BASIC-PLUS
Language Manual

Order No. DEC-11-ORBPB-B-D

digital equipment corporation · maynard. massachusetts

First Printing, July 1975
Revised: December 1976

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.)

PREFACE

This manual describes the BASIC-PLUS programming language as implemented for the RSTS/E operating system. Information is organized for the benefit of the beginning programmer, allowing the reader to gradually acquire increased programming capabilities.

The BASIC-PLUS language is an extension of BASIC[1] as originally developed at Dartmouth College. However, BASIC-PLUS offers many features not found in standard Dartmouth BASIC-PLUS or many other versions of BASIC-PLUS.

Part I (Chapters 1, 2, 3) describes the RSTS/E system, its hardware and user features, and the simplest level of the BASIC-PLUS language. BASIC-PLUS as described here is essentially (but not exclusively) Dartmouth BASIC as originally developed.

Part II (Chapters 4 through 8) describes some of the advanced features of BASIC-PLUS. The reader will find references to some of these additional capabilities in Part I.

Part III (Chapters 9 through 12) describes the complete range of BASIC-PLUS data handling, including data storage, file I/O, the virtual storage concept and information on particular I/O devices. The appendices review the BASIC-PLUS language in summary form, and provide a list of RSTS/E error messages that BASIC-PLUS users are apt to encounter.

As a language, BASIC-PLUS is easy to learn. Although BASIC-PLUS provides many advanced features for the more experienced programmer, it does not penalize the beginning user. Most problems can be solved on at least a rudimentary level using the statements described in Part I. The BASIC-PLUS statements and features described in Parts II and III allow more efficient code that uses less machine time and memory space, as well as more sophisticated data handling and output.

The content of this manual pertains only to the writing and execution of correct programs in the BASIC-PLUS language. A description of the various RSTS/E commands (NEW, OLD, LIST, RUN, SAVE etc.) can be found in the *RSTS/E System User's Guide.*

There are two modes of BASIC-PLUS operation: NO EXTEND mode and EXTEND mode. In EXTEND mode, certain features are available to the user that are not available in NO EXTEND mode; also, the syntax requirements are stricter in EXTEND mode. In general, descriptions of features should be assumed to apply to both EXTEND and NO EXTEND unless stated otherwise. All BASIC-PLUS coding examples are written in NO EXTEND mode, except those specifically designed to illustrate EXTEND mode features.

For information on all of the current manuals pertaining to RSTS/E operation, consult the *RSTS/E Documentation Directory.*

[1] BASIC is a registered trademark of the Trustees of Dartmouth College. BASIC stands for Beginner's All-purpose Symbolic Instruction Code. In this manual, the term BASIC-PLUS is used to refer to the BASIC language as specifically implemented for RSTS/E; when the term "BASIC" is used, it refers to the language in the generic sense.

PART I

This first part introduces the RSTS/E system and the BASIC-PLUS programming language. Chapter 1 orients the user to the overall RSTS/E environment and to the notation conventions used in the manual. Chapter 2 analyzes an example BASIC-PLUS program to show the fundamental rules of program syntax. Chapter 3 describes the simple BASIC-PLUS program statements. After reading Part I, the user should have a sound introduction to the language, and be able to solve most programming problems. The extent to which Parts II and III will be studied depends largely on individual need and inclination.

CHAPTER 1
AN INTRODUCTION TO RSTS/E BASIC-PLUS

1.1 INTRODUCTION TO PROGRAMMING

For the benefit of the new programmer there are four phases in programming a computer:

1. Defining the problem
2. Designing, writing, and documenting the program
3. Entering, testing, and debugging
4. Using and maintaining the finished program

BASIC-PLUS is one language in which the user can write programs designed for the RSTS/E system. The completed program is generally introduced from a terminal keyboard connected to the RSTS/E system. A program can be input through various peripheral media, such as paper tape, magnetic tape, DECtape, or punched cards; however, the initial creation of a BASIC-PLUS program is usually performed on-line to the computer from the terminal keyboard.

Ideally a program runs correctly as written. In practice this is seldom the case. A program can contain simple typing mistakes or complex logical errors. Typing and syntactical errors are detected as the program is typed at the keyboard and appropriate error messages are printed. Other errors cannot be detected until the user attempts to run the program. Program errors are corrected on-line from the terminal keyboard.

The testing and debugging process is continued until the program appears to produce the correct results. This is a good time to explain to the new user that a computer program only does what the programmer has instructed (not what he meant to instruct). Thus the actions performed by the computer do not necessarily produce the correct results. In order to obtain correct results from a computer, the user must write a program free of format errors that performs the desired computations correctly.

RSTS/E provides keyboard commands which enable the user to create and execute a program, and then to save the program within the system for later retrieval and execution or modification. This saving process is known as storing or filing the program, and a BASIC-PLUS program saved in this fashion is called a program file.

1.2 INTRODUCTION TO TIME-SHARING

RSTS/E is a time-sharing system. This means that when a user is working with RSTS/E, he is probably sharing the system with other users, but the other users are not visible to him so far as the operation of the terminal is concerned.

Many users can be on-line to RSTS/E at one time because RSTS/E controls the scheduling of program execution. User programs are brought into memory from disk, allowed to execute for a short time, and returned to disk. This process is called swapping. RSTS/E records the point in the program at which execution interruption occurs, and is able to resume operation at that point.

Each user is allotted a block of memory between 2K[1] and 16K for storage of a particular program. This block is swapped between memory and disk. If only one user job is active in the system at a given time, that job is allowed to execute without interruption until another program is ready.

[1] The term "K" refers to 1024 (decimal) words of storage in a computer. Hence, 2K = 2048 words and 8K = 8192 words.

1.3 THE BASIC-PLUS PROGRAMMING LANGUAGE

BASIC-PLUS is one of the simplest of all programming languages because of its small number of powerful but easily understood statements and commands and its easy application to problem solving. The wide use of BASIC in scientific, business, and educational installations attests to its value and straightforward application. (See the bibliography at the end of this manual for a list of BASIC texts and other elementary computing texts.)

BASIC is similar to other programming languages in some respects but is especially suited for time-sharing because of its conversational nature. A conversational language is one that allows the user to communicate with the language processor by typing on the terminal keyboard. BASIC responds by printing on the terminal. This provides an interactive man/machine relationship.

BASIC-PLUS contains both elementary statements used to write simple programs and advanced programming features and statements to produce more complex and efficient programs. The key word here is efficient. As the user progresses and gains programming experience, he will naturally find himself becoming more efficient and able to use more sophisticated data manipulations. Almost any problem can be solved with the simple BASIC-PLUS statements. Later in the user's programming experience, the advanced techniques can be added.

1.4 CONVENTIONS USED IN THIS MANUAL

Certain documentation conventions are used throughout this manual to clarify examples of BASIC-PLUS syntax.[1] Each BASIC-PLUS statement is described at least once in general terms using the following conventions:

1. Items in lower case type (formula, variable, etc.) are supplied by the user according to rules explained in the text. Items in capital letters (LET, IF, THEN, etc.) must appear exactly as shown because they form the vocabulary of the BASIC-PLUS language; i.e., they are BASIC-PLUS keywords.
2. Angle brackets < > indicate essential elements of the statement of command being described. For example:

$$\left\{\text{LET}\right\} \langle\text{variable}\rangle = \langle\text{expression}\rangle$$

3. Square brackets [ ] indicate a required choice of one element among two or more possibilities. For example:

$$\text{IF } \langle\text{expression}\rangle \begin{bmatrix}\text{THEN } \langle\text{statement}\rangle \\ \text{THEN } \langle\text{line number}\rangle \\ \text{GOTO } \langle\text{line number}\rangle\end{bmatrix}$$

4. Braces { } indicate an optional statement element or a choice of one element among two or more optional elements:

$$\text{IF } \langle\text{expression}\rangle \begin{bmatrix}\text{THEN } \langle\text{statement}\rangle \\ \text{THEN } \langle\text{line number}\rangle \\ \text{GOTO } \langle\text{line number}\rangle\end{bmatrix} \begin{Bmatrix}\text{ELSE } \langle\text{statement}\rangle \\ \text{ELSE } \langle\text{line number}\rangle\end{Bmatrix}$$

The use of some terms in this document may be unfamiliar to the new user. The following definitions and explanations are valid throughout this manual:

1. BASIC-PLUS prints on the terminal whereas the user types on the keyboard.
2. A statement is a single BASIC-PLUS language instruction identified by one or more BASIC-PLUS language keywords characteristic of the syntax for that instruction.

[1]The syntax of a language is the set of rules governing the combination and ordering of language elements.

3. A BASIC-PLUS program line consists of a line number, followed by one or more BASIC-PLUS language statements entered over one or more terminal lines. A program line is terminated by a RETURN key. A program line may contain one or several statements separated by backslashes (see Section 2.4.1). For compatibility with past versions, a colon is allowed in place of the backslash.

   A BASIC-PLUS terminal line consists of a set of BASIC-PLUS statements, or portions thereof, entered on one physical line, that either by itself or in combination with other terminal lines constitutes a BASIC-PLUS program line.

   The first terminal line of a program line must begin with a line number. Each continued terminal line must be terminated by a LINE FEED key or, if EXTEND mode is current, by an ampersand character followed by a RETURN key. The last terminal line of a program line must be terminated by a RETURN key with no preceding ampersand character.
4. Commands cause BASIC-PLUS or a system program to perform some operation immediately and are not preceded by a line number. A command is terminated by typing the RETURN key.
5. A user program consists of a series of statements written in the BASIC-PLUS language.
6. There is no standard RSTS/E terminal. RSTS/E can accommodate a wide variety of terminals such as a DECwriter II (LA36), VT05 display, VT50 or VT52 DECscope. All terminals have a keyboard and either a printer or cathode ray tube (CRT) screen. The RSTS/E user terminal is referred to as terminal, display, teleprinter, or keyboard, depending upon whether a part or the whole device is indicated. The use of terminals and other peripheral devices is described in the RSTS/E System User's Guide.
7. The term BASIC-PLUS indicates the BASIC-PLUS language, the BASIC-PLUS Interpreter (the system program which accepts and executes BASIC-PLUS programs) or both, depending on the context of usage.

CHAPTER 2
FUNDAMENTALS OF PROGRAMMING IN BASIC-PLUS

2.1 MODE SELECTION (EXTEND VS. NO EXTEND)

There are two program modes: EXTEND and NO EXTEND. All programs written for versions of BASIC-PLUS prior to RSTS/E V06B operate under NO EXTEND. The user can switch modes at any time on either the command or the program level. Immediately following log-in, the system is in NO EXTEND mode, so that programs written in NO EXTEND mode can be executed without any special action by the operator.

When operating in EXTEND mode, the user has access to several features that are not available in NO EXTEND. However, format requirements are somewhat more stringent; in NO EXTEND, spaces and tabs are ignored by the compiler except in string constants. In EXTEND mode, blanks and tabs are syntactically significant. EXTEND mode features will be described throughout the manual.

All examples are shown in NO EXTEND mode except those specifically designed to show EXTEND mode features.

To enter EXTEND mode at the command level, the user types:

```
EXTEND
```

To switch from EXTEND to NO EXTEND mode, the user types:

```
NO EXTEND

    or

NOEXTEND
```

There is no effect if the command entered names the current mode.

Mode switching is possible at the program statement level. To do this, precede the EXTEND or NO EXTEND command with a line number. For example:

```
10 EXTEND

    or

10 NOEXTEND
```

In general, it is not good programming practice to switch modes repeatedly within a program. Recommended usage is to include an EXTEND or NO EXTEND statement at the beginning of a program to ensure that it is compiled under the proper mode.

