:

CLOSE [[file number(s)]]

where:

file number(s) represent one or several files separated by commas.

For example:

```
65 CLOSE #3
```

If you do not specify file numbers, BASIC closes all files.

10.2.3 Restoring a File

The RESTORE # statement allows you to bring the file pointer back to the beginning of the file without disturbing the data. All file organizations can use this feature.

The RESTORE # statement has the following format:

RESTORE # file number [[,KEY # num exp]]

where:

file number	is the file you want to reset.
,KEY # num exp	is for indexed files only. This allows you to establish a new key of reference.

For example:

```
25 RESTORE #6, KEY #0
```

This example brings the file pointer to the index table designated by 0. This is the Primary key.

10.2.4 Truncating a File

The SCRATCH statement truncates a file at the current file pointer. The SCRATCH statement has the following format:

SCRATCH [[#]] file number(s)

where:

file number(s) is one or more open files. Separate each file number with a comma.

SCRATCH erases the contents of the file but does not delete the file. To use the SCRATCH statement, the file must be OPENed, with ACCESS SCRATCH. See Section 10.2.1.

10.3 RECORD OPERATIONS

There are several operations that you can perform on individual records in a file, depending on its organization. Record file operations allow you to add, remove, examine, and modify the contents of a file. When writing into a file, a program builds records and passes them for storage in the file. When reading a file, a program requests records from the file. With BASIC, you can

1. Read a record – GET statement
2. Write a record – PUT statement
3. Locate a record – FIND statement
4. Replace a record – UPDATE statement
5. Remove a record – DELETE statement

The GET statement reads a record from the file into a buffer.

The PUT statement writes a new record from the buffer to the file.

The FIND statement locates the specific record in the file and points to it.

The UPDATE statement replaces an existing record with a new one. You must do a FIND or a GET before you can UPDATE.

The DELETE statement erases an existing record from the file. You must do a FIND or a GET before a DELETE operation.

The following sections describe your options in relation to each file organization.

10.3.1 Sequential Record Operations

The following are the operations you can perform on a sequentially organized file.

GET file exp
[[,LOCKED line no.]]
[[,INVALID line no.]]

PUT file exp
[[{,MAP line no.}]] [[,INVALID line no.]]
[[{,COUNT exp }]] [[,LOCKED line no.]]

UPDATE file exp
[[{,MAP line no.}]] [[,INVALID line no.]]
[[{,COUNT exp }]]

FIND file exp
[[,LOCKED line no.]][[,INVALID line no.]]

In a Sequential file, a GET operation is performed on succeeding records starting at the beginning of the file. Each successive GET statement retrieves the next record in the file and places it in the buffer identified by the MAP statement. If you retrieve a record that is smaller than the buffer, BASIC fills the buffer with nulls.

A PUT statement in a Sequential file writes the record from the buffer to the end of the file without truncating records. You can write only at the end of a Sequential file without truncating records. When you are writing to a file, you must specify the record size with a MAP or COUNT. The MAP line number in the PUT statement specifies the size of the RECORD to PUT. It cannot be used to specify a different buffer from that previously specified in the OPEN statement. If the MAP or COUNT clause is not specified, the record size is defined by the MAP or RECORDSIZE clause in the OPEN statement.

In order to replace an existing record with the UPDATE statement, you must first do a successful GET or FIND. You also must specify the size of the record by referencing the MAP line number or giving the actual size in characters with COUNT. The size of the new record must be the same as the one being replaced.

Because you can only access Sequential files sequentially, a FIND operation locates the next record in sequence.

Note that you cannot DELETE records in a sequential file.

10.3.2 Relative Record Operations

The following operations can be performed with a Relative file:

GET file exp
[[,RECORD num exp]]
[[,LOCKED line no.]] [[,INVALID line no.]]

```
PUT file exp      [[,RECORD num exp]]
                  [[{,MAP line no.  }]]
                  [[{,COUNT num exp}]]
                  [[,LOCKED line no.]]    [[,INVALID line no.]]

UPDATE file exp   [[{,MAP line no.  }]]
                  [[{,COUNT num exp}]]
                  [[,LOCKED line no.]]    [[,INVALID line no.]]

DELETE file exp   [[,INVALID line no.]]

FIND file exp     [[,RECORD num exp]]
                  [[,LOCKED line no.]]    [[,INVALID line no.]]
```

With Relative files, you are allowed random access as well as sequential access. Therefore, you can specify which record you want to GET and PUT. If you leave off the record number in the statement, BASIC will read, write, or locate the next record in sequence. Notice that the DELETE and UPDATE statements do not have [[,RECORD num exp]] as an option. The record number is already specified when you do the necessary GET or FIND operation.

Some record operations change the value of the record pointer and some do not. In a Relative file, a sequential GET and a sequential PUT each modify the value of the record pointer.

For example:

```
00100 GET #7, RECORD 2   !RANDOM RETRIEVES RECORD 2
00200 GET #7             !SEQUENTIAL RETRIEVES RECORD 3
00300 GET #7             !SEQUENTIAL RETRIEVES RECORD 4
```

A random GET operation also modifies the value of the record pointer; however, a random PUT does not. Consider the following example:

```
00300 GET #1,RECORD 15          !RANDOM RETRIEVES RECORD 15
00400 PUT #1,RECORD 20          !RANDOM WRITES RECORD 20
00500 PUT #1                    !SEQUENTIAL WRITES RECORD 16
```

Note that line 500 PUTs record 16 (not 21) because the random PUT in line 400 did not change the value of the record pointer.

A FIND operation is only relevant if the next operation is a GET, DELETE, or UPDATE. A PUT after a FIND invalidates the FIND; therefore, a subsequent GET retrieves the next record rather than the record located by the FIND.

In the following example line 600 PUTs record 11:

```
00400 GET #1,RECORD 10          !RETRIEVES RECORD 10
00500 FIND #1,RECORD 20         !LOCATES RECORD 20
00600 PUT #1                    !WRITES RECORD 11
```

However, in the example:

```
00700 FIND #1,RECORD 20         !LOCATES RECORD 20
00800 UPDATE #1                 !REPLACES RECORD 20
```

line 800 UPDATEs record 20.

10.3.3 Indexed Record Operations

The following operations deal with Indexed files only:

GET file exp [[,KEY #num exp {EQ / GT / GE} string]]
[[,LOCKED line no.]][[,INVALID line no.]]

PUT file exp [[{,MAP line no. / ,COUNT num exp}]]
[[,LOCKED line no.]][[,INVALID line no.]]

UPDATE file exp [[{,MAP line no. / ,COUNT num exp}]]
[[,INVALID line no.]]

DELETE file exp [[,INVALID line no.]]

FIND file exp [[,KEY #num exp {EQ / GT / GE} string exp]]
[[,LOCKED line no.]][[,INVALID line no.]]

In random access to an Indexed file, you supply a key previously defined in an OPEN statement as a PRIMARY or ALTERNATE key. To locate a specific record, specify one of the two key matches:

1. Exact key match
2. Approximate key match

With the exact key match, BASIC looks for the record that matches the value you assign to the key.

For example:

```
00010 MAP(FILE1) SURNAME$=20,GIVENNAME$=10,SSN$=9,ADDRESS$=40&
   ,ZIPCODE$=5
00020 MAP(FILE1) NAME$=30,ID$=9,ADDR$=45
00030 OPEN "ACCOUNT" FOR INPUT AS #5,INDEXED VARIABLE,MAP FILE1,&
   PRIMARY NAM$,ALTERNATE SSN$,ALTERNATE ZIPCODE$
00040 GET #5, KEY #1 EQ "013445695"
00050 GET #5,KEY #0 EQ "MURPHY"
```

The map at line 10 defines the record as follows:

SURNAME$ GIVENNAME$ SSN$ ADDRESS$ ZIPCODE$

The map at line 20 defines the record as follows:

NAMES$ ID$ ADDR$

The OPEN statement at line 50 defines the keys within the records as follows:

KEY#	Starting Position	Length
0 (PRIMARY)	0	30
1	30	9
2	79	5

When executing line 40, BASIC refers to the keys specified in the OPEN statement. KEY#1 is the first ALTERNATE key SSN$. This field is defined in the MAP at line 10. BASIC searches for an exact match of a record with "013446595" starting at position 30.

When executing line 50, BASIC again refers to the keys in the OPEN statement. KEY#0 is the PRIMARY key NAM$. This key is part of the record described in the MAP on line 20. BASIC searches an exact match of a record with a field "MURPHY" in starting position 0.

The second type of search, approximate key match, allows you to request the record closest to the value you specify. The proximity is determined by the ASCII collating sequence. The approximate key search allows your program to select either of the following relationships:

1. Equal to or greater than (GE)
2. Greater than (GT)

If the key requested does not exist, BASIC returns the record that contains the next higher key value. This allows you to retrieve records without knowing the exact key.

For example:

```
40 GET #5,KEY #0 GE "JONES"
50 GET #5,KEY #0 GT "ABRAMSON"
```

Line 40 defines the key as PRIMARY and searches for a data field that is greater than or equal to (GE) the value "JONES", e.g., "KNIGHT".

Line 50 also uses the PRIMARY key but searches for a data field greater than "ABRAMSON", e.g., "ADAMS". "ABRAMSON" is not an acceptable match in this case.

You can also affect a match by specifying a key value with fewer characters than were specified for the corresponding field in the record. The match occurs if the first characters in the field are identical to the key value.

If you do not specify a KEY, you effect a sequential GET according to the previous KEY specified. BASIC will then retrieve the next record in the index according to the ASCII collating sequence.

When you PUT to an Indexed file you merely specify

```
PUT file exp
```

BASIC places the record in the proper index.

In addition to read, write, and find operations, your program can delete any record in an Indexed file and update any record. However, during an update operation, be sure that the contents of the modified record do not change the Primary key value. You can change alternate key values if CHANGES is specified.

10.3.4 Record Locking

If you plan to allow file sharing (simultaneous access) by specifying ALLOW READ, ALLOW WRITE, or ALLOW MODIFY, you should be aware of the correlation between an I/O operation and the locked status of a record. Records are locked according to the attributes specified in the OPEN statement and according to the particular I/O operation you perform on the record. The ACCESS clause determines the effect the I/O operation has on the locked status of a record.

If you OPEN with ACCESS READ,

GET	locks the record only for the duration of the operation. The record is unlocked when the GET is completed.
FIND[1]	locks the record until a GET is completed or until another record is accessed with a FIND or GET operation.

If you OPEN with ACCESS WRITE, MODIFY, or APPEND,

GET	locks the record until another record is accessed by a GET, FIND, or PUT operation; or until the current record is UPDATEd or DELETEd.
FIND	locks the record until another record is accessed by a GET, FIND, or PUT operation; or until the current record is UPDATEd or DELETEd.

10.4 DYNAMIC MAPPING OF AN I/O BUFFER

The MAP statement (described in Section 10.1.4) defines the format of a record when that format can be specified at compile time. When this is not the case, the MOVE statement can be used to dynamically access the data in a record. For example, you can use MOVE to access a record in which the lengths of strings or arrays in the record are specified by fields at the beginning of the record. The MOVE statement associates the data in a record with the variables you specify in an I/O list. The format of the MOVE statement is:

MOVE [[{ALIGNED / UNALIGNED}]] {FROM / TO} file exp, I/O list

where:

ALIGNED UNALIGNED	is optional. These keywords refer to the way data is placed in the buffer. ALIGNED is the default. For more information, refer to the BASIC-PLUS-2 User's Guide.
FROM	moves the data from the buffer associated with the file number and places the data in the elements in the I/O list.
TO	moves the data from the elements in the I/O list and places it in the buffer associated with the file number.
file exp	is the file number associated with the file OPENed previously.
,	the comma is mandatory between the file exp and the I/O list.
I/O list	is a list of legal elements.

Legal elements in an I/O list are:

1. numeric, integer, string variables
2. arrays

[1] On RSTS/E, the record is unlocked after the FIND operation is completed.

3. array elements
4. fill specifiers

The length of a string may be defined in the I/O list, e.g., A$=n. The default length for MOVE TO is LEN(A$); the default for MOVE FROM is 16.

An array specified in a MOVE statement must have the following format:

A( ) list
A(,) matrix

Note that row zero and column zero are affected by the MOVE.

You specify an array element by name, i.e.,

A(25)

The following are examples of MOVE statements:

```
60 MOVE FROM #5, A$,B,FILL%,C( )
85 MOVE TO #5, A$,B,FILL%,C( )
```

Successive MOVE statements to or from the same file each start at the beginning of the buffer. The size of the buffer is not affected by MOVE's. If a MOVE only partially fills a buffer, the rest of the buffer is unchanged.

To retrieve a record from a file, first read the record with a GET statement. This places the record in the buffer. The buffer can be a system buffer or a buffer you have set up with a MAP statement.

Then a MOVE FROM places the data from the buffer into the elements in the I/O list. Once the data is associated with the elements, you can reference them in the program.

A MOVE TO moves the data from the elements in the I/O list to a buffer. To move the data into a file, you do a write operation with the PUT statement.

Consider the following program:

```
00010 OPEN "MOVE.DAT" AS FILE #1, ORGANIZATION SEQUENTIAL,&
ACCESS MODIFY, ALLOW NONE, RECORDSIZE 50
00020 GET #1
00030 MOVE FROM #1, I, A$=I
00040 A$=A$ + ","
00050 I = I + 1
00060 MOVE TO #1, I, A$=I
00070 UPDATE #1
00080 CLOSE #1
00090 END
```

This program opens an existing file named TEST, reads the first record into the buffer, and associates the data with the variables in the MOVE FROM statement.

The MOVE TO places the record into the buffer, and the PUT statement writes the record back into file #1. The file is closed and the program ends.

CHAPTER 11
PROGRAM SEGMENTATION

11.1 SUBPROGRAMS

In addition to functions, and subroutines, BASIC supplies a third method for writing procedures to be used several times; subprograms. A subprogram allows you to divide a large task into smaller, more manageable units which, in turn, can be accessed individually.

You can use subprograms in two ways:

1. as a segment of a main program which can be called several times from the main program
2. as a mini program, which can be called by several different main programs.

In both cases, the subprogram is executed by a CALL statement (Section 11.1.1) contained in the main program.

The SUB statement marks the beginning of a subprogram and has the following format:

SUB name [[(dummy argument(s))]]

where:

name	is the unique name for the subprogram. (The length of the name is system-specific.)
(dummy argument(s))	represent one or more parameter variables and file references separated by commas. The variables must agree in type and number with that of the calling sequence. (See Section 11.1.2.)

If you use the SUB statement in a multi-statement line, it must be the first statement in that line.

The body of the subprogram may contain any legal BASIC statement except for those which affect transfer out of the body of the subprogram. Transfers into and out of the body of a subprogram are illegal.

All variables in a subprogram are local to that subprogram. These local variables are initialized to 0 or a null string upon each entry to the subprogram. Also, any data used from DATA statements are local to the subprogram. The DATA pointer in the main program is not affected.

Exit from a subprogram and return to the main program with the SUBEND statement or SUBEXIT statement. The SUBEND statement has the following format:

SUBEND

The SUBEXIT statement has the following format:

SUBEXIT

SUBEXIT returns control to the calling program. It has the same effect as GOTO n, where n is the line number of the appropriate SUBEND statement.