2.2 EXAMPLE BASIC-PLUS PROGRAM

The program in Figure 2-1 is an example of a user program written in the BASIC-PLUS language. It illustrates syntax and elements of the language as well as standard formatting of statements and the appearance of terminal output. Even one who has never studied BASIC can probably gather what the program does and understand at least vaguely how it works.

```
LISTNH
100     RANDOMIZE
        ! THIS IS A RANDOM DICE ROLL ROUTINE
        ! THE USER CAN SPECIFY HOW MANY DICE TO BE IN
        ! EACH ROLL AND HOW MANY ROLLS ARE TO BE MADE.
        ! WHETHER TO PRINT THE TOTAL OF EACH ROLL IS ALSO
        ! UNDER USER CONTROL
110     PRINT 'THIS PROGRAM GIVES RANDOM DICE ROLLS'
        \ PRINT 'HOW MANY DIES IN EACH ROLL';
        \ INPUT N
        \ PRINT 'HOW MANY ROLLS';
        \ INPUT D
        \ PRINT 'IF YOU WANT THE TOTAL OF EACH ROLL, TYPE Y';
        \ INPUT R$
        \ PRINT
120     FOR J=1 TO D
        \ PRINT 'THE';N; 'DIES OF ROLL';J;'ARE:';
130     FOR I=1 TO N
        \ A%=(INT(6*RND) + 1)
        \ B%=A% + B%
        \ PRINT A%;
        \ NEXT I
140     IF R$ = 'Y' THEN PRINT '---TOTAL OF ROLL =';B%
150     PRINT
        \ B%=0
        \ NEXT J
32767   END

Ready

RUNNH
THIS PROGRAM GIVES RANDOM DICE ROLLS
HOW MANY DIES IN EACH ROLL? 5
HOW MANY ROLLS? 3
IF YOU WANT THE TOTAL OF EACH ROLL, TYPE Y? Y

THE 5 DIES OF ROLL 1 ARE: 4  2  4  4  3 ---TOTAL OF ROLL = 17

THE 5 DIES OF ROLL 2 ARE: 6  5  1  4  5 ---TOTAL OF ROLL = 21

THE 5 DIES OF ROLL 3 ARE: 5  5  4  6  5 ---TOTAL OF ROLL = 25


Ready

RUNNH
THIS PROGRAM GIVES RANDOM DICE ROLLS
HOW MANY DIES IN EACH ROLL? 2
HOW MANY ROLLS? 5
IF YOU WANT THE TOTAL OF EACH ROLL, TYPE Y? N

THE 2 DIES OF ROLL 1 ARE: 2  3
THE 2 DIES OF ROLL 2 ARE: 4  4
THE 2 DIES OF ROLL 3 ARE: 4  3
THE 2 DIES OF ROLL 4 ARE: 3  6
THE 2 DIES OF ROLL 5 ARE: 3  5

Ready
```

Figure 2-1 Sample BASIC-PLUS Program

A user program is composed of lines of statements containing instructions to BASIC-PLUS. Each line of the program begins with a line number, followed by one or more BASIC statements. Line numbers indicate the sequence of statement execution, although this sequence can be interrupted by certain statements. Each statement begins with a word specifying the type of operation to be performed, such as printing, data input, a determination of a condition, or a change of the contents of a variable. If a program line contains multiple statements, those statements must be separated by backslash characters.

2.3 LINE NUMBERS

Each BASIC-PLUS program line is preceded by a line number. Line numbers perform the following:

1. Indicate the order in which statements are normally evaluated.
2. Enable the normal order of evaluation to be changed; that is, the execution of the program can branch or loop through designated statements (this is explained further in the sections on the GOTO, GOSUB, and IF-THEN statements in Chapter 3).
3. Enhance program debugging by permitting modification of any specified program line without affecting any other portion of the program.

Line numbers are in the range of 1 to 32767. BASIC-PLUS maintains programs in line number sequence, rather than the order in which lines are entered to the system. It is good programming practice to number lines in increments of 5 or 10 when first writing a program, to allow for insertion of lines when debugging or enhancing the program.

A program line can have more than one statement. The first statement on a program line must be preceded by a line number, and each subsequent statement must be preceded by a backslash (\). Also, a program line can be continued over several physical terminal lines as long as no integral BASIC elements (e.g., key words, constants, variable names) are broken.

Using multiple statements and multiple terminal lines in a program line saves memory space. It does make debugging and editing of a program more difficult, however.

NOTE

For compatibility with programs written for earlier versions of BASIC, RSTS/E accepts the colon (:) as a valid statement separator.

When a program is executed (with the use of the RUN command), the BASIC-PLUS processor evaluates the statements in the order of their line numbers, starting with the smallest line number and going to the largest.

2.4 STATEMENTS

Each line number is followed by a BASIC-PLUS statement. The first word of a BASIC-PLUS statement identifies the type of statement and informs BASIC-PLUS of the operation to be performed and how to treat the data (if any) that follows the word.

2.4.1 Multiple Statements on a Single Line

More than one statement can be written on a single line as long as each statement (except the last) is terminated with a colon or a backslash. Only the first statement on a line can (and must) have a line number. For example:

```
10 INPUT A,B,C
```

is a single statement line. However:

```
20 LET X=X+1 \ PRINT X,Y,Z \ IF Y=2 GOTO 10
```

is a multiple-statement line containing three statements: a LET, a PRINT, and an IF-GOTO statement.

Any statement can be used anywhere in a multiple-statement line except as noted in individual statement descriptions.

2.4.2 Line-to-Line Statement Continuation

A statement can be continued on successive lines of the program. To continue a statement in NO EXTEND mode, press the LINE FEED key instead of the RETURN key. The LINE FEED performs a carriage return/line feed operation on the terminal and the continuation line does not contain a line number. For example:

```
10 LET W7=(W-X4*3)*(Z-A/ (LF)
(A-B)-17)
```

This is equivalent to:

```
10 LET W7=(W-X4*3)*(Z-A/(A-B)-17)
```

To continue a statement in EXTEND mode, type an ampersand at the end of the line and then press RETURN. Any number of tabs and space characters are allowed between the ampersand and the RETURN. The line feed method used in NO EXTEND is valid in EXTEND, but the syntax

&[tabs/spaces] (RET)

is recommended to ensure compatibility with future versions of BASIC-PLUS. For example:

```
10      EXTEND
200     LET TOTAL.RECEIPTS =                &
                  GROCERIES.IN +            &
                  DELI.IN +                 &
                  PRODUCE.IN +              &
                  DAIRY.IN +                &
                  MEAT.IN +                 &
                  PAPERBACKS.IN
```

The LINE FEED and RETURN keys do not cause a printed character to appear on the page.

There is no limit to the length of a multiple-line statement in either mode.

Where line continuation is used, it must occur between the integral elements of a BASIC-PLUS statement. Examples of BASIC-PLUS elements are a BASIC-PLUS language keyword, an alphanumeric string, a variable name, or a line number. No integral element can be interrupted at the end of one line and continued on another. For example:

```
10 IF A1=0
THEN 100
```

is acceptable where a continuation follows 0. However,

```
10 IF A
1=0 THEN 100
?Illegal conditional clause at line 10
```

is not acceptable. Neither is:

```
10 IF A1=0 THEN 1
00
?Modifier error at line 10
```

Each illegal form generates an error message. A number of multiword phrases are processed as one keyword and cannot be broken by a line continuation.

These phrases are:

GO TO	AS FILE	MAT READ
GO SUB	FOR INPUT AS FILE	MAT PRINT
ON ERROR	FOR OUTPUT AS FILE	MAT INPUT
ON ERROR GO TO	NO EXTEND	MAT LET
INPUT LINE		

Blanks within these phrases may be omitted. For example, GOTO is equivalent to GO TO.

2.5 SPACES AND TABS

In NO EXTEND mode spaces can be used freely throughout the program.

In EXTEND mode, however, spaces are significant, and can be used only between BASIC-PLUS language elements and in alphanumeric strings. Moreover, spaces are required to delimit elements that are not otherwise delimited by a character not in the set of characters that is valid for EXTEND mode variable names (see 2.6.2). For example, consider the following statement lines:

```
(A) 10        LET X = Y*2 + 1

(B) 10LETX=Y*2+1

(C) 10        L  ETX = Y * 2 + 1
```

In NO EXTEND mode the above statements are identical in effect.

In EXTEND mode, however, only (A) is valid. (B) requires a space between LET and X; all other elements are properly delimited by arithmetic symbols. In (C) the keyword LET contains imbedded spaces, which must be removed, and, as in (B), LET and X must be separated by spaces. Note that any number of spaces can follow or precede an element.

Tabs, like spaces, can make a program easier to read. For example:

```
2000    FOR K=1 TO 3
2010            FOR I=1 TO 10
2020                    FOR J=1 TO 10
2040                    A(I,J) = K/(I+J-1)+A(I,J)
2050                    NEXT J
2060            NEXT I
2070    NEXT K
32767   END
```

2.6 EXPRESSIONS

An expression is a group of symbols which can be evaluated by BASIC-PLUS. Expressions are composed of constants, variables, functions, or a combination of the preceding separated by arithmetic, relational, or logical operators.

The following are examples of expressions acceptable to BASIC-PLUS:

Arithmetic Expressions

```
5.135
4% + Z%
A7*(B^2% +1.)
```

Relational Expressions

```
X < Y
Y9% > 0%
A% = B
```

Logical Expressions

```
(A < 0.) AND (B = 1.)
((A > B) OR (C = D)) AND A/B < > C/D
```

Arithmetic expressions yield either floating-point or integer values. Relational expressions yield a truth value that reflects the result of comparing two values. Logical expressions yield a truth value reflecting the existence or non-existence of a specified set of relational or other conditions.

A constant or a variable name with no % or $ suffix indicates a floating-point value. Floating point values are stored in either a 2-word or 4-word floating point format, depending on the type of math package installed.

A constant or variable name with a contiguous % suffix indicates an integer value. An integer value is stored in a single word as a base 2 integer. Chapter 4 describes arithmetic operations, and Appendix E describes integer and floating point formats. Example:

Floating Point	**Integer**
A	A%
9.	9%
B3 + 5.	B3% + 5%
3.1416	3.1416% (stored as 3%)

The use of an explicit decimal point or percent sign is recommended in all numeric constants to avoid unnecessary data conversions and to improve documentation. Mixing of data types in a statement should be avoided and integers should be used whenever possible. When raising to an integer power, the power value should be indicated explicitly as an integer. Raising a floating point number to an integer power does not constitute mode mixing. (However, some examples in this manual do not follow these suggestions, in the interest of readability.)

Another type of expression not described in detail in this section is the string expression, a value that consists of a sequence of characters, each character occupying a byte (i.e., one half of a memory word). A string expression can be expressed either as a constant (a sequence of characters enclosed in quotation marks) or as a variable (a variable name with a $ suffix).

String expressions and operations are described in detail in Chapter 5.

Not all kinds of expressions can be used in all statements, as is explained in the sections describing the individual statements. In the following sections the reader is introduced to the elements which compose BASIC-PLUS expressions.

2.6.1 Numeric Constants

Numeric constants retain a constant value throughout a program, and can be positive or negative. Numeric constants can be written using decimal notation as follows:

```
+2
−3.675
1234.56
−123456
−.000001
```

The example constants would be stored as floating point, since they have no % suffix.

Scientific notation allows further flexibility in number representation. Numeric constants can be written using the letter E to indicate "times 10^n"; the number following the letter E indicates the exponent n.

.000123456	can be written in BASIC-PLUS as 123456E-6
1234560000.	can be written in BASIC-PLUS as 123456.E4
-12345678900.	can be written in BASIC-PLUS as -1.2345679E10

The E format representation of numbers is very flexible since a number such as .001 can be written as 1E-3, 01E-1, 100E-5, or any number of ways within the allowable range of exponents. If more than six digits are generated during any computation, the result of that computation is automatically printed in E format. (If the exponent is negative, a minus sign is printed after the E; if the exponent is positive, a space is printed: 1E-04; 1E 04.)

The combination E7, however, is not a constant, but a variable. The term 1E7 is used to indicate that 1 is multiplied by 10 7, i.e., the number 10000000.

The set of floating-point numbers is approximately as follows:

X = 0, and the range 10^38 < |X| <10^38

The range of integer numbers is −32768 through 32767.

2.6.2 Numeric Variables

A variable is a data item whose value can be changed during program execution. A numeric variable is denoted by a fixed variable name.

In NO EXTEND mode, a variable name consists of a single letter or a single letter followed by a single digit. In EXTEND mode a variable name consists of a single letter followed contiguously by 0 to 29 characters from the set:

A,B, . . . , Z	(letters)
0,1, . . . , 9	(digits)
	(period or point)

A name can also have an FN prefix (denoting a function name), a % suffix (denoting an integer), a $ suffix (denoting a string), or a subscript suffix that consists of a set of subscripts enclosed in parentheses. These prefixes and suffixes are not counted in the 30-character limit.

Variables are assigned values by LET, INPUT, and READ statements. The value assigned to a variable does not change until the next time a LET, INPUT or READ statement is encountered that contains a new value for that variable or until the variable is incremented by a FOR statement. (These conditions are explained further in later sections.) All variables are set equal to 0 before program execution. It is necessary to assign a value to a variable only when an initial value other than 0 is required. However, it is good programming practice to set variables equal to 0 wherever necessary. This ensures that later changes or additions will not cause misinterpretation of values.