SUBEXIT is illegal in a main program or multi-line function definition.

11.1.1 The CALL Statement

The CALL statement transfers control to the subprogram, provides a parameter transfer, and saves the state of the calling program.

The CALL statement has the following format:

CALL name [[(actual argument(s))]]

where:

name	is the subprogram name defined in the SUB statement.
(actual argument(s))	is one or more variables, constants, and expressions, separated by commas. These parameters must agree in position, type, and number with the dummy list in the SUB statement

You place the CALL statement anywhere in a main program, subprogram, or multi-line DEF. When you reference a subprogram with the CALL statement, BASIC replaces the dummy arguments with the corresponding actual arguments (Section 11.1.2) listed with the CALL. The subprogram then works with these parameters.

The following is a SUB statement:

```
500 SUB TEST (A,B$)
```

This is a corresponding CALL:

```
50 CALL TEST (C,A$)
```

Upon returning to the main program, BASIC executes the statement following the CALL statement.

11.1.2 Dummy And Actual Arguments

Because you can reference subprograms at more than one point throughout a program, many of the values used by the subprogram may change each time it is used. Dummy arguments in subprograms represent the actual values passed to the subprogram when it is called.

These dummy arguments indicate the data type of the actual arguments they represent. The position, number, and type of each dummy argument in a subprogram list must agree with the position, number and type of each actual argument in the reference to the subprogram (CALL statement).

Items passed to subprograms can be any legal variable, constant, expression, array, or array element. The value of any parameter can be used as a file number in the subprogram. BASIC passes items from the main program to the subprogram either by value or by reference. When passing by value, BASIC makes a temporary copy of the value in the calling program and uses the copy for calculations in the subprogram. The value in the calling program remains unchanged. The following items are passed by value:

1. constants
2. expressions
3. array elements

When passing by reference or address, BASIC takes the actual value from the location in the main program, uses the value in the subprogram, then replaces the value in the main program. In this case, because of calculations in

the subprogram, the value passed by reference could change in the main program. The following items are passed by reference:

1. variables
2. entire arrays

It is not possible to pass complete arrays by value. Individual elements of a list or table, however, are always passed by value. When an individual entry in an array is passed to a subprogram, it is received as a numeric or string variable depending on its type. For example:

```
50 SUB ELEMENTS (ARRAY)

20 CALL ELEMENTS (BCD(5))
```

The CALL passes the copy of the value in array element BCD(5) to the subprogram. The SUB statement accepts the value in the variable name ARRAY.

If you specify an entire array in either argument list, you do not include the subscript. For example:

C( ) is a list
C(,) is a matrix

When you pass an array to a subprogram, its dimensions remain the same as in the main program. It is illegal to use a DIM statement on an array you specify in the SUB statement. You can, however, redimension such an array with a MAT statement (Chapter 8). The array will also be redimensioned in the main program as well. Arrays local to the subprogram must appear in DIM statements within the subprogram.

Functions can be defined inside subprograms. A function definition is local to the subprogram in which it is defined. However, you can pass the value of a function as an expression.

You can also pass files to a subprogram. BASIC passes the position of the file pointer to the subprogram unchanged from its position after the last operation affecting the file in the main program. Any operation on a file in a subprogram also affects the file in the main program.

You can also open a file within a subprogram. The file remains open after BASIC returns to the main program. When you include a file reference in a SUB statement, the reference must be a variable name.

The following example illustrates argument lists in the SUB and CALL statements:

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

READY
RUNNH
A%= 5 B%= 300 C%= 450
```

Note that on RSTS/E, the main program and any subprograms must be compiled separately. See your User's Guide for details.

The subprogram ARG is called twice in this program with the CALL statements on lines 20 and 30. The first time the subprogram is referenced, the value stored in B% is passed to the integer variable D% to be used in calculations.

The second CALL statement passes the value stored in C% to integer variable D%. Notice that variables are local to the subprogram (A%). Consequently, you can use the same variable name in a main program and a subprogram without interference.

BASIC passes all constants in the CALL statement to the subprogram by value. That is, the value of the constant does not change in the main program. However, BASIC passes all variables by reference. The value stored in the variable location is passed to the subprogram. Consequently, this value may change in the main program after BASIC executes the subprogram.

The following table summarizes the proper form for variable names, functions, arrays, and files references in SUB and CALL statements.

Table 11-1 Arguments

Data Type	Dummy Argument SUB Statement	Actual Argument Call Statement
numeric	A	B
integer	A%	B%
string	A$	B$
entire list	A( ), A%( ), A$( )	B( ), B%( ), B$( )
entire matrix	A(,), A%(,), A$(,)	B(,), B%(,), B$(,)
file	A,C	1, N
array element	A	D(I)

11.2 TRANSFERRING CONTROL TO ANOTHER PROGRAM – THE CHAIN STATEMENT

The CHAIN statement transfers control from the current program to a program stored in a file. CHAIN first closes all files and erases all program lines, arrays and variables. Finally, it loads the program from the file, compiles it, if necessary, and starts the execution of the new program.

The CHAIN statement has the same effect as an END statement followed by an OLD command and then a RUN command.

The format of the CHAIN statement is

CHAIN string [[LINE line number]]

where:

string	is a file specification for the file containing the new program. The string can be any string expression.
[[LINE line number]]	specifies the line to start execution in the new program.

If no line number is specified, then execution starts at the lowest numbered line.

Consider the following example:

The file specified by "SEG1" contains:

```
00005 PRINT "SEG1 IS WORKING"              !PRINTS IDENTIFYING MESSAGE
00010 OPEN "DATA1" FOR OUTPUT AS #1        !OPENS OUTPUT FILE
00020 FOR I = 1 TO 100                     !WRITES OUT ALL THE
00030 PRINT #1, I*2                        !EVEN NUMBERS 2 TO 200
00040 NEXT I                               !TO THE FILE
00050 CLOSE #1                             !CLOSES THE FILE
00060 CHAIN "SEG2.B20"                     !CHAINS TO THE NEXT
00070 END                                  !SEGMENT
```

The file specified by "SEG2" contains:

```
00005 PRINT "SEG2 IS WORKING"              !PRINTS IDENTIFYING MESSAGE
00010 OPEN "DATA1" FOR INPUT AS #1         !OPENS EXISTING FILE
00020 FOR I = 1 TO 100                     !INPUTS THE NUMBERS
00030 INPUT #1, I                          !FROM THE FILES
00040 T = T+I                              !AND ADDS THEM TOGETHER
00050 NEXT I                               !STORING THE TOTAL IN T
00060 PRINT "THE TOTAL IS";T               !PRINTS THE TOTAL
00070 CLOSE #1                             !CLOSES INPUT FILE
00080 END
```

A run of these programs produces the following output.

```
RUNNH
SEG1 IS WORKING
SEG2 IS WORKING
THE TOTAL IS 2550
```

If the specified file does not exist, BASIC prints an error message. To allow the error recovery, BASIC does not erase the current program lines, variables, or arrays. However, all files are closed as they normally would be.

Remember to SAVE a program containing a CHAIN statement before running it, otherwise, the program will erase itself from memory.

11.2.1 Preserving Variables – The COMMON Statement[1]

The COMMON statement preserves data passed from one segment of a program to another. Data values associated with the names of items in the COMMON statement are placed in a common area. The value assigned will be retained when you transfer control from one segment of a program to another with the CALL statement.

COMMON defines an area that is available to any subprogram that defines it. Thus names need not be the same in all routines that access a given COMMON block.

The format of the COMMON statement is:

COM [[MON]] [[(name)]] list

[1]The COMMON statement is not available on the DECSYSTEM-20 in Version 1.

where:

(name) is an optional name for the storage block. The name can be 1 to 6 characters.

list specifies the variables and arrays to be preserved in COMMON. It is in the general form:

var [[(int [[,int]])]], var[[(int[[,int]])]], . . .

The value of each variable in a global common area is position dependent. Therefore, the COMMON statements in the program CALLED must specify the same variable types and array dimensions. The line numbers, variable names, and arrays of each COMMON statement can be different from the original program. But the order of the variables and arrays must be maintained.

Consider these examples:

MAIN

```
10 COMMON A,B,C$
20 COMMON D%(100)
30 COMMON G$(2)
```

Subprogram 1

```
10 COMMON A,B

30 COMMON C$,D%(100),G$(2)
```

Subprogram 2

```
10 COMMON A,B,D%(100)
20 COMMON C$
30 COMMON G$(2)
```

MAIN and Subprogram 1 have equivalent COMMON statements. Subprogram 2 has a different order of variables; D%(100) appears before C$.

It is possible to extend COMMON by placing additional variables and arrays after the existing ones. For example,

MAIN

```
10 COMMON A,B%(100)
20 COMMON G$(5)
```

Subprogram

```
10 COMMON A,B%(100)
20 COMMON C$(5),F9(100)
30 COMMON A$(30)
```

The subprogram has the equivalent COMMON statements and has extended the original COMMON with F9(100) and A$(30).

COMMON lists from all COMMON statements with the same name in a program segment are collected into a global common area with that name. The order in which the COMMON list appears textually determines the order of the list in the global area.

The variables in a COMMON list retain their values until an END statement is executed.

BASIC automatically dimensions arrays that are in COMMON. Therefore, if an array is in COMMON, do not dimension it with a DIM statement. If an array is in both a COMMON and a DIM statement, BASIC prints an error message and stops program execution.

If more than one common block is used in a program, all declarations for one block must appear together before any declarations for another common block. For example:

LEGAL

```
10 COM (BLOCKNAME) A,B
20 COM (BLOCKNAME) C
30 COM X
```

ILLEGAL

```
10 COM (BLOCKNAME) A,B
20 COM X
30 COM (BLOCKNAME) C
```

Variables that you declare in COMMON statements cannot be referenced before the COMMON statements that declare them.

APPENDIX A

SUMMARY OF BASIC-PLUS-2 STATEMENTS, FUNCTIONS, AND OPERATORS

This appendix summarizes the BASIC-PLUS-2 statements, functions, and operators. The descriptions contain statement format, examples, and brief notes on statement usage. Section numbers are listed for quick reference.

STATEMENTS

CALL 11.1.1

CALL name [[(actual arguments)]]

```
200 CALL SUB1 (A,B)
```

The CALL statement transfers control to a specified subprogram, transfers parameters, and saves the state of the calling program. Parameters contained in the argument list must agree in type and number with the corresponding SUB statement.

CHAIN 11.2

CHAIN string [[LINE line number]]

```
15 CHAIN "SEE" LINE 70
```

The CHAIN statement passes control to a specified program. If no line number is specified, execution starts at the beginning of the program.

CHANGE 7.4.4

CHANGE list TO string variable

or

CHANGE { string variable / string expression } TO list

```
25 CHANGE A TO A$
```

The CHANGE statement converts a list of integers (real numbers are truncated) into a string of characters and vice versa. The length of the string is determined by the value found in element 0 of the list.

CLOSE 9.2.2

CLOSE [[#]] expression(s)

```
150 CLOSE #6,8
```

The CLOSE statement terminates I/O to a device and writes all active buffers. The number sign and file expressions are optional. When no files are specified, all files currently open are closed.

COM[[MON]] **11.2.1**

COM [[(name)]] list

50 COM (TEST) A,B,C

The COM, or COMMON statement allows you to establish a named storage area that can be shared by 2 or more subprograms. The variables and arrays in the variable list are assigned to the named area and, when accessed by more than 1 subprogram, must be of the same data type. The common area name must be 1 to 6 characters.

DATA **3.1.3**

DATA constant(s)

50 DATA 4.3, "string", 18, 42%

The DATA statement allows you to provide a pool of information that is accessible to the program by means of a READ statement. A DATA statement must be the only statement on the line and, when you specify more than one item, you must separate them with commas. DATA statements cannot be continued.

DEF (single-line) **7.6.1**

DEF FNa [[(b1,b2,b3, . . .bn)]]=expression

10 DEF FNX (A,B)=A*B

The DEF statement establishes a user-defined function. The function name can be any legal variable name and must begin with FN. The variable type determines the function type. The optional arguments represent dummy parameters and cannot contain array elements. The function definition can refer to any of the dummy parameters or to other program variables but the definition cannot be recursive. Single-line user-defined functions are local to the main program or subroutine in which they are contained.

DEF (multi-line) **7.6.2**

DEF FNa [[(b1,b2,b3, . . .bn)]], [[c1,c2,c3, . . .cn]]

10 DEF FNX (A,B), C

The multi-line DEF establishes user-defined functions and allows you to include other statements in the body of the function. The function name can be any variable name preceded by FN. Any statement can appear in a function except SUB, SUBEND, RETURN or another DEF. The DATA and DIM statements are not local to the function definition. A GOTO, GOSUB, ONGOTO, or ONGOSUB transfer outside the function is not allowed. The function definition must end with an FNEND statement.

DEF* (multi-line) **7.6.3**

DEF* FNa[[(b1,b2,b3, . . .bn)]], [[c1,c2,c3, . . .cn]]

10 DEF* FNZ%

This statement has been added for compatibility with other BASICs. The distinguishing asterisks appear only in the DEF statement and not in program references to the function. The statement permits global references to function parameters and places parameter values in global locations. Transfers are allowed into and out of DEF* functions; however, random transfers can produce unpredictable results. If you transfer out of the function, be sure to transfer in and exit through the FNEND statement.

DELETE 10.3

DELETE file exp [[,INVALID line no.]]

```
60 DELETE #5, INVALID 500
```

The DELETE operation is used on relative and indexed files. The operation erases an existing record from the file. The INVALID clause shifts control to an error-handling routine if the operation fails.

DIM[[ENSION]] 2.6.1

DIM subscripted variable(s)

```
30 DIM B(2,3)
```

The DIM statement reserves storage for arrays. The size of the reserved storage is determined by the subscripts, (constant). A maximum of 2 subscripts is permitted and, when 2 are used, must be separated by a comma.

DIM # 9.3.1

DIM #expression, array(s) [[=integer]]

```
50 DIM #2, A(10,15), B(50)
```

This statement allocates space for the specified arrays on the file associated with the logical number. Storage is allocated at the beginning of the file such that the right most subscript varies the fastest. The default string storage length is 16 and the space is pre-allocated.

END 5.4

END

```
100 END
```

The END statement trerminates program execution and closes all files. It is optional. When used, END must be the last statement in the program.

FIND 10.3

FIND file exp [[,RECORD num exp]]

[[,KEY # num expr { GT / GE / EQ } string exp]]
[[,LOCKED line no.]][[,INVALID line no.]]