2.6.3 Mathematical Operators

BASIC-PLUS automatically performs the mathematical operations of addition, subtraction, multiplication, division and exponentiation. Formulas to be evaluated are represented in a format similar to standard mathematical notation. There are five arithmetic operators used to write such formulas, as follows:

Operator	Example	Meaning
+	A+B	Adds B to A
–	A–B	Subtracts B from A
*	A*B	Multiplies A by B
/	A/B	Divides A by B
^	A^B	Calculates A to the B power, A^B

BASIC-PLUS permits the operator ** in place of ^ (up arrow) to denote the exponentiation operation. For example:

```
A**B
```

indicates the quantity A raised to the B power, equivalent to A^B. The ** operator is included for compatibility with some other BASIC processors. The symbol ^ is the standard BASIC-PLUS symbol for exponentiation and is used throughout this manual.

Unary plus and minus are also allowed, e.g. the – in –A+B or the + in +X–Y. Unary plus is ignored. Unary minus is treated as explained below.

When more than one operation is to be performed in a single formula, rules are observed as to the precedence of the above operators. The arithmetic operations are performed in the following sequence, with the operation described in item 1 having precedence.

1. Any formula within parentheses is evaluated before the parenthesized quantity is used in further computations. Where parentheses are nested, as follows:

   ```
   (A+(B*(D^2)))
   ```

   the innermost parenthetical quantity is calculated first.
2. In the absence of parentheses in a formula, BASIC-PLUS performs operations as follows:

   a. Exponentiation
   b. Unary minus
   c. Multiplication and division
   d. Addition and subtraction

   Thus, for example, –A^B with a unary minus, is a legal expression and is the same as –(A^B). This implies that –2^2 evaluates as –4. The one exception to this rule is that the term A^–B is allowed and is evaluated as A^(–B).

3. In the absence of parentheses in a formula involving more than one operation on the same level (see item 2 above), the operations are performed left to right, in the order that the formula is written. For example:

A/B/C is evaluated as (A/B)/C
A*B/C is evaluated as (A*B)/C

The expression A+B*C^D is evaluated in the following order:

1. C is raised to the D power
2. The result of the first operation is multiplied by B.
3. The result of the previous operation is added to A.

Parentheses are used to indicate any other order of evaluation. For example, to raise the product of B and C to the D power, the user writes the expression as follows:

```
A+(B*C)^D
```

To multiply the quantity A+B by C to the D power, the user writes the expression as follows:

```
(A+B)*C^D
```

The user is encouraged to use parentheses even where they are not strictly required in order to make expressions easier to read and to reduce the possibility of writing an unintended expression.

2.6.4 Relational Symbols

Relational symbols are used in IF-THEN statements (see Section 3.5): in conditional FOR loops (see Section 8.6); in IF, UNLESS, WHILE and UNTIL clauses (see Sections 3.5, 8.5, and 8.7) where it is necessary to compare values; and where any integer expression can be used (see Section 6.6). The relational symbols are as follows (where A and B are numeric variables or expressions):

Mathematical Symbol	BASIC-PLUS Symbol	Example	Meaning
=	=	A=B	A is equal to B
<	<	A<B	A is less than B
<	<=	A<=B	A is less than or equal to B
>	>	A>B	A is greater than 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

The term "approximately equal to" means that the two quantities look the same when printed to six decimal places of precision. Within the computer, floating-point numbers with a fractional part can differ by a miniscule amount in the last decimal place but still be considered equal for all practical purposes. This last decimal place within the computer does not always cause two numbers to have a different value when printed. Numbers are carried internally at greater than 6 digits of precision, but are rounded to 6 digits for output or a comparison. Thus, two numbers identical when rounded to 6 digits of precision are approximately equal, whereas two numbers equal to the internally carried limits of precision are truly equal (=).

2.6.5 Logical Operators

Logical operators are used in IF-THEN and such statements (see Section 3.5) where some condition is used to determine subsequent operations within the user program. For this discussion, A and B are relational expressions having only TRUE (-1) and FALSE (0) values. Logical operators can also be used in certain logical operations involving integers. (See Sections 6.5 and 6.6.) The logical operators are as follows:

Operator	**Example**	**Meaning**
NOT	NOT A	The logical negative of A. If A is true, NOT A is false.
AND	A AND B	The logical product of A and B. A AND B has the value true only if A and B are both true and has the value false if either A or B is false.
OR	A OR B	The logical sum of A and B. A OR B has the value true if either A or B or both is true and has the value false only if both A and B are false.
XOR	A XOR B	The logical exclusive OR of A and B. A XOR B is true if either A or B (but not both) is true, and 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.
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.

The following tables are called truth tables and describe graphically the results of the above logical operations with both A and B given for every possible combination of values.

A	B	A AND B
T	T	T
T	F	F
F	T	F
F	F	F

A	B	A OR B
T	T	T
T	F	T
F	T	T
F	F	F

A	B	A XOR B
T	T	F
T	F	T
F	T	T
F	F	F

A	B	A EQV B
T	T	T
T	F	F
F	T	F
F	F	T

A	B	A IMP B
T	T	T
T	F	F
F	T	T
F	F	T

A	NOT A
T	F
F	T

CHAPTER 3
ELEMENTARY BASIC-PLUS STATEMENTS

This chapter describes the simplest forms of the more elementary BASIC-PLUS statements. These statements are sufficient, by themselves, for the solution of most problems. Once these statements are mastered, the user can investigate the more advanced applications of these statements and the additional statements and features explained in Parts II and III.

The reader should understand that any problem which can be solved with the more advanced techniques can also be solved with the simpler statements, although the process may not be as efficient. As long as the user understands the details of his problem he can represent it in BASIC-PLUS on a number of levels ranging from the simple to the sophisticated.

3.1 REMARKS AND COMMENTS

It is often desirable to insert notes and messages within a user program. Information such as the name and purpose of the program, how to use it, how certain parts of the program work, and expected results at various points is useful in the program for ready reference by anyone using that program.

There are two ways of inserting comments into a user program:

1. Using the REM statement
2. Using the exclamation mark (!)

REM statements can contain any printing characters on the keyboard, except that in EXTEND mode the word REM must be followed by a space or other valid word delimiter. BASIC-PLUS completely ignores anything on a line following the letters REM. (The line number of a REM statement can be used in a GOTO or GOSUB statement, see Sections 3.4 and 3.8.1, as the destination of a jump in program execution.) Typical REM statements are shown below:

```
10        REM - THIS PROGRAM COMPUTES THE
11        REM - ROOTS OF A QUADRATIC EQUATION
```

The exclamation mark is normally used to terminate the executable part of a line and begin the comment part of the line. The ! character can also begin the line, in which case the entire line is treated as a comment. For example:

```
125       LET A=3             ! FIRST VALUE OF A
130       PRINT A/2           ! DOES NOT CHANGE A
140       ! ENTIRE LINE IS COMMENTARY
```

In every statement other than the DATA statement, BASIC-PLUS ignores everything on the line following the exclamation mark. An exclamation mark must not appear on the same program line as a DATA statement unless it is part of an item in the DATA statement.

Messages in REMARK statements are generally called remarks, those after the exclamation mark, comments. Remarks and comments are printed when the user program is listed but do not affect program execution. (They do affect program size, however.)

The lines below indicate three ways of putting the same remark on two lines. Lines 10 and 11 are REM statements. Line 20 is one REM statement broken into two lines with the LINE FEED key. Line 30 is one comment (begun with a !) and broken into two lines with the LINE FEED key. Line 40 is one comment broken into two lines with the ampersand character followed by a RETURN key; this is legal only under EXTEND mode.

```
10        REM - THIS PROGRAM COMPUTES THE (RET)
11        REM - ROOTS OF A QUADRATIC EQUATION (RET)

20        REM - THIS PROGRAM COMPUTES THE (LF)
          ROOTS OF A QUADRATIC EQUATION (RET)

30        ! THIS PROGRAM COMPUTES THE (LF)
          ROOTS OF A QUADRATIC EQUATION (RET)

40        ! THIS PROGRAM COMPUTES THE      & (RET)
          ! ROOTS OF A QUADRATIC EQUATION (RET)
```

When a comment is continued on a subsequent terminal line with an ampersand and carriage return, the continuation line must also start with an exclamation point or it will be interpreted as an executable statement. For example:

```
10        EXTEND
20        !THIS PROGRAM COMPUTES THE & (RET)
          !ROOTS OF A QUADRATIC EQUATION & (RET)
          PRINT 'ENTER THREE COEFFICIENTS'
RUNNH
ENTER THREE COEFFICIENTS

Ready
```

3.2 LET STATEMENT

The LET statement assigns a numeric value to a variable. Each LET statement is of the form:

{LET} <variable> = <expression>

This statement does not indicate algebraic equality, but performs the calculations within the expression (if any) and assigns the numeric value to the indicated variable. For example:

```
10        LET X = X+1
20        LET W2 = (A4-X3)*(Z-A/B)
```

In line 10, the old value of X is increased by 1 and becomes the new value of X. In line 20, the formula on the right-hand side is evaluated and the numeric value assigned to W2.

The LET statement can be a simple numerical assignment, such as

```
50        LET A=35
```

or require the evaluation of a formula so long that it is continued on the next line (see Section 2.3.2).

BASIC-PLUS allows the user to omit the word LET from the LET statement. The user may find it easier to type:

```
10         X=12*(S+7)
```

than

```
10        LET X=12*(S+7)
```

This is a convenience and does not alter the effect of the statement.

The LET statement can be used anywhere in a multiple statement line. For example:

```
    100       X=44 \ Y=X^2 +Y1 \ B2=3.5*A
```

The LET statement allows the user to assign a value to several variables in the same statement. For example:

```
    200       LET X, Y, Z = 5.7
```

causes each of the three variables to be set equal to 5.7.

3.3 PROGRAMMED INPUT AND OUTPUT

This section describes the techniques used in performing BASIC-PLUS program I/O (an abbreviation for the term input/output which includes the processes by which data is brought into and sent out of the computer). The most elementary forms of the PRINT, INPUT, READ, and DATA statements are presented here so that the user can create simple BASIC-PLUS programs and obtain tangible results.

More advanced I/O techniques are described in Part III.

3.3.1 READ, DATA, and RESTORE Statements

READ and DATA statements are coordinated to furnish a fixed list of data values to the user program. A READ statement contains the list of variables whose values are obtained from a DATA statement. Neither statement is operative without the other, and the data types of the READ variables and DATA values must be compatible.

A READ statement is of the form:

READ <variable list>

A DATA statement is of the form:

DATA <value list>

A READ statement causes the variables listed to be assigned values sequentially from the set of DATA statements in the program. Before the program is run, BASIC-PLUS takes all DATA statements in the order they appear and creates a data block. Each time a READ statement is encountered in the program, the data block supplies the next value. If the data block runs out of data, the program is assumed to be finished and an OUT OF DATA message is printed by BASIC-PLUS.

READ and DATA statements appear as follows:

```
    150       READ X, Y%, Z, S1, Y2, Q9
    330       DATA 4,2,1.7
    350       DATA 6.73E-3, -174.321, 3.1415927
```

Note that only numbers are used in this particular DATA statement. (Input of string data is described in Section 5.3.) The assignments performed by line 150 are as follows:

```
  X=4.0
Y%=2%
  Z=1.7
 S1=6.73E-3
Y2=174.321
Q9=3.1415927
```

Data must be read before it can be used in a program; thus READ statements normally occur near the beginning of a program. The location of DATA statements is arbitrary, although their order is significant. A good practice is to collect all DATA statements near the end of the program. A DATA statement must be the only statement or the last statement on a line, while a READ statement can be placed anywhere in a multiple statement line.

NOTE

Comments are not permitted at the end of a DATA statement.

If it should become necessary to use the same data more than once in a program, the RESTORE statement makes it possible to recycle through the complete set of DATA statements in that program, beginning with the lowest numbered DATA statement. The RESTORE statement is of the form:

    RESTORE

For example:

```
300     RESTORE
```

causes the next READ statement following line 30 to begin reading data from the first DATA statement in the program, regardless of where the last data value was found.