```
50 FIND #7, RECORD 25, LOCKED 120
```

The FIND operation causes a RECORD search in the specified file. For sequential files, the FIND starts at the beginning of the file and locates each successor record for each FIND operation. Relative files allow the specification of a record number. Indexed files allow the specification of a key or a sequential search through the key table. The RECORD and KEY specifications are restricted to relative and indexed files.

FNEND 7.6

FNEND

```
40 FNEND
```

The FNEND statement causes an exit from a user-defined function and signals the function's logical and physical end.

FNEXIT **7.6**

FNEXIT

70 FNEXIT

The FNEXIT statement is equivalent to a GOTO n, where n is the line number of the FNEND statement for the current multi-line DEF. FNEXIT is legal only inside a multi-line DEF.

FOR **5.2.1**

FOR variable=num exp1 TO num exp2 [[STEP num exp3]]

25 FOR I=1 TO 5 STEP 2

The FOR statement initiates and controls a loop. A simple numeric variable must be used after the FOR, and the same variable must appear in the required NEXT statement. The first numeric expression is the initial loop value; the second expression is the terminating loop value. The optional STEP expression is the loop increment; +1 is the default. Transfer into an uninitialized loop is illegal.

FOR (conditional) **5.2.3**

FOR variable=num exp1 [[STEP num exp2]] { WHILE / UNTIL } conditional exp

80 FOR I=1 UNTIL I>10

The conditional FOR statement duplicates the previous FOR statement except that loop termination is determined by a false expression in the WHILE clause or a true expression in the UNTIL clause.

GET **10.3**

GET file exp [[,KEY # num exp { GE / GT / EQ } string exp]]
[[,RECORD num exp]]
[[,LOCKED line no.]][[,INVALID line no.]]

50 GET #5

The GET operation reads a record from a specified file into a buffer. On sequential files, GET operations are performed on succeeding records starting at the beginning of the file. Relative files allow the specification of a record number, and indexed files allow the specification of a key name.

GOSUB **5.5.1**

GOSUB line number

25 GOSUB 120

The GOSUB statement transfers control to a subroutine that begins at a specified line number.

GOTO **5.1.1**

GOTO line number

40 GOTO 85

The GOTO statement unconditionally transfers control to a specified line number.

IF **5.1.3**

IF conditional exp { THEN / GOTO } line number

IF conditional exp THEN statement(s)

IF conditional exp { THEN / GOTO } line number ELSE { line number / statement(s) }

IF conditional exp THEN statement(s) ELSE { line number / statements }

```
25 IF A=0 THEN PRINT "A EQUALS 0"
```

The various forms of the IF statement allow branches in the program. The IF statement can also cause execution of statements except the following:

DIM, REM, DATA, END, DEF, FNEND, SUB, SUBEND, WHILE, UNTIL, NEXT, and IMAGE.

IFEND# **9.2.6**

IFEND # expression, { THEN / GOTO } { statement / line number }

```
25 IFEND #6 THEN 50
```

The IFEND # statement checks for the end of file. If the file pointer is at the end of the file, you can transfer control to another line of the program or execute a statement.

IFMORE **9.2.7**

IFMORE # expression, { THEN / GOTO } { statement / line number }

```
10 IFMORE #7 GOTO 100
```

The IFMORE # statement tests whether the file pointer is at the end of file. If NOT at the end, BASIC executes the statement or goes to the line number specified.

IMAGE **4.5**

{ IMAGE unquoted string / :unquoted string }

```
10 : #.## ##.##
```

The IMAGE statement is used in conjunction with the PRINT USING statement. The characters following the colon, or keyword IMAGE, define the format of output. IMAGE must be the only statement on the line and cannot contain comments.

INPUT **3.1.1**

INPUT variable(s)

```
25 INPUT A,B,C%
```

The INPUT statement allows you to type in data to the program from the terminal. The program requests data by printing a question mark on the terminal and then waiting for you to respond.

INPUT # 9.2.3

INPUT # expression, variable(s)

```
25 INPUT #6, A,B,C
```

The INPUT # statement acts very much as the INPUT statement. However, the INPUT # statement requests data from a terminal-format file rather than from you.

INPUT LINE and LINPUT 3.1.2

INPUT LINE string variable(s)
LINPUT string variable(s)

```
15 INPUT LINE A$, B$
```

The INPUT LINE statement allows a character string (ending with a line terminator) to be input to a specified variable. The line terminator is included in the string with INPUT LINE but discarded with LINPUT.

INPUT LINE # and LINPUT # 9.2.3.1

INPUT LINE # expression, string variable(s)
LINPUT # expression, string variable(s)

```
10 INPUT LINE #4, A$, B$
```

The INPUT LINE # and LINPUT # statements read strings from a terminal-format file.

KILL 9.4.2

KILL string expression

```
10 KILL "SALARY"
```

The KILL statement deletes a file from storage.

LET 2.5

LET variable(s)=expression
variable(s)=expression

```
10 A=65
```

The LET statement assigns constants and expressions to variables. The keyword LET is optional.

LINPUT 3.1.2

See INPUT LINE

LINPUT # 9.2.3.1

See INPUT LINE #

MAP 10.1.4

MAP (name) ALIGNED element(s)
UNALIGNED

10 MAP (Buff1) A%, B$, C

The MAP statement associates a named buffer with a file. Specified data in the element list is moved from the file to the buffer on a GET and from the buffer to the file on a PUT.

MARGIN 9.2.9

MARGIN [[#expression,]] num exp

10 MARGIN 5

The MARGIN statement allows you to modify the margin of your terminal or terminal format file.

MAT INPUT 8.4.1

MAT INPUT array(s)

50 MAT INPUT A

The MAT INPUT statement allows element values to be entered in an array. Input is read from the terminal. Elements are stored in row order as they are typed.

MATRIX OPERATIONS 8.3

MAT PRINT 8.4.2

MAT PRINT array(s)

120 MAT PRINT A;

The MAT PRINT statement outputs each element of a specified array.

MAT READ 8.4.3

MAT READ array(s)

50 MAT READ B,C

The MAT READ statement reads the values into elements of a 1- or 2-dimensional array from a DATA statement.

MOVE 10.3.4

MOVE [[{ALIGNED | UNALIGNED}]] {FROM | TO} file exp, I/O list

15 MOVE TO #5, A$, B, C( ), FILL%

The MOVE statement associates the data in a record with the variables you specify in the I/O list.

NAMEAS 9.4.1

NAME string1 AS string2

15 NAME "MONEY" AS "ACCNTS"

The NAME-AS statement renames a file without changing the contents of the file or the file number associated with it.

NEXT **5.2.1**

NEXT variable(s)

15 NEXT I

The NEXT statement terminates a FOR, WHILE, or UNTIL loop. The variable must correspond with the variable in the initial FOR statement. Nested loops cannot cross each other.

NODATA **9.2.8**

NODATA [[#expression,]] line number

25 NODATA #4, 85

BASIC checks for the end of file (or data in a DATA statement) and goes to line number specified if no data is left.

ONERROR **5.6.1**

ONERROR GOTO line number

25 ONERROR GOTO 50

The ONERROR statement allows control to shift to an error-handling routine.

ON-GOSUB **5.5.2**

ON num exp GOSUB line number(s)

50 ON A+B GOSUB 80, 95, 100

The ON GOSUB statement is used to conditionally transfer control to one of several subroutines or to one of several entry points into one or more subroutines.

ON-GOTO **5.1.2**

ON num exp GOTO line number(s)
THEN

20 ON J% GOTO 85

The ON-GOTO statement allows you to transfer control to another line of the program.

ON-THEN **5.1.2**

See ON-GOTO

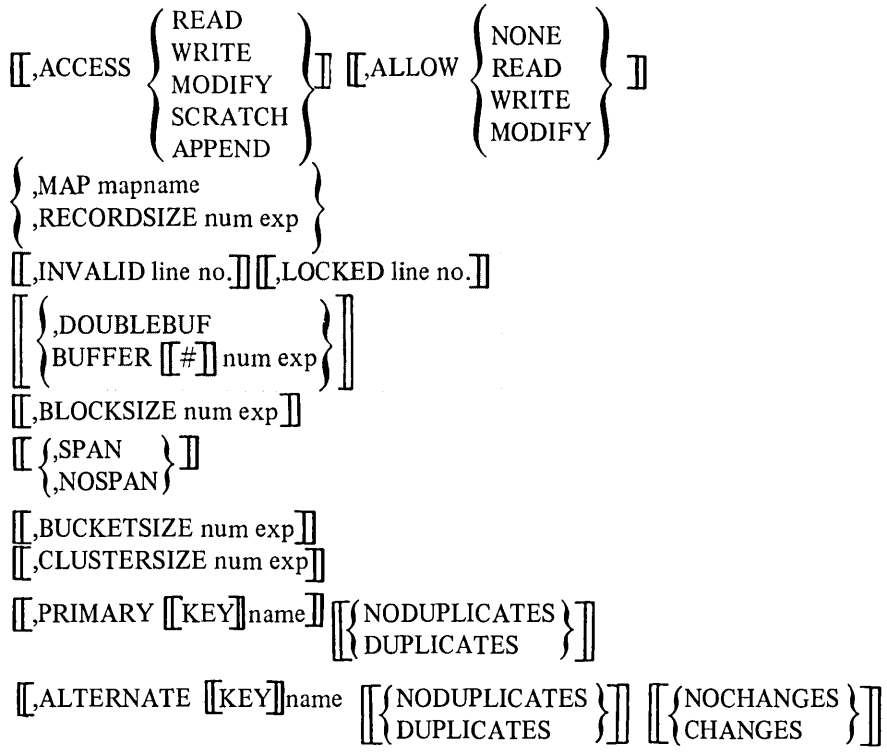
OPEN

9.2.1 Terminal-Format
9.3.2 Virtual Array Files
10.2.1 Record Files

OPEN filename exp [[FOR { INPUT / OUTPUT }]]

AS [[FILE]] [[#]] expression

,[[ORGANIZATION]] { SEQUENTIAL / RELATIVE / INDEXED / VIRTUAL } [[{ FIXED / VARIABLE / STREAM }]]



10 OPEN "FILE" FOR INPUT AS FILE 4

The OPEN statement enables you to create a new file or access an existing file. You can use the OPEN statement to access terminal-format files as well as record files.

PAGE **9.2.10**

PAGE [[#expression,]] num exp

20 PAGE #2, 60

The PAGE statement allows you to set a PAGE size of any positive number of lines.

PRINT **3.2**

PRINT [[expression(s)]]

30 PRINT A+B

The PRINT statement causes the data you specify to be output on the terminal. The expression list can be expressions, variables, or quoted strings separated by a comma or a semicolon. Commas cause output to terminal print zones; semicolons ignore the print zones.

PRINT # **9.2.4**

PRINT # expression, list

65 PRINT # 6, A, B+C

The PRINT # statement writes data into the specified terminal-format file.

PRINT USING 4.1

PRINT USING string, list

```
10 PRINT USING "**##.##", A,B,C
```

The PRINT USING statement causes output to be printed in a specified format. The list indicates the elements to be printed and the image can be a line reference to an IMAGE statement.

PUT 10.3

PUT file exp [[,RECORD num exp]]
[[{,MAP line no. / ,COUNT num exp}]]
[[,LOCKED line no.]]
[[,INVALID line no.]]

```
25 PUT #7, RECORD 15
```

The PUT statement writes a record from a buffer to a specified file. The RECORD clause is used for relative files; the KEY clause is used for indexed files. Sequential files allow PUT operations only at the end of the file. The MAP and COUNT clauses define the size of the record.

RANDOMIZE 7.2.3

RANDOM[[IZE]]

```
10 RANDOM
```

The RANDOMIZE statement changes the starting point of the RND function to a new unpredictable location.

READ 3.1.3

READ variable(s)

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

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

REM 1.4

REM comment

```
30 REM this is a comment
```

The REM statement contains user written comments and has no effect on program execution.

RESTORE [[#]] (or RESET) 3.1.3 9.2.5 10.2.3

RESTORE # expression

```
30 RESTORE #3
```

The RESTORE # statement resets the specified terminal-format file or record file to its beginning from the current position of the file. Restore without a file exp restores the data in a DATA statement.

RESUME 5.6.1

RESUME [[line number]]

50 RESUME 35

The RESUME statement is the last statement in an error-handling subroutine. If no line number is specified, control is shifted back to the point of error generation. If a line number is specified, control is shifted to that line.

RETURN 5.5.1

RETURN

60 RETURN

The RETURN statement is the last statement in a subroutine. It shifts control to the statement following the last executed GOSUB statement.

SCRATCH 10.2.2.1

SCRATCH #file exp

25 SCRATCH #6

The SCRATCH statement allows you to truncate the file. SCRATCH can only be used if the file was OPENed with ACCESS SCRATCH.

SLEEP 5.3.1

SLEEP num exp

10 SLEEP A*B

The SLEEP statement causes a temporary halt in execution. The length of delay is determined by the value of the expression in seconds.

STOP 5.4

STOP

110 STOP

The STOP statement causes a halt in program execution. Files are not closed and a message indicating the location of the halt is printed.

SUB 11.1

SUB name [[(dummy argument(s))]]

40 SUB TEST (A,B%)

The SUB statement marks the beginning of a subprogram and defines the type and number of subprogram parameters.

SUBEND 11.1

SUBEND

25 SUBEND

The SUBEND statement marks the end of the subprogram and returns control to the calling program. It must appear at the end of all subprograms.

SUBEXIT 11.1

SUBEXIT

45 SUBEXIT

The SUBEXIT statement returns control to the calling program. It has the same effect as GOTO n, where n is the line number of the appropriate SUBEND statement. SUBEXIT is legal only inside a subprogram.

UNTIL 5.2.4

UNTIL conditional exp

50 UNTIL I=0

The UNTIL statement sets up a loop that must have a corresponding NEXT statement. The loop executes until the expression is true.

UPDATE 10.3

UPDATE file exp [[{,MAP line no.}]] [[{,COUNT exp}]] [[,INVALID line no.]]

The UPDATE statement changes an existing record in the file. The new record size as defined in the MAP or COUNT clause, must be the same as the record it replaces. On sequential files, an UPDATE must be preceded by a successful GET or FIND.

WAIT 5.3.2

WAIT num exp

40 WAIT 15

The WAIT statement specifies the maximum number of seconds allowed for input before an error is generated. A zero or null value disables the WAIT.

WHILE 5.2.4

WHILE conditional exp

75 WHILE A%<10%

The WHILE statement also sets up a loop that must have a NEXT statement. The WHILE expression is evaluated before each loop iteration. If the expression is true, BASIC executes the statements in the loop. If the expression is false, BASIC executes the statements following the NEXT statement.

FUNCTIONS

Function	Usage
ABS(x)	returns absolute value of x.
ASCII(x$)	returns the decimal ASCII value of the first character of a specified string.
ATN(x)	returns the arctangent of x in radians.
CHR$(x%)	returns the character equivalent of the ASCII value x%.
CLK$	returns an 8-character string representing the time of day (hh:mm:ss)
COS(x)	returns the cosine of x.
DAT$	returns an 8-character string representing the current date (dd-mm-myy).
DATE$(0%)	returns the current date in the form mm/dd/yy.
DATE$(x%)	returns the date according to the formula; x = day of the year + (years since 1970*1000) in the form dd-mmm-yy.
EDIT$(string,n%)	converts a string according to integer specified in table.
EXP(x)	returns value of e^x where 3=2.71828.
FIX(x)	returns truncated value of x.
INSTR(z%,x$,v$)	returns the position of substring y$ in main string x$ starting at position z%.
INT(x)	returns the integral part of x.
LEFT$(x$,y%)	returns a substring of x$ beginning at the leftmost position for a total length of y characters. (Also LEFT(x$,y%)).
LEN(x$)	returns the number of characters in x$.
LOG(x)	returns the natural logarithm of x.
LOG10(x)	returns the common logarithm of x.
MID(string,n1%,n2%)	returns a substring of string starting at position n1% with n2% characters.
MOD(x,y)	returns the real result of x mod y, which is equal to x-y*INT(x/y).
MOD%(x,y)	returns the integer result of x mod y, which is the remainder of x/y.
PI	constant value, 3.14159
POS(x$,y$,z%)	returns the position of substring y$ in main string x$ beginning of position z. (See also INSTR.)
RAD$(x%)	converts the integer x% to its RADIX-50 equivalent.
RIGHT$(x$,y%)	returns a substring of x$ that ranges from the yth character to the end of the string. (Also, RIGHT(x$,y%)).
RND	returns a random number between 0 and 1.

Function	Usage
SEG$(x$,v%,z%)	returns a substring of x$ that ranges from the yth character to the xth character.
SGN(x)	returns 1 if x is positive; 0 if x is zero; -1 if x is negative.
SIN(x)	returns the sine of x in radians.
SPACE$(x)	produces a string of x spaces.
SQR(x)	returns the square root of x; also SQRT(x).
STR$(x)	returns the value of an expression without the leading and trailing blanks. (Also NUM$(x)).