The same variable names can be used the second time through the data or not, as is most convenient, since the values are being read as though for the first time. In order to skip unwanted values, dummy variables must be read. In the following example, BASIC-PLUS prints:

```
4     1     2     3
```

on the last line because it did not skip the value for the original N when it executed the loop beginning at line 1600.

```
LISTNH
100     REM     THIS PROGRAM ILLUSTRATES USE OF THE RESTORE
1500    READ N \ PRINT 'VALUES OF X ARE:'
1600    FOR I = 1 TO N \ READ X\ PRINT X,
1700    NEXT I
1800    RESTORE
1900    PRINT \ PRINT 'SECOND LIST OF X VALUES'
2000    PRINT 'FOLLOWING RESTORE STATEMENT:'
2100    FOR I=1 TO N \ READ X \ PRINT X,
2200    NEXT I
6000    DATA 4,1,2
6100    DATA 3,4
32767   END

Ready

RUNNH
VALUES OF X ARE:
 1              2              3              4
SECOND LIST OF X VALUES
FOLLOWING RESTORE STATEMENT:
 4              1              2              3
Ready
```

3.3.2 PRINT Statement

The PRINT statement is used to output data onto the terminal teleprinter. The general format of the PRINT statement is:

PRINT $\{\text{list}\}$

where the list can contain expressions, text strings, or both. As the braces indicate, the list is optional. Used alone, the PRINT statement:

```
250       PRINT
```

causes a carriage return/line feed operation.

PRINT statements can be used to perform calculations and print results. Any expression within the list is evaluated before a value is printed. Consider the following program:

```
LISTNH
2000      LET A=1 \ LET B=2 \ LET C=3+A
2100      PRINT
2200      PRINT A+B+C
32767     END

Ready

RUNNH

 7

Ready
```

All numbers are printed in the form:

$\begin{bmatrix} \text{space} \\ - \end{bmatrix}$ <number> <space>

The PRINT statement can be used anywhere in a multiple statement line. For example:

```
1600     A=1 \ PRINT A \ A=A+5 \ PRINT \ PRINT A
```

would cause the following to be printed on the terminal when executed:

```
RUNNH
 1

 6

Ready
```

Notice that the teleprinter performs a carriage return/line feed at the end of each PRINT statement. Thus the first PRINT statement causes a 1 and a carriage return/line feed, the second PRINT statement is responsible for the blank line, and the third PRINT statement causes a 6 and another carriage return/line feed to be output.

BASIC-PLUS considers the terminal printer to be divided into zones of 14 spaces each. On most terminals the maximum print line contains 72 characters, in which case there are 5 print zones. Terminals with 84 or more print characters per line have additional print zones in units of 14 spaces.

When an item in a PRINT statement is followed by a comma, the next value to be printed appears in the next available print zone. For example:

```
LISTNH
1500     LET A=3 \ LET B=2
1600     PRINT A,B,A+B,A*B,A-B,B-A,A^B,B^(B/A)
32767    END

Ready

RUNNH
 3             2              5              6              1
-1             9              1.5874

Ready
```

Notice that the sixth element in the PRINT list is printed as the first entry on a new line, since a 72-character line has five print zones.

Two commas together in a PRINT statement cause a print zone to be skipped. For example:

```
LISTNH
100      REM      THIS SHOWS HOW TO SKIP A PRINT ZONE
                  AND SHOWS LINE CONTINUATION.
110      LET A=1
         \ LET B=2
         \ PRINT A,B,,A+B
         !NOTE DOUBLED COMMA AFTER B
32767    END

Ready

RUNNH
 1             2                             3

Ready
```

If the last item in a PRINT statement is followed by a comma, no carriage return/line feed is output, and the next value to be printed (by a later PRINT statement) appears in the next available print zone. For example:

```
LISTNH
100      REM      THIS SHORT PROGRAM ILLUSTRATES PRINT FORMAT,
                  USING COMMA AND SPACE
110      LET A=1
         \   B=2
         \   C=3
120      ! IN LINE 110, NOTE USE OF BACKSLASH (\)  TO PUT MULTIPLE
         ! STATEMENTS ON ONE PROGRAM LINE; THAT IS, ON ONE LINE
         ! NUMBER.  MINIMIZING THE NUMBER OF LINE NUMBERS RESULTS
         ! IN MORE  EFFICIENT CODE.  EACH STATEMENT IS PLACED
         ! ON ITS OWN PHYSICAL LINE BY USING LINE CONTINUATION.
         ! TO CONTINUE A LINE IN EITHER EXTEND OR NOEXTEND MODE
         ! PRESS THE LINE FEED KEY.
130      PRINT A,
140      PRINT B
150      PRINT C
         ! THE THREE PRINT STATEMENTS ARE ON SEPARATE
         ! LINE NUMBERS.  THIS IS LEGAL BUT LESS EFFICIENT.
32767    END

Ready

RUNNH
 1             2
 3

Ready
```

If a tighter packing of printed values is desired, the semicolon character can be used in place of the comma. A semicolon causes no extra spaces to be output. A comma causes the print head to move at least one space to the next print zone or possibly perform a carriage return/line feed. The following example shows the effects of the semicolon and comma.

```
LISTNH
100      LET A=1
         \ LET B=2
         \ LET C=3
110      PRINT A;B;C;
120      PRINT A+1; B+1; C+1
130      PRINT A,B,C,
140      PRINT A+B+C,C^B,C^(B^C)
150      PRINT
160      PRINT 50,100;150;200,250,300
32767    END

Ready

RUNNH
 1  2  3  2  3  4
 1             2             3             6             9
 6561

 50            100  150  200               250           300

Ready
```

The PRINT statement can be used to print a message, either alone or together with the evaluation and printing of numeric values. Characters are delimited for printing by placing single or double quotation marks at each end of the string. The same type of quotation mark must be used at the beginning and the end of each string.

```
LISTNH
100      PRINT "TIME'S UP"
110      PRINT 'QUOTH THE RAVEN, "NEVERMORE"'
32767    END

Ready

RUNNH
TIME'S UP
QUOTH THE RAVEN, "NEVERMORE"

Ready
```

As another example, consider the following line:

```
550      X=87.4
580      PRINT 'AVERAGE GRADE IS';X
```

which prints the following:

```
AVERAGE GRADE IS 87.4
```

When a character string is printed, only the characters between the quotes appear; no leading or trailing spaces are added. Leading and trailing spaces can be added within the quotation marks using the keyboard space bar; spaces appear in the printout exactly as they are typed within the quotation marks.

When a comma separates a text string from another PRINT list item, the item is printed at the beginning of the next available print zone. Semicolons separating text strings from other items are optional, but should be included for compatibility with other versions of BASIC.

```
580      PRINT 'AVERAGE GRADE IS' X
```

A comma or semicolon appearing as the last item of a PRINT list suppresses the carriage return/line feed operation.

The following example demonstrates the use of the comma and semicolon as formatting characters.

```
LISTNH
150        INPUT 'STUDENT NUMBER';X
160        INPUT 'GRADE';G
170        INPUT 'AVERAGE';A
180        INPUT 'RANK';R

           !  IRRELEVANT CODE OMITTED

300        N=562
640        PRINT 'STUDENT NUMBER' X, 'GRADE ='G;
650        PRINT 'AVERAGE =' A
660        PRINT '      RANK IN CLASS'R; 'OF' N
32767      END

Ready

RUNNH
STUDENT NUMBER? 2574
GRADE? 89
AVERAGE? 90.6
RANK? 14
STUDENT NUMBER 2574            GRADE = 89 AVERAGE = 90.6
      RANK IN CLASS 14 OF 562

Ready
```

3.3.3 INPUT Statement

The second way to provide data to a program is with an INPUT statement. This statement is used when writing a program to process data to be supplied while the program is running. During execution, the programmer can type values as the computer asks for them. (Nonterminal INPUT is described in Part III.) Depending upon how many values are to be accepted by the INPUT command, the programmer may include a PRINT statement that reminds the user of the kind of input required. Such messages can also be generated from within the INPUT statement, as shown in the accompanying example.

The INPUT statement is of the form:

    INPUT <list>

For example:

```
10        INPUT A,B,C
```

causes the computer to pause during execution, print a question mark, and wait for the user to type three numeric values separated by commas. The values typed are entered to the computer when the user presses the RETURN key, LINE FEED key, or the ESCAPE key (ESC on some terminals, ALT MODE on others).

In the following example, the executing program requests data by asking four questions: INTEREST IN PERCENT?, AMOUNT OF LOAN?, NUMBER OF YEARS?, and NO. OF PAYMENTS PER YEAR?. The programmer knows which value is requested and proceeds to type and enter the appropriate value.

```
LISTNH
10       REMARK - THIS PROGRAM COMPUTES INTEREST PAYMENTS
20       INPUT 'INTEREST IN PERCENT';J
30       LET J=J/100
40       INPUT 'AMOUNT OF LOAN';A
50       INPUT 'NUMBER OF YEARS';N
60       INPUT 'NUMBER OF PAYMENTS PER YEAR';M
70       LET N=N*M  \  I=J/M  \  B=1+I
80       LET R=A*I/(1-1/B^N)
90       PRINT
100      PRINT 'AMOUNT PER PAYMENT ='; INT(R*10^2+.5)/10^2
110      PRINT 'TOTAL INTEREST     ='; INT((R*N-A)*10^2+.5)/10^2
120      PRINT
130      LET B=A
140      PRINT 'INTEREST   APP TO PRIN    BALANCE OF PRIN'
150      LET L=B*I  \  P=R-L  \  B=B-P
160      PRINT INT(L*10^2+.5)/10^2,
170      PRINT INT(P*10^2+.5)/10^2,
180      PRINT INT(B*10^2+.5)/10^2
190      IF B>=R GOTO 150
200      PRINT INT((B*I)*10^2+.5)/10^2, INT((R-B*I)*10^2+.5)/10^2
210      PRINT 'LAST PAYMENT ='; INT((B*I+B)*10^2+.5)/10^2
32767    END

Ready

RUNNH
INTEREST IN PERCENT? 10
AMOUNT OF LOAN? 6000
NUMBER OF YEARS? 1
NUMBER OF PAYMENTS PER YEAR? 12

AMOUNT PER PAYMENT = 527.5
TOTAL INTEREST     = 329.94

INTEREST   APP TO PRIN    BALANCE OF PRIN
 50           477.5          5522.5
 46.02        481.47         5041.03
 42.01        485.49         4555.54
 37.96        489.53         4066.01
 33.88        493.61         3572.4
 29.77        497.73         3074.67
 25.62        501.87         2572.8
 21.44        506.06         2066.75
 17.22        510.27         1556.47
 12.97        514.52         1041.95
 8.68         518.81         523.14
 4.36         523.14
LAST PAYMENT = 527.5

Ready
```

As in the previous program, the question mark generated by BASIC-PLUS is grammatically useful if a printed question is to prompt the typing of the input values.

The output for the program begins after the command RUNNH and includes a verbal description of the numbers. This verbal description on the output is optional with the programmer, although it has a definite advantage in ease of use and understanding.

When the correct number of variables has been typed in answer to the printed ? character, pressing the RETURN key enters the values to the computer. If too few values are typed, the computer prints another ? to indicate that more data is requested. If too many values are typed, the excess data on that line is ignored.

Messages to be printed at execution time can be inserted within the INPUT statement itself. The message is set off by single or double quotes from the other arguments of the INPUT statement. For example:

```
100      INPUT 'YOUR AGE IS ';A
```

is equivalent to

```
100      PRINT 'YOUR AGE IS';
110      INPUT A
```

The use of the comma or semicolon character (or no character) to separate a character string to be printed from input variable names is analogous to the PRINT statement (see Section 3.3.2).

3.4 UNCONDITIONAL BRANCH, GOTO STATEMENT

The GOTO statement transfers program execution immediately and unconditionally to a specified program line; usually the specified line is not the next sequential line in the program. The general format of the statement is as follows:

GOTO <line number>

The line number to which the program jumps can be either greater than or less than the current line number. It is thus possible to jump forward or backward within a program.

Consider the following example:

```
LISTNH
10       LET A=2
20       GO TO 50
30       LET A=SQR(A+14)
50       PRINT A,A*A
32767    END

Ready

RUNNH
 2              4

Ready
```

When the program encounters line 20, control transfers to line 50. After line 50 is executed, the program terminates. Line 30 is never executed. Any number of lines can be skipped in either direction.

When written as part of a multiple statement line, GOTO should always be the last statement on the line, since any statement following the GOTO on the same line is never executed. For example:

```
110      LET A=ATN(1)              110      LET A=ATN(1)
         \PRINT A                           \GO TO 370
         \GO TO 370                         \PRINT A
370      PRINT 'FINISHED'          370      PRINT 'FINISHED'
32767    END                       32767    END
RUNNH                              RUNNH
 .785398                           FINISHED
FINISHED
                                   Ready
Ready
```

3.5 CONDITIONAL BRANCH, IF-THEN AND IF-GOTO STATEMENTS

The IF-THEN and IF-GOTO statements are used to transfer conditionally from the normal consecutive order of statement numbers, depending upon some mathematical relation or relations. The basic format of the IF statement is as follows:

```
                     ┌ THEN <statement>   ┐
IF <condition>       │ THEN <line number> │
                     └ GOTO <line number> ┘
```

The specified condition is tested. If it is false, control proceeds to the statement following the IF statement (the next sequentially numbered line). If the condition is true, the statement following THEN is executed or control is transferred to the line number given after THEN or GOTO. (An extension of this statement, the IF-THEN-ELSE statement, is described in Section 8.5.)

The deciding condition can be either a simple relational expression in which two mathematical expressions are separated by a relational operator, or a logical expression in which two relational or logical expressions are separated by a logical operator. For example:

Relational Expression	**Logical Expression**
A + 2 > B	A > B AND B < = SQR(C)

Both types of condition, when evaluated, are either true or false; no numeric value is associated with the results of an IF statement. The relational and logical operators are described in Sections 2.6.4 and 2.6.5 and are presented in Appendix A for reference.

```
75       IF A*B>=B*(B+1) THEN LET D4=D4+1
```

In the above line the quantities A*B and B*(B+1) are compared. If the first value is greater than or equal to the second value, the variable D4 is incremented by 1. If B*(B+1) is greater than A*B, D4 is not incremented and control passes immediately to the next line following line 75.

When a line number follows the word THEN, the IF-THEN statement is the same as the IF-GOTO statement. The word THEN can also be followed by any BASIC-PLUS statement, including another IF statement. For example:

```
250      IF A>B THEN IF B>C THEN PRINT 'A>B>C'
350      IF A>B AND B>C THEN PRINT 'A>B>C'
```

The preceding two lines are logically equivalent and perform the following operation:

if B is both less than A and greater than C, the message

A > B > C

is printed; otherwise the next line is executed.

In the following example, the IF-GOTO statement in line 110 is used to limit the value of the variable A in line 100. Execution of the loop continues until the relationship A > 4 is true, then immediately branches to line 32767 to end the program. (A program loop is a series of statements which are written so that, when the statements have been executed, control transfers to the beginning of the statements. This process continues to occur until some terminal condition is reached.)

```
LISTNH
100       LET A=A+1
          \ X=A^2
110       IF A>4 GO TO 32767
120       PRINT 'X='X; ', AND VALUE OF A IS' A
130       GO TO 100
32767     END

Ready

RUNNH
X= 1 , AND VALUE OF A IS 1
X= 4 , AND VALUE OF A IS 2
X= 9 , AND VALUE OF A IS 3
X= 16 , AND VALUE OF A IS 4

Ready
```

(The novice BASIC-PLUS programmer is advised to follow the operation of the computer through these short example programs.)

In IF statements, the following priorities are associated with each operator, in order to provide unambiguous evaluation of the conditions specified (where item 1 has the highest priority):

1. Expressions in parentheses
2. Intrinsic or user-defined functions
3. Exponentiation (^)
4. Unary minus (-), that is, a negative number or variable such as -3, -A, etc.
5. Multiplication and division (* and /)
6. Addition and subtraction (+ and -)
7. Relational operators (=, <, <=, >, >=, ==, < >)
8. NOT
9. AND
10. OR and XOR
11. IMP
12. EQV

For each class of operators indicated above, operations proceed from left to right.

Examples of IF-THEN statements follow:

```
100       IF A>B THEN 340
200       IF A=B OR B=C THEN 260
300       IF A>B THEN A=-B          ! CONDITIONAL ASSIGNMENT
400       IF X>Y IMP Y>Z THEN PRINT "QED"
```

An IF statement would normally be the last statement on a multiple statement line (to avoid confusion); however, the following rules govern the transfer path of the IF statement in other positions:

1. The physically last THEN clause is considered to be followed by the next statement (or statements) on the line:

```
LISTNH
90        INPUT 'ENTER A VALUE';A
100       IF A=1 THEN PRINT A;
          \ PRINT 'TRUE CASE'
          \ GOTO 32767
110       PRINT 'NOT = 1'
32767     END

Ready

RUNNH
ENTER A VALUE? 2
NOT = 1

Ready

RUNNH
ENTER A VALUE? 1
 1 TRUE CASE

Ready
```

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

```
LISTNH
190       INPUT 'ENTER A,B, AND C';A,B,C
200       IF A>B THEN IF B>C THEN PRINT 'A>C'
          \ GO TO 32767
210       PRINT 'DOUBLE CONDITION NOT TRUE'
32767     END

Ready

RUNNH
ENTER A,B, AND C? 12,34,5
DOUBLE CONDITION NOT TRUE

Ready

RUNNH
ENTER A,B, AND C? 123,45,6
A>C

Ready
```

Only in the case where "A > C" is printed is the statement GOTO 32767 executed.

3.6 PROGRAM LOOPS

Loops were first mentioned in the section of the IF-THEN and IF-GOTO statements. Programs frequently are designed to perform certain instructional sequences repetitively. Computers are particularly well suited for such tasks. With simple tasks, such as computing a list of prime numbers between 1 and 1,000,000, a computer can perform the operations and obtain correct results in a minimal amount of time. To write a loop, the programmer must ensure that the series of statements is repeated until a terminal condition is met.

Programs containing loops can be illustrated by using two versions of a program to print a table of the positive integers 1 through 100 together with the square root of each. Without a loop, the first program is 101 lines long and reads:

```
10          PRINT 1, SQR(1)
20          PRINT 2, SQR(2)
30          PRINT 3, SQR(3)
                 .
                 .
                 .
990         PRINT 99, SQR(99)
1000        PRINT 100, SQR(100)
32767       END
```

With the following program example, using a simple sort of loop, the same table is obtained with fewer lines:

```
10          LET X=1
20          PRINT X, SQR(X)
30          LET X=X+1
40          IF X<=100 THEN 20
32767       END
```

Statement 10 assigns a value of 1 to X, thus setting up the initial conditions of the loop. In line 20, both the value of X and its square root are printed. In line 30, X is incremented by 1. Line 40 asks whether X is still less than or equal to 100; if so, BASIC-PLUS returns to print the next value of X and its square root. This process is repeated until the loop has been executed 100 times. After the number 100 and its square root have been printed, X becomes 101. The condition in line 40 is now false so control does not return to line 20, but goes to line 32767 which ends the program.

All program loops have four characteristic parts:

1. Initialization, the conditions which must exist for the first execution of the loop (line 10 above).
2. The body of the loop in which the operation which is to be repeated is performed (line 20 above).
3. Modification, which alters some value and makes each execution of the loop different from the one before and the one after (line 30 above).
4. Termination condition, an exit test which, when satisfied, completes the loop (line 40 above). Execution continues to the program statements following the loop.

3.6.1 FOR and NEXT Statements

The FOR statement is of the form:

FOR <variable> = <expression> TO <expression> {STEP <expression>}

For example:

```
110       FOR K=2 TO 20 STEP 2
```

which causes program execution to cycle through the designated loop using K as 2, 4, 6, 8, ..., 20 in calculations involving K. When K=20, the loop is left behind and the program control passes to the line following the associated NEXT statement. The variable in the FOR statement, K in the preceding example, is known as the control variable.

The control variable must be unsubscripted, although a common use of such loops is to deal with subscripted variables using the control variable as the subscript of a previously defined variable (this is explained in further detail in Section 3.6.2). The expressions in the FOR statement can be any acceptable BASIC-PLUS expression as defined in Section 2.6.

The NEXT statement signals the end of the loop which began with the FOR statement. The NEXT statement is of the form:

NEXT <variable>

where the variable is the same variable specified in the FOR statement. Together the FOR and NEXT statements describe the boundaries of the program loop. When execution encounters the NEXT statement, the computer adds the STEP expression value to the variable and checks to see if the variable is still less than or equal to the terminal expression value. When the variable exceeds the terminal expression value, control falls through the loop to the statement following the NEXT statement.

If the STEP expression is omitted from the FOR statement, +1 is the assumed value. Since +1 is a common STEP value, that portion of the statement is frequently omitted.

The expressions within the FOR statement are evaluated once upon initial entry to the loop. The test for completion of the loop is made prior to each execution of the loop. (If the test fails initially, the loop is never executed.)

The control variable can be modified within the loop. When control falls through the loop, the control variable retains the last value used within the loop.

The following is a demonstration of a simple FOR-NEXT loop. The loop is executed 10 times; the value of I is 10 when control leaves the loop and +1 is the assumed STEP value:

```
LISTNH
10        FOR I=1 TO 10
20        PRINT I;
30        NEXT I
40        PRINT I

Ready

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

Ready
```

The loop itself is lines 10 through 30. The numbers 1 through 10 are printed when the loop is executed. After I=10, control passes to line 40 which causes 10 to be printed again. If line 10 had been:

```
10        FOR I=10 TO 1 STEP -1
```

the value printed by line 40 would be 1.

```
        100     FOR I=2 TO 44 STEP 2
                \ LET I=44
                \ NEXT I
```

The above loop is only executed once since the value of I=44 has been reached and the termination condition is satisfied.

If, however, the initial value of the variable is greater than the terminal value, the loop is not executed at all. A statement of the format:

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

cannot be used to begin a loop, although a statement like the following will initialize execution of a loop properly:

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

For positive STEP values, the loop is executed until incrementing the control variable would cause it to be greater than its final value. For negative STEP values, the loop continues until incrementing the control variable would cause it to be less than its final value.

FOR loops can be nested but not overlapped. The depth of nesting depends upon the amount of user storage space available (in other words, upon the size of the user program and the amount of memory each user has available). Nesting is a programming technique in which one or more loops are completely within another loop. The field of one loop (the numbered lines from the FOR statement to the corresponding NEXT statement, inclusive) must not cross the field of another loop. Figure 3-1 illustrates correct and incorrect nesting of loops.

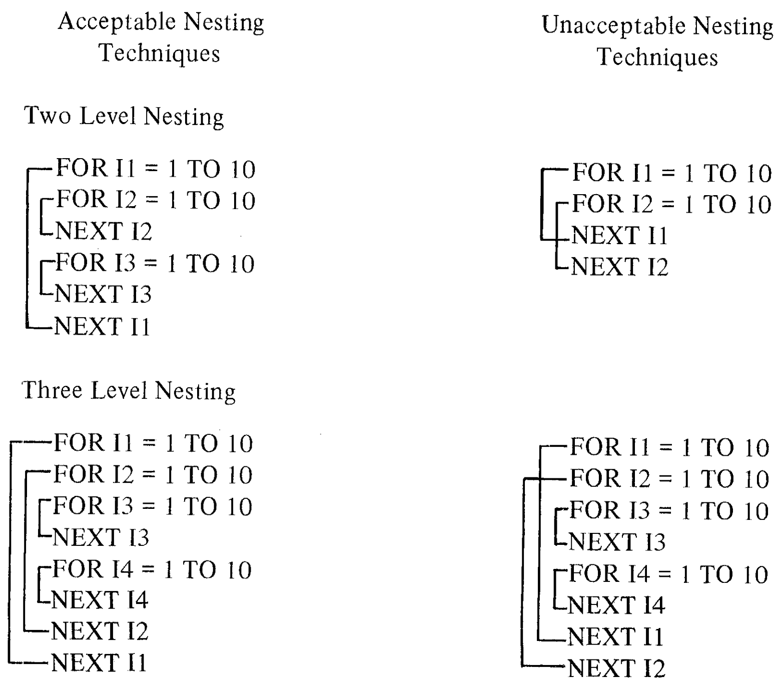


Figure 3-1 Correct and Incorrect Nesting

An example of nested FOR-NEXT loops is shown below:

```
Ready

LISTNH
100      FOR A%=1% TO 5%
110      FOR B%=2% TO 10% STEP 2%
120      PRINT A%;B%,
130      NEXT B%
140      PRINT
150      NEXT A%
32767    END

Ready

RUNNH
 1  2          1  4          1  6          1  8          1  10

 2  2          2  4          2  6          2  8          2  10

 3  2          3  4          3  6          3  8          3  10

 4  2          4  4          4  6          4  8          4  10

 5  2          5  4          5  6          5  8          5  10
```

It is possible to exit from a FOR-NEXT loop without the control variable reaching the termination value. A conditional or unconditional transfer can be used to leave a loop. Control can only transfer into a loop which had been left earlier without being completed, ensuring that termination and STEP values are assigned.