STRING$(x%,y%)	creates a string of x length whose characters represent the ASCII value of y.
TAB(x%)	moves the print head to the xth position.
TAN(x)	returns the tangent of x in radians.
TIME$(x%)	returns time as x minutes before midnight.
TIME$(0%)	returns current time.
TIME(0)	returns clock time in seconds since midnight.
TIME(1%)	returns used CPU time in tenths of seconds.
TIME(2%)	returns connect time in minutes.
VAL(x$)	computes the numeric value of the numeric string x$; x$ must be acceptable numeric input.

Table A-1 Arithmetic Operators

Operator	Use	Meaning
^ or **	5^2 or 5**2	exponentiation
*	A*B	multiplication
/	A/B	division
+	A+B	addition
-	A-B	subtraction, unary minus

Table A-2 Logical Operators

Operator	Use	Meaning
NOT	NOT A	logical negative of A
AND	A AND B	logical product of A and B
OR	A OR B	logical sum of A and B
XOR	A XOR B	logical exclusive OR of A and B
EQV	A EQV B	A is logically equivalent to B
IMP	A IMP B	logical implication of A and B

Table A-3 Relational Operators

Operator	Use	Meaning
=	A=B	A is equal to B
<	A<B	A is less than B
>	A>B	A is greater than B
<= or =<	A<=B	A is less than or equal to B
>= or =>	A>=B	A is greater than or equal to B
# or <> or ><	A<>B	A is not equal to B
==	A==B	A is approximately equal to B
+,&	A$+B$	string concatenation
Note that A is approximately equal to B (A==B) if the difference between A and B is less than 10^(-6). If A$ and B$ are strings, the relation (==) is true if the contents of A$ and B$ are the same in length and composition.		

APPENDIX B
ASCII CODE

Table B-1 ASCII Table

ASCII Decimal Number	Character	Meaning	ASCII Decimal Number	Character	Meaning
0	NUL	Null	40	(	Left parenthesis
1	SOH	Start of heading	41	)	Right parenthesis
2	STX	Start of text	42	*	Asterisk
3	ETX	End of text	43	+	Plus sign
4	EOT	End of transmission	44	,	Comma
5	ENQ	Enquiry	45	-	Minus sign or hyphen
6	ACK	Acknowledgement	46	.	Period or decimal point
7	BEL	Bell	47	/	Slash
8	BS	Backspace	48	0	Zero
9	HT	Horizontal tab	49	1	One
10	LF	Line feed	50	2	Two
11	VT	Vertical tab	51	3	Three
12	FF	Form feed	52	4	Four
13	CR	Carriage return	53	5	Five
14	SO	Shift out	54	6	Six
15	SI	Shift in	55	7	Seven
16	DLE	Data link escape	56	8	Eight
17	DC1	Device control 1	57	9	Nine
18	DC2	Device control 2	58	:	Colon
19	DC3	Device control 3	59	;	Semicolon
20	DC4	Device control 4	60	<	Left angle bracket
21	NAK	Negative acknowledgement	61	=	Equal sign
22	SYN	Synchronous idle	62	>	Right angle bracket
23	ETB	End of transmission block	63	?	Question mark
24	CAN	Cancel	64	@	At sign
25	EM	End of medium	65	A	Upper case A
26	SUB	Substitute	66	B	Upper case B
27	ESC	Escape	67	C	Upper case C
28	FS	File separator	68	D	Upper case D
29	GS	Group separator	69	E	Upper case E
30	RS	Record separator	70	F	Upper case F
31	US	Unit separator	71	G	Upper case G
32	SP	Space or blank	72	H	Upper case H
33	!	Exclamation mark	73	I	Upper case I
34	"	Quotation mark	74	J	Upper case J
35	#	Number sign	75	K	Upper case K
36	$	Dollar sign	76	L	Upper case L
37	%	Percent sign	77	M	Upper case M
38	&	Ampersand	78	N	Upper case N
39	'	Apostrophe	79	O	Upper case O

Table B-1 (Cont.) ASCII Table

ASCII Decimal Number	Character	Meaning	ASCII Decimal Number	Character	Meaning
80	P	Upper case P	104	h	Lower case h
81	Q	Upper case Q	105	i	Lower case i
82	R	Upper case R	106	j	Lower case j
83	S	Upper case S	107	k	Lower case k
84	T	Upper case T	108	l	Lower case l
85	U	Upper case U	109	m	Lower case m
86	V	Upper case V	110	n	Lower case n
87	W	Upper case W	111	o	Lower case o
88	X	Upper case X	112	p	Lower case p
89	Y	Upper case Y	113	q	Lower case q
90	Z	Upper case Z	114	r	Lower case r
91	[	Left square bracket	115	s	Lower case s
92	\	Back slash	116	t	Lower case t
93	]	Right square bracket	117	u	Lower case u
94	^ or	Circumflex or up arrow	118	v	Lower case v
95	_ or	Back arrow or underscore	119	w	Lower case w
96	`	Grave accent	120	x	Lower case x
97	a	Lower case a	121	y	Lower case y
98	b	Lower case b	122	z	Lower case z
99	c	Lower case c	123	{	Left brace
100	d	Lower case d	124	|	Vertical line
101	e	Lower case e	125	}	Right brace
102	f	Lower case f	126	~	Tilde
103	g	Lower case g	127	DEL	Delete

APPENDIX C
RESERVED WORDS

Certain words in the BASIC-PLUS-2 language are reserved and, therefore, are not legal variable names. However, variations, i.e., IF$, AND%, DIM$, MAT%, are legal and proper variable names. Note that a space must follow all keywords. You may not recognize all the words in this list, since they are for both RSTS/E and the DECSYSTEM-20.

ABORT
ABS
ACCESS
ACCESS%
ALIGNED
ALL
ALLOW
ALTERNATE
AND
APPEND
AS
ASCII
ATN
ATN2
BACK
BEL
BIN
BIN$
BINARY
BIT
BLOCKSIZE
BROADCAST
BS
BUCKETSIZE
BUFFER
BUFFERSIZE
BUFSIZ
BY
CALL
CCPOS
CHAIN
CHANGE
CHANGES
CHR
CLK
CLK$
CLOSE
CLUSTERSIZE
COM
COMMON
COMP%
CON
CONTIGUOUS
COS
COT
COUNT
CR
CTRLC
CVT$
CVT%
DAT
DAT$
DATA
DATE
DEF
DEF*
DEL
DELETE
DELIMIT
DENSITY
DET
DIF$
DIM
DIMENSION
DOUBLEBUF
DUPLICATES
ECHO
ELSE
END
EQ
EQV
ERL
ERN$
ERR
ERROR
ESC
EXP
EXTEND
FF
FIELD
FILE
FILESIZE
FILL
FILL$
FILL%
FIND
FIX
FIXED
FNAME$
FNEND
FNEXIT
FOR
FORCEIN
FROM
GE
GET
GO
GOSUB
GOTO
GT
HANGUP
HT
IDN
IF
IFEND
IFMORE
IMAGE
IMP
INDEXED
INIMAGE
INPUT
INSTR
INT
INV
INVALID
KEY
KILL
LEFT
LEFT$
LEN
LET
LF
LINE
LINO
LINPUT
LOCK
LOCKED
LOG
LOG10
LSA
LSET
MAGTAPE
MAP
MAR
MAR%
MARGIN
MAT
MID
MID$
MOD
MOD%
MODE
MODIFY
MOVE
NAME
NEXT
NOCHANGES
NODATA
NODUPLICATES
NOECHO
NONE
NOPAGE
NOQUOTE
NOREWIND
NOSPAN
NOTAPE
NOT
NUL
NUL$
NUM
NUM$
NUM1$
NUM2
OCT

OCT$
ON
ONECHR
ONENDFILE
ONERROR
OPEN
OR
ORGANIZATION
OUTPUT
PAGE
PI
PLACES
POS
POS%
PPS
PRIMARY
PRINT
PROD$
PUT
QUOTE
RAD$
RANDOM
RANDOMIZE
RCTRLC
RCTRLO
READ
RECORD
RECORDSIZE
RECOUNT
REF$
RELATIVE
REM
RESET
RESTORE
RESUME
RETURN
RIGHT
RIGHT$
RND
SCRATCH
SEG$
SEQUENTIAL
SGN
SI
SIN
SLEEP
SO
SP
SPACE$
SPAN
SQR
SQRT
STATUS
STEP
STOP
STR$
STREAM
STRING$
SUB
SUBEND
SUBEXIT
SUM$
SWAP%
SYS
TAB
TAN
TAPE
TERMINAL
THEN
TIM
TIME
TIME$
TO
TRM$
TRN
TYPE$
UNALIGNED
UNLESS
UNLOCK
UNTIL
UPDATE
USAGE$
USING
USR$
VAL
VARIABLE
VIRTUAL
VPS%
VT
WAIT
WHILE
WITH
WRITE
XLATE
XOR
ZER

APPENDIX D
GLOSSARY

Alphanumeric	The letters of the alphabet (A through Z) and the numerals (0 through 9).
Array	An arrangement of elements in one or more dimensions, i.e., an ordered arrangement of subscripted variables.