Both FOR and NEXT statements can appear anywhere in a multiple statement line. For example:

```
LISTNH
100      FOR I=1 TO 10 STEP 5\ NEXT I \PRINT 'I=';I

Ready

RUNNH
I= 6

Ready
```

Neither the FOR nor NEXT statement can be executed conditionally in an IF statement. The following statements are incorrect:

```
790      IF I <> J THEN NEXT I
800      IF I=J THEN FOR I=1 TO J
```

3.6.2 Subscripted Variables and the DIM Statement

In addition to the simple variables which were described in Chapter 2, BASIC-PLUS allows the use of subscripted variables. Subscripted variables provide the programmer with additional computing capabilities for dealing with lists, tables, matrices, or any set of related variables. In BASIC-PLUS, variables are allowed one or two subscripts.

The name of a subscripted variable is any acceptable BASIC-PLUS variable name followed by one or two integer expressions in parentheses. For example, a list might be described as A(I) where I goes from 0 to 5 as shown below (all matrices are created with a zero element, even if that element is never specified):

A(0), A(1), A(2), A(3), A(4), A(5)

This allows the programmer to reference each of six elements in the list, which can be considered a 1-dimensional algebraic matrix as follows:

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

A 2-dimensional matrix B(I, J) can be defined in a similar manner and graphically illustrated in Figure 3-2.

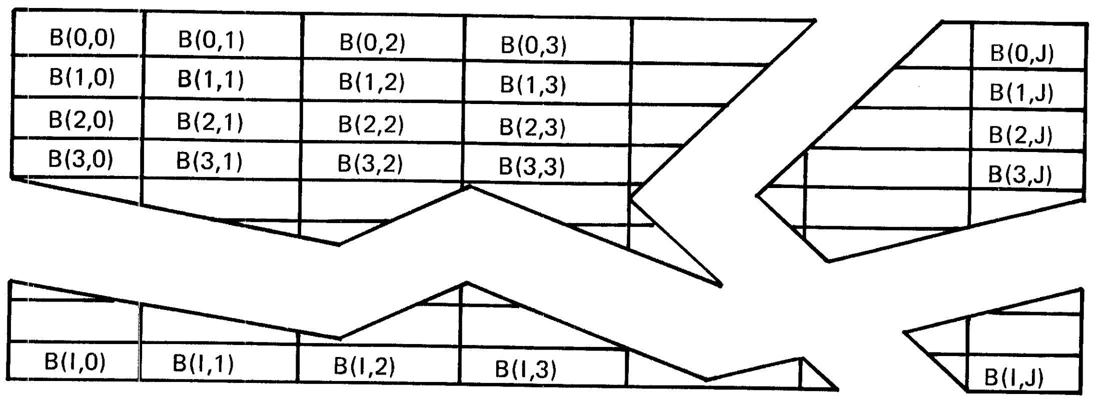


Figure 3-2 Matrix Structure

Subscripts used with subscripted variables can be constants or any legal numeric expression.

It is possible to use the same variable name as both a subscripted and an unsubscripted variable. Both A and A(I) are valid variables and can be used in the same program without affecting one another. However, BASIC-PLUS does not accept the same variable name as both a singly and a doubly subscripted variable name in the same program.

NOTE

There are cases in BASIC-PLUS where a variable name without subscripts refers to an entire matrix, and not to a simple numeric variable. See the CHANGE statement, described in Section 5.2, and the MAT statements described in Chapter 7.

A dimension (DIM) statement is used to define the maximum number of elements in a matrix. ("Matrix" is the general term used in this manual to describe all the elements of a subscripted variable.) The DIM statement is of the form:

DIM <variable (n)>, <variable (n, m)>, . . .

Where the variables specified are indicated with their maximum subscript value(s).

For example:

```
110       DIM X(5), Y(4,2), A(10,10)
120       DIM I4(100)
```

Only non-negative integer constants can be used in DIM statements to define the size of a matrix. Any number of matrices can be defined in a single DIM statement as long as their representations are separated by commas.

If a subscripted variable is used without appearing in a DIM statement, it is assumed to be dimensioned to length 10 in each dimension (that is, having 11 elements in each dimension, 0 through 10). However, all matrices should be correctly dimensioned in a program. DIM statements are usually grouped together among the first lines of a program.

The first element of every matrix is automatically assumed to have a subscript of 0. Dimensioning A(6, 10) sets up room for a matrix with 7 rows and 11 columns. This zero element is illustrated in the following program:

```
LISTNH
10        REM - MATRIX CHECK PROGRAM
20        DIM A(6,10)
30        FOR I=0 TO 6
40        LET A (I,0)=I
50        FOR J=0 TO 10
60        LET A(0,J)=J
70        PRINT A(I,J);
          \ NEXT J
          \ PRINT
          \ NEXT I
32767     END

Ready

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

Ready
```

Notice that a matrix element, like a simple variable, has a value of 0 until it is assigned a value.

To conserve memory, the user may use the zero-subscripted elements set up within the matrix. For example, DIM A(5, 9) generates a 6 X 10 matrix that can be referenced beginning with the A(0, 0) element.

The size and number of matrices which can be defined depend upon the amount of user storage space available.

Additional information on matrices can be found in Chapter 7.

A DIM statement can be placed anywhere in a multiple statement line and anywhere in the program. It need not appear prior to the first reference to an array that it defines. DIM statements are usually placed at or near the beginning of a program, however, to make them easy to locate should changes be indicated.

3.7 MATHEMATICAL FUNCTIONS

Within the course of a user's programming experience, he encounters many cases where relatively common mathematical operations are performed. The results of these common operations can often be found in volumes of mathematical tables; i.e., sine, cosine, square root, log, etc. Since it is this sort of operation that computers perform with speed and accuracy, such operations are built into BASIC-PLUS. The user need never consult tables to obtain the value of the sine of 23 degrees or the natural log of 144. When such values are to be used in an expression, intrinsic functions, such as:

```
SIN (23.*PI/180.)
LOG (144.)
```

are substituted.

The various mathematical functions available in BASIC-PLUS are detailed in Table 3-1.

Table 3-1
Mathematical Functions

Function Code	Meaning
ABS(X)	Returns the absolute value of X
SGN(X)	Returns the sign function of X, a value of 1 preceded by the sign of X, SGN(0)=0
INT(X)	Returns the greatest integer which is less than or equal to X, (INT(-.5)=-1)
FIX(X)	Returns the truncated value of X, SGN(X)*INT(ABS(X)), (FIX(-.5)=0)
COS(X)	Returns the cosine of X (X in radians)
SIN(X)	Returns the sine of X (X in radians)
TAN(X)	Returns the tangent of X (X in radians)
ATN(X)	Returns the arctangent (in radians) of X
SQR(X)	Returns the square root of X
EXP(X)	Returns the value of e^X, where e=2.71828 . . .
LOG(X)	Returns the natural logarithm of X, $\log_e X$
LOG10(X)	Returns the common logarithm of X, $\log_{10} X$
PI	Has a constant value of 3.1415927
RND(X) or RND	Returns a random number between 0 and 1. Unless the RANDOMIZE statement is encountered in the RND program prior to encountering the RND function, the same sequence of random numbers is generated each time a program is run. The value of X is ignored, and can be omitted.

Most of these functions are self-explanatory. Those which are not are explained in the following sections.

3.7.1 Sign Function, SGN(X)

The sign function returns a value of +1 if X is a positive value, 0 if X is 0, and -1 if X is negative. For example: SGN(3.42) = 1, SGN(-42) = -1, and SGN(23-23) = 0.

```
LISTNH
10       REM - SGN FUNCTION DEMO
100      READ A,B
         \ PRINT 'A=';A, 'B=';B
110      PRINT 'SGN(A)=';SGN(A), 'SGN(B)=' SGN(B)
120      PRINT 'SGN(INT(A))=';SGN(INT(A))
1000     DATA    -7.32,  0.44
32767    END

Ready

RUNNH
A=-7.32          B= .44
SGN(A)=-1        SGN(B)= 1
SGN(INT(A))=-1

Ready
```

3.7.2 Integer Function, INT(X)

The integer function returns the value of the greatest integer not greater than X. For example, INT(34.67) = 34. INT can be used to round numbers to the nearest integer by asking for INT(X+.5). For example, INT(34.67+.5) = 35. INT can also be used to round to any given decimal place, by asking for

INT(X*10.^D%+.5)/10.^D%

Where D is the number of decimal places desired, as in the following program:

```
LISTNH
10       ! DEMONSTRATION OF INTEGER (INT) FUNCTION
         \!       INT DOES NOT ROUND TO NEAREST
         \!       INTEGER, BUT DROPS THE FRACTION PART
100      INPUT 'NUMBER TO BE PROCESSED BY INT FUNCTION';A
110      INPUT 'NUMBER OF DECIMAL PLACES FOR ROUNDING';D
120      PRINT 'TRUNCATED INTEGER=';INT(A)
130      PRINT 'ROUNDED INTEGER=';INT(A+.5)
140      PRINT 'ROUNDED TO ';D; 'PLACES=';
                  INT(A*10^D+.5)/(10^D)
150      PRINT
160      PRINT 'ENTER ANOTHER NUMBER, TYPE A ZERO TO STOP --'
170      INPUT A
180      IF A<> 0 GO TO 110
32767    END

Ready
```

```
RUNNH
NUMBER TO BE PROCESSED BY INT FUNCTION? 23.67
NUMBER OF DECIMAL PLACES FOR ROUNDING? 1
TRUNCATED INTEGER= 23
ROUNDED INTEGER= 24
ROUNDED TO  1 PLACES= 23.7

ENTER ANOTHER NUMBER, TYPE A ZERO TO STOP --
? 456.50505
NUMBER OF DECIMAL PLACES FOR ROUNDING? 2
TRUNCATED INTEGER= 456
ROUNDED INTEGER= 457
ROUNDED TO  2 PLACES= 456.51

ENTER ANOTHER NUMBER, TYPE A ZERO TO STOP --
? 0

Ready
```

For negative numbers, the largest integer contained in the number is a negative number with the same or a larger absolute value. For example: INT(-23) = -23, but INT(-14.39) = -15.

3.7.3 Random Number Function, RND(X)

The random number function produces a random number between 0 and 1. The numbers are reproducible in the same order for later checking of a program. The argument X in the RND(X) function call can be any number, or simply omitted as that value is ignored.

```
LISTNH
10      REM - RANDOM NUMBER DEMONSTRATION
20      INPUT "HOW MANY RANDOM NUMBERS";N
30      FOR I=1 TO N
40      PRINT RND,
50      NEXT I
32767   END

Ready

RUNNH
HOW MANY RANDOM NUMBERS? 13
 .204935        .229581        .533074        .132211        .995602
 .783713        .741854        .397713        .709588        .67811
 .682372        .991239        .806084
Ready
```

In order to obtain random digits from 0 to 9, change line 40 to read:

```
40        PRINT INT(10*RND),
```

and tell BASIC-PLUS to run the program again. This time the results are:

```
RUNNH
HOW MANY RANDOM NUMBERS? 13
 2              2              5              1              9
 7              7              3              7              6
 6              9              8
Ready
```

It is possible to generate random numbers over any range. In general, if the range (A, B) is desired, use:

(B-A)*RND(X)+A or the equivalent (B-A)*RND+A

to produce a random number in the range A<n<B.

3.7.4 RANDOMIZE Statement

The RANDOMIZE statement is written as follows:

RANDOMIZE

or, alternatively:

RANDOM

If the random number generator is to calculate different random numbers every time a program is run, the RANDOMIZE statement is used. RANDOMIZE is placed before the first use of random numbers (the RND function) in the program. When executed, RANDOMIZE causes the RND function to choose a random starting value, so that the same program run twice gives different results. For this reason, it is a good practice to debug a program completely before inserting the RANDOMIZE statement.

To demonstrate the effect of the RANDOMIZE statement on two runs of the same program, the RANDOMIZE statement is included as statement 15 in the following program:

```
LISTNH
10        REM - RANDOM NUMBER DEMONSTRATION
15        RANDOMIZE
20        INPUT "HOW MANY RANDOM NUMBERS";N
30        FOR I=1 TO N
40        PRINT RND,
50        NEXT I
32767     END

Ready
```

```
RUNNH
HOW MANY RANDOM NUMBERS? 35
 .541559        .249281        .621653        .486387        .32345
 .563214        .468236        .740494        .228836        .708573
 .191913        .774316        .918683        .543249        .99135
 .058857        .430996        .562636E-1     .458616        .245325
 .344403        .858497        .513523E-1     .581639        .276619E-1
 .931222        .338373        .649245        .850108        .257448
 .893713        .452437E-1     .228047        .961087        .714104

Ready

RUNNH
HOW MANY RANDOM NUMBERS? 12
 .448782        .692627        .116732        .466741        .749863
 .29851         .422888E-1     .567144        .022262        .292799E-1
 .975321        .588409

Ready
```

The output from each run is different.

3.7.5 User-Defined Functions

In some programs it may be necessary to execute the same sequence of statements or mathematical formulas in several different places. BASIC-PLUS allows the programmer to define his own functions and call these functions in the same way he would call the standard system functions, such as RND, SQR, or COS.

These user-defined functions consist of a function name, the first two letters of which are FN followed by any valid variable name. For example:

FNA
FNA1

The function can be defined anywhere in the program, even before its first use. The defining or DEF statement is formed as follows:

DEF FN α (arguments) = <expression (arguments)>

where α is any legal variable name. The arguments may consist of zero to five dummy variables. The expression, however, need not contain all the arguments and may contain other program variables not among the arguments. For example:

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

causes a later statement:

```
110       LET R =FNA(4) +1
```

to be evaluated as R=17. As another example:

```
250       DEF FNB(A,B) = A+X^2
260       Y=FNB(14.4,R3)
```

causes the function to be evaluated with the current value of the variable X within the program. In this case the dummy argument B (which becomes the actual argument R3 in the function call) is unused.

The following two programs each produce the same output.

Program 1:

```
LISTNH
10          ! DEMO OF FUNCTION DEFINITION
100         DEF FNS(A)=A^A
110         FOR I=1 TO 5
            \ PRINT I, FNS(I)
            \ NEXT I
32767       END

Ready
```

Program 2:

```
LISTNH
10          ! DEMO OF FUNCTION DEFINITION
100         DEF FNS(X)=X^X
110         FOR I=1 TO 5
            \ PRINT I, FNS(I)
            \ NEXT I
32767       END

Ready
```

The output is the following:

```
RUNNH
 1             1
 2             4
 3             27
 4             256
 5             3125

Ready
```

DEF statement arguments are formal variables. When the defined function is later invoked, the variables are keyed by position in the argument list, not by the characters used to form the arguments. The function itself can be defined in the DEF statement in terms of numbers, variables, other functions, or mathematical expressions. For example:

```
LISTNH
100         DEF FNA(X) = X^2+3*X+4
200         DEF FNB(X) = FNA(X)/2 + FNA(X)
300         DEF FNC(X) = SQR(X+4) + 1
```

The statement in which the user-defined function appears can have that function combined with numbers, variables, other functions, or mathematical expressions. For example:

```
40        LET R = FNA(X+Y+Z)*N/(Y^2+D)
```

A user-defined function can be a function of zero to five variables, as shown below:

```
25        DEF FNL(X,Y,Z) = SQR(X^2 + Y^2 + Z^2)
```

A later statement in a program containing the above user-defined function might look like the following:

```
55        LET B = FNL(D,L,R)
```

where D, L, and R have some values in the program.

The number of arguments with which a user-defined function is called must agree with the number of arguments with which it is defined. For example:

```
100       DEF FNA(X) = X*3
110       PRINT FNA(3,2)
```

will cause an error message:

```
?Arguments don't match at line 110
```

In a DEF statement or function reference, where a function has zero arguments, the function name can be written with or without parentheses. For example:

```
Ready

OLD C3P332

Ready

LISTNH
10        DEF FNA=X^2
20        INPUT 'TYPE A NUMBER';X
30        PRINT FNA; FNA()
32767     END

Ready

RUNNH
TYPE A NUMBER? 3.65
 13.3225   13.3225

Ready
```

When calling a user-defined function, the parenthesized arguments can be any legal expressions. The value of each expression is substituted for the corresponding function variable. For example:

```
LISTNH
10       DEF FNZ(X)=X^2
20       LET A=2
         \PRINT FNZ(2+A)
32767    END

Ready

RUNNH
 16

Ready
```

If the same function name is defined more than once, an error message is printed.

```
100      DEF FNX(X)=X^2
110      DEF FNX(X)=X^4
?Illegal FN redefinition at line 110

Ready
```

The function variable need not appear in the function expression as shown below:

```
LISTNH
100      ! FUNCTION VARIABLE NOT IN FUNCTION EXPRESSION
110      DEF FNA(X) = 4*A +2
120      FOR A = 0 TO 3
130      LET R = FNA(10) + 1
         \ PRINT R
         \ NEXT A
32767    END

Ready

RUNNH
 3
 7
 11
 15

Ready
```

The program in Figure 3-3 contains examples of a multi-variable DEF statement in lines 30, 50, and 70.

3.8 SUBROUTINES

When a particular mathematical expression is evaluated several times throughout a program, the DEF statement enables the user to write that expression only once. The technique of looping allows the program to do a sequence of instructions a specified number of times. If the program should require that a sequence of instructions be executed several times in the course of the program, this is also possible.

```
LISTNH
10      REM -- MODULUS ARITHMETIC PROGRAM
                THIS PROGRAM CREATES ADDITION AND
                MULTIPLICATION TABLES FOR A SPECIFIED
                MODULUS M.
20      REM -- FUNCTION TO FIND X MOD M
30      DEF FNM(X,M)=X-M*INT(X/M)
        !
40      REM -- FUNCTION TO FIND A+B MOD M
50      DEF FNA(A,B,M)=FNM(A+B,M)
        !
60      REM -- FUNCTION TO FIND A*B MOD M
70      DEF FNB(A,B,M)=FNM(A*B,M)
100     PRINT \ PRINT 'ADDITION AND MULTIPLICATION TABLES MOD M'
        \ INPUT 'WHAT MODULUS';M
        \ PRINT \ PRINT 'ADDITION TABLES MOD';M
        \ GOSUB 800
110     FOR I=0 TO M-1
120     PRINT I;'  ';
        \ FOR J=0 TO M-1
        \ PRINT FNA(I,J,M);
        \ NEXT J
        \ PRINT
        \ NEXT I
        \ PRINT \ PRINT
200     PRINT 'MULTIPLICATION TABLES MOD';M
        \ GOSUB 800
210     FOR I=0 TO M-1
        \PRINT I;'  ';
        \FOR J=0 TO M-1
        \PRINT FNB(I,J,M);
        \NEXT J
        \PRINT
        \NEXT I
220     GOTO 32767
800     REM -- THIS SUBROUTINE PRINTS THE HEADINGS FOR EACH TABLE
810     PRINT
        \ PRINT TAB(4);
820     FOR I=0 TO M-1
        \PRINT I;
        \NEXT I
        \PRINT
830     FOR I=1 TO 3*M+4
        \ PRINT '-';
        \ NEXT I
        \ PRINT
        \ RETURN
32767   END

Ready
```

Figure 3-3 Modulus Arithmetic

```
RUNNH

ADDITION AND MULTIPLICATION TABLES MOD M
WHAT MODULUS? 7

ADDITION TABLES MOD 7

        0   1   2   3   4   5   6
-------------------------------------

 0      0   1   2   3   4   5   6
 1      1   2   3   4   5   6   0
 2      2   3   4   5   6   0   1
 3      3   4   5   6   0   1   2
 4      4   5   6   0   1   2   3
 5      5   6   0   1   2   3   4
 6      6   0   1   2   3   4   5

MULTIPLICATION TABLES MOD 7

        0   1   2   3   4   5   6
-------------------------------------

 0      0   0   0   0   0   0   0
 1      0   1   2   3   4   5   6
 2      0   2   4   6   1   3   5
 3      0   3   6   2   5   1   4
 4      0   4   1   5   2   6   3
 5      0   5   3   1   6   4   2
 6      0   6   5   4   3   2   1

Ready
```

Figure 3-3 (Cont.) Modulus Arithmetic

A subroutine is a section of code performing some operation required at more than one point in the program. Sometimes a complicated I/O operation for a volume of data, a mathematical evaluation which is too complex for a user-defined function, or any number of other processes may be best performed in a subroutine.

More than one subroutine can be used in a single program, in which case they can be placed one after another at the end of the program (in line number sequence). A useful practice is to assign distinctive line numbers to subroutines; for example, if the main program uses line numbers up to 199, use 200 and 300 as the first numbers of two subroutines. Consider the following example:

```
LISTNH
10       REM -- THIS PROGRAM ILLUSTRATES GOSUB AND RETURN
20       DEF FNA(X)=ABS(INT(X))
30       PRINT 'THIS SUBROUTINE FINDS QUADRATIC SOLUTIONS'
40       INPUT 'ENTER THREE COEFFICIENTS'; A,B,C
         \ PRINT
         \ PRINT 'SOLUTIONS FOR COEFFICIENTS ENTERED ARE '
         \  PRINT 'SHOWN FIRST, THEN SOLUTIONS FOR'
         \ PRINT 'ABSOLUTE VALUE COEFFICIENTS.'
50       GOSUB 500
60       LET A=FNA(A)
         \    B=FNA(B)
         \    C=FNA(C)
70       PRINT
         \PRINT 'WITH ALL COEFFICIENTS CONVERTED TO ABSOLUTE:'
80       GOSUB 500
90       PRINT 'TYPE "MORE" IF YOU WANT TO CONTINUE'
100      INPUT A$
         \ IF A$ = "MORE" THEN 40
110      GO TO 32767
500      REM -- THIS SUBROUTINE PRINTS OUT SOLUTIONS OF THE
         EQUATION A*X^2+B*X+C=0
510      PRINT
         \PRINT 'THE EQUATION IS ';A;'*X^2 + ';B;' *X + ';C;' =
520      LET D=B*B-4*A*C
         \ IF D<>0 THEN 600
530      PRINT 'ONE SOLUTION: X=';-B/(2*A)
         \ RETURN
600      IF D<0 THEN 700
605      PRINT 'TWO SOLUTIONS: X=(';
610      PRINT (-B+SQR(D))/(2*A);') AND (';(-B-SQR(D))/(2*A);')'
         \ RETURN
700      PRINT 'IMAGINARY SOLUTION: X=(';
710      PRINT -B/(2*A);'+'; SQR(-D)/(2*A);'I)'
720      PRINT '     AND (';
730      PRINT -B/(2*A);'-';SQR(-D)/(2*A);'I)'
         \ RETURN
32767    END

Ready
```

```
RUNNH
THIS SUBROUTINE FINDS QUADRATIC SOLUTIONS
ENTER THREE COEFFICIENTS? 1,-5,-6

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

THE EQUATION IS  1 *X^2 + -5  *X + -6  = 0
TWO SOLUTIONS: X=( 6 ) AND (-1 )

WITH ALL COEFFICIENTS CONVERTED TO ABSOLUTE:

THE EQUATION IS  1 *X^2 +  5  *X +  6  = 0
TWO SOLUTIONS: X=(-2 ) AND (-3 )
TYPE "MORE" IF YOU WANT TO CONTINUE
? MORE
ENTER THREE COEFFICIENTS? 2,3,-2

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

THE EQUATION IS  2 *X^2 +  3  *X + -2  = 0
TWO SOLUTIONS: X=( .5 ) AND (-2 )

WITH ALL COEFFICIENTS CONVERTED TO ABSOLUTE:

THE EQUATION IS  2 *X^2 +  3  *X +  2  = 0
IMAGINARY SOLUTION: X=(-.75 + .661438 I)
    AND (-.75 - .661438 I)
TYPE "MORE" IF YOU WANT TO CONTINUE
? NO

Ready
```

Lines 500 through 730 constitute the subroutine. The subroutine is executed from line 50 and again from line 80.

3.8.1 GOSUB Statement

Subroutines usually are placed physically at the end of a program before DATA statements, if any, and always before the END statement. The program begins execution and continues until it encounters a GOSUB statement of the form:

GOSUB <line number>

where the line number following the word GOSUB is the first line number of the subroutine. Control then transfers to that line in the subroutine. For example:

```
50        GOSUB 200
```

Control is transferred to line 200 in the user program. The first line in the subroutine can be a remark or any executable statement.

3.8.2 RETURN Statement

Having reached the line containing a GOSUB statement, control transfers to the line indicated after GOSUB; the subroutine is processed until the computer encounters a RETURN statement of the form:

```
RETURN
```

which causes control to return to the statement following the original GOSUB statement. Subroutine exit is always through a RETURN statement.