ASCII Code	American Standard Code for Information Interchange. A 7-bit code in which textual information is recorded. Characters in the code include upper- and lower-case letters, numbers, common punctuation marks, and special control characters.
BASIC	Beginner's All-purpose Symbolic Instruction Code. A computer programming language that is used for direct communication between terminal units and computers. The language was developed at Dartmouth College.
Binary	1. Pertaining to the number system with a radix of two. 2. Pertaining to information in the form of a bit stream.
Block	A set of records, words, characters, or digits handled as a unit.
Buffer	A device or area used to temporarily hold information being transmitted between two processes, such as external and internal storage devices, or I/O devices and internal high-speed storage.
Buffer Pointer	A position indicator that is located between two characters in an editing buffer, before the first character in the buffer, or after the last character in the buffer.
Call	To transfer control to a specified subroutine.
Caller	The program or routine which calls another program or routine.
Calling Sequence	A specified arrangement of instructions, pointers, and data necessary to pass parameters and control to, and return from, a given subroutine.
Character	One symbol of a set of elementary symbols such as those corresponding to the keys on a typewriter. The symbols usually include the decimal digits 0 through 9, letters A through Z, punctuation marks, operation symbols, and any other special symbols that a computer may read, store, or write.
COMMON area	A section in a program's address space that is set aside for shared use by many modules.
Concatenation	The joining of two character strings to produce a longer string.
Constant	A quantity that does not vary in value.
Data	A general term used to denote any or all information (facts, numbers, letters, and symbols that refer to or describe an object, idea, condition, or situation.

Disk	A form of mass storage device in which information is stored on rotating magnetic platters.
E Notation	A system for representing numbers in exponential format, see exponentiation.
Exponentiation	A mathematical operation whereby a number is increased by a specified factor.
Expression	Any legal combination of data and operators.
File	An ordered collection of characters containing computer instructions and/or data.
Function	An instruction that defines a computer operation.
I/O	Abbreviation for input, output, or both.
Input	Information read by a computer.
Input process	Transmitting data from a peripheral to internal storage.
Integer	A whole number containing no fraction or decimal point.
Jump	A departure from the normal sequence of executing instructions, i.e., a transfer of control to another section of the program.
Loop	A sequence of instructions that is executed repeatedly until a terminal condition is satisfied.
Magnetic Tape	A tape with a magnetic surface on which data can be stored by polarizing selective portions of a surface.
Main Program	The main program exercises primary control over the operations performed and calls subroutines to perform specific functions.
Nesting	To include a loop, routine, or block of data within a larger loop, routine, or block of data.
Offset	The number of locations, or bytes, relative to the base of an array, string, or block. For example, the number of locations relative to zero.
Operand	The data that is accessed when an operation is executed.
Operating System	The collection of programs that administer the operation of the computing system by scheduling and controlling the operation of user and system programs.
Output	1. Data that has been transferred from memory to a medium readable by a person. 2. Pertaining to a device, process, or channel involved in the output process.
Parameter	A variable that is given a constant value for a specific purpose or process.
Pointer	A location containing an address rather than data.
Program	The plan for the solution of a problem. (Sequence of machine instructions necessary to solve a problem.)

Random Access	A process having the characteristic such that the access time is effectively independent of the location of data.
Record	A collection of adjacent related items of data treated as a unit. i-25 Recursive A repetitive process in which the result of each process is dependent upon the result of the previous one.
Routine	A set of instructions and data for performing one or more specific functions.
Run	To transfer a save file from a device into memory and begin program execution.
Statement	An expression or instruction written in a source language.
String	A set of contiguous items of similar type, a connected sequence of characters.
Subscript	A notation, enclosed in parentheses, written to the right of an array name, that represents a specific item in that array.
Subscripted Variable	A variable name followed by one or more subscripts in parentheses.
Syntax	The rules governing statement structure in a computer language: the structure of a language.
Truncate	Dropping part of a number without rounding off.
Variable	A symbol whose value can change during program execution.
Vector	A horizontal or vertical list. Also, an array with only one dimension.
Word	An ordered set of bits that occupies one storage location and is treated by the computer as a unit.

INDEX

INDEX (Cont.)

INDEX (Cont.)

INDEX (Cont.)

INDEX (Cont.)

INDEX (Cont.)

INDEX (Cont.)

INDEX (Cont.)

INDEX (Cont.)

DECSYSTEM-20
BASIC-PLUS-2
Language Manual
AA-0153A-TK

READER'S COMMENTS

NOTE: This form is for document comments only. DIGITAL will use comments submitted on this form at the company's discretion. Problems with software should be reported on a Software Performance Report (SPR) form. If you require a written reply and are eligible to receive one under SPR service, submit your comments on an 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 _____

Please cut along this line.

---------- Fold Here ----------

---------- Do Not Tear - Fold Here and Staple ----------

FIRST CLASS
PERMIT NO. 152
MARLBORO, MASS

BUSINESS REPLY MAIL
NO POSTAGE STAMP NECESSARY IF MAILED IN THE UNITED STATES

Postage will be paid by:

digital

Software Documentation
200 Forest Street MR1-2/E37
Marlboro, Massachusetts 01752