Before transferring to the subroutine, BASIC-PLUS internally records the next sequential statement to be processed after the GOSUB statement; the RETURN statement is a signal to transfer control to this statement. In this way, no matter how many subroutines or how many times they are called, BASIC-PLUS always knows where to go next.

3.8.3 Nesting Subroutines

Subroutines can be nested; that is, one subroutine can call another subroutine. If the execution of a subroutine encounters a RETURN statement, it returns control to the line following the GOSUB which called that subroutine. Therefore, a subroutine can call a subroutine, including itself. Subroutines can be entered at any point and can have more than one RETURN statement. It is possible to transfer to the beginning or to any part of a subroutine; multiple entry points and returns make a subroutine more versatile.

The maximum level of GOSUB nesting is dependent on the size of the user program and the amount of memory available at the installation. Exceeding this limit causes the message:

```
?MAXIMUM MEMORY EXCEEDED AT LINE line number
```

3.9 STOP AND END STATEMENTS

The STOP and END statements are used to terminate program execution. The END statement is the last statement in a BASIC-PLUS program. The STOP statement can occur several times throughout a single program with conditional jumps determining the actual end of the program. The END statement is of the form:

```
END
```

The line number of the END statement should be the largest line number in the program, since any lines with line numbers greater than that of the END statement are not retrieved by a subsequent OLD command after the program is saved. The user can habitually number each END statement 32767 to avoid this problem.

A program will execute without an END statement. However if a program retrieved by an OLD command includes no END statement, RSTS/E prints the following informative message:

```
?End of file on device
```

The STOP statement is of the form:

```
line number STOP
```

and causes:

```
STOP AT LINE line number
READY
```

to be printed when executed. A CONTINUE command entered at this point resumes execution at the statement following STOP.

Execution of a STOP or END statement causes the message:

```
        Ready
```

to be printed by the teleprinter. This signals that the execution of a program has been terminated or completed, and BASIC-PLUS is able to accept further input. The execution of an END statement also closes all files in a BASIC-PLUS program.

PART II

BASIC-PLUS ADVANCED FEATURES

Part II describes certain features of BASIC-PLUS that give the language flexibility for a greater variety of applications. Additional capabilities of the statements previously described are included, along with new statements, character string manipulation, string arithmetic (an optional feature), integer mode variables and arithmetic, and intrinsic matrix functions (an optional feature). Also described in the immediate mode of operation which causes BASIC-PLUS to treat single statements as commands.

In general, the techniques presented here allow the user to write programs which conserve memory space and reduce execution time. With the ability to manipulate character strings, the user can write sophisticated programs to handle a wide range of data. Also, the ability to perform arithmetic with numeric string data enables the user to obtain greater precision than is possible with floating point and integer operands.

The matrix functions allow the user to perform matrix I/O and the matrix operations of addition, subtraction, multiplication, inversion and transposition.

CHAPTER 4
IMMEDIATE MODE OPERATIONS

4.1 USE OF IMMEDIATE MODE FOR STATEMENT EXECUTION

It is not necessary to write a complete program to use BASIC-PLUS. Many BASIC-PLUS statements are executable as on-line commands, i.e., executed immediately by the BASIC-PLUS processor. This facility provides a powerful on-line desk calculator. It allows the user to check and change variables of the current program, or perform calculations independent of the current program. Thus it is a helpful tool for program design and debugging.

BASIC-PLUS distinguishes between lines entered for later execution and those entered for immediate execution solely on the presence (or absence) of a line number. Statements which begin with line numbers are stored; statements without line numbers are compiled and executed immediately upon being entered to the system. Thus the line:

```
10        PRINT 'THIS IS AN EXECUTABLE BASIC-PLUS PROGRAM'
```

produces no action at the terminal upon entry, while the statement:

```
PRINT 'THIS IS AN IMMEDIATE MODE STATEMENT'
```

when entered causes the following immediate output:

```
THIS IS AN IMMEDIATE MODE STATEMENT

Ready
```

The READY message is then printed to indicate the system readiness for further input.

Although only one statement at a time can be executed in immediate mode, statements typed in immediate mode can reference variable values established either by the running of the current program or by other immediate mode statements. For example:

```
A=3

Ready

B=4

Ready

PRINT ATN(A/B)
 .643501

Ready

PRINT SQR(A^2 + B^2)
 5

Ready
```

4.2 PROGRAM DEBUGGING

Immediate mode operation is especially useful in two areas: program debugging and the performance of simple calculations in situations which do not occur with sufficient frequency or with sufficient complications to justify writing a program.

In order to facilitate debugging a program, the user can place STOP statements liberally throughout the program, Each STOP statement causes the program to halt, printing the line number at which the STOP occurred. The user then can examine various data values, perhaps change them in immediate mode, and then give the CONT command to continue program execution. However, a syntax error in immediate mode or one of several other conditions could prevent continuation of program execution with the CONT command.

When using immediate mode, nearly all the standard statements can be used to generate or print results.

The user can also halt program execution at any time by typing CTRL/C. Immediate mode can then be used to examine and/or change data values. Typing the CONT command resumes program execution.

However, if the CONT command is entered and execution cannot be resumed, the following message is printed.

```
?Can't CONTinue

Ready
```

When a program is interrupted by typing the CTRL/C combination, the integer variable LINE contains the line number of the statement being executed when the interrupt occurred. The PRINT command is used to display the contents of LINE.

```
^C

Ready

PRINT LINE
 300

Ready
```

4.3 MULTIPLE STATEMENTS PER LINE

Multiple statements cannot be used on a single line in immediate mode. For example:

```
A=1 \ PRINT A
?Illegal in immediate mode
```

The use of the FOR modifier (and all other modifiers described in Section 8.7) is allowed. Thus a table of square roots can be produced as follows:

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

Ready
```

4.4 RESTRICTIONS ON IMMEDIATE MODE

Certain commands make no logical sense when used in immediate mode. Commands in this category include:

DEF
FNEND
DIM
DATA
FOR
NEXT

When any of these is given, the message ILLEGAL IN IMMEDIATE MODE is printed.

CHAPTER 5
CHARACTER STRINGS

5.1 CHARACTER STRINGS

The previous chapters describe the manipulation of numerical information; however, BASIC-PLUS also processes information in the form of character strings. A string, in this context, is a sequence of characters treated as a unit. A string can be composed of any combination of ASCII characters.

In Chapter 3 the INPUT and PRINT statements were shown printing messages along with the input and output of numeric values (see lines 10 and 15 above). These messages consist of character string constants (just as 4 is a numeric constant). In a similar way, there are character string variables and functions.

5.1.1 String Constants

Just as numbers can be used as constants or referenced by variable names, BASIC-PLUS allows for character string constants. Character string constants are delimited by either single or double quotes. For example:

```
100     LET Y$ = "FILE4"
200     B1$ = 'CAN'
300     IF A$ = "YES" GO TO 250
```

where "FILE4", 'CAN' and "YES" are character string constants.

5.1.2 Character String Variables

Variable names can be introduced for simple strings and for both lists and matrices composed of strings (which is to say 1- and 2-dimensional string matrices). Any legal name followed by a dollar sign ($) character is a legal name for a string variable. (The rules for forming legal variable names are described in Section 2.5.2.) For example:

```
A$
C7$
NAME.OF.CUSTOMER$       (EXTEND mode only)
```

are simple string variables. Any list or matrix variable name followed by the $ character denotes the string form of that variable. For example:

```
V$(N%)        M2$(N%)
CS(M%,N%)     G1$(M%,N%)
```

(where M and N indicate the position of that element of the matrix within the whole) are list and matrix string variables.

The same name in combination with various prefixes and suffixes can appear in the same program, and generate mutually independent variables. For example, the name A refers to a floating-point variable A. The name A can be used as follows:

A	floating point variable A
A%	integer variable A%
A$	string variable A$
A(d)	floating point array A with dimension specification d
A%(d)	integer array A% with dimension specification d
A$(d)	string array A$ with dimension specification d
FNA	floating point function FNA
FNA%	integer function FNA%
FNA$	string function FNA$

Thus any name can be used to reference up to nine distinct data entities. In the case of an array, there can be only one dimension specification in the program for each A(d), A%(d), or A$(d). For example, the following combination is legal.

```
10        DIM A%(100), A(200)
20        A%=50%
30        A%(25)=100%
40        A%(A%)=200%
```

It will place a value of 50 in the integer variable A%, 100 in element 25 of array A%, and 200 in element 50 of array A%. Array A is not affected. On the other hand, the following DIM statement is illegal.

```
250       DIM A$(20), A$(10,4)
?Redimensioned array at line 250
```

because it attempts to use the same string array name to identify two arrays. There would be no logical conflict if one instance of A$ were A or A%.

Just as numeric variables are automatically initialized to 0 when a program is run, string variables are initialized to a null string containing no characters (the character string constant "").

5.1.3 Subscripted String Variables

String lists and matrices are defined with the DIM statement, as are numerical lists and matrices. For example:

```
100       DIM S1$(5)
```

indicates the S1$ is a string matrix with six elements, S1$(0) through S1$(5), which can be separately accessed. If a DIM statement is not used, a subscripted string variable is assumed to have a dimension of 10 (11 elements including the zero element) in each direction. Note that the dimension of a string matrix specifies the number of strings and not the number of characters in any one string. For example, if the first statements in a program are:

```
1050      FOR I=1 TO 7
          \ LET B$(I)="PDP-11"
          \ NEXT I
```

a list B$(n) is created having 11 accessible elements, B$(0) through B$(10). The elements B$(1) through B$(7) are set equal to "PDP-11" and the others would be null strings (have no characters), as shown below.

```
LISTNH
1050      FOR I=1 TO 7
          \ LET B$(I)="PDP-11"
          \ NEXT I
1060      FOR H=0 TO 10
          \ PRINT H,B$(H)
          \ NEXT H
32767     END

Ready

RUNNH
 0
 1                 PDP-11
 2                 PDP-11
 3                 PDP-11
 4                 PDP-11
 5                 PDP-11
 6                 PDP-11
 7                 PDP-11
 8
 9
 10

Ready
```

As a general rule, all lists should be dimensioned to the maximum size referenced in the program.

5.1.4 String Size

A character string can contain any number of characters limited only by the amount of memory available. However, the LINE FEED key cannot be used to type a string on two or more terminal lines. To create a string longer than a terminal line, it is necessary to use string operations (concatenation, for instance) described later. Since memory is limited, strings can also be saved in files on the system disk.

5.1.5 Relational Operations

When applied to string operands, the relational operators indicate alphabetic sequence. For example:

```
55        IF A$(I) < A$(I+1) GOTO 100
```

When line 55 is executed the following occurs: A$(I) and A$(I+1) are compared; if A$(I) occurs earlier in alphabetical order than A$I+1), execution continues at line 100. Table 5-1 contains a list of the relational operators and their string interpretations.

In any string comparison (except ==), trailing spaces are ignored. The string "YES" is equivalent to "YES ".

When two strings of unequal length are compared, the shorter string (of length n) is compared with the first n characters of the longer string. If this comparison is not equal, that inequality is the result of the original comparison. If the first n characters of the strings are the same, the strings are equal if the excess characters in the longer string are all blanks. Otherwise, the longer string is greater than the shorter string.

A null string (of length zero) is less than any string of length greater than zero unless that string consists of all blanks in which case the two strings are equivalent.

Table 5-1
Relational Operators Used with String Variables

Operator	Example	Meaning
=	A$ = B$	The strings A$ and B$ are equivalent, except for possible trailing spaces.
<	A$ < B$	The string A$ occurs before B$ in collating sequence.
<=	A$ <= B$	The string A$ is equivalent to or occurs before B$ in collating sequence.
>	A$ > B$	The string A$ occurs after B$ in collating sequence.
>=	A$ >= B$	The string A$ is equivalent to or occurs after B$ in collating sequence.
<>	A$ <> B$	The strings A$ and B$ are not equivalent.
==	A$ == B$	The strings A$ and B$ are identical (i.e., same length, composed of the same characters in the same order).

5.2 ASCII STRING CONVERSIONS, CHANGE STATEMENT

Individual characters in a string can be referenced through use of the CHANGE statement. The CHANGE statement permits the user program to transform a character string into a list of numeric values or a list of numeric values into a character string. Each character in a string can be converted to its ASCII equivalent or vice versa.

Appendix D, Section D.2 describes the relationship between the ASCII characters and their corresponding decimal values. Note that several ASCII characters have no graphic equivalent; that is, they cause no character to be printed.

As an illustration, consider the following:

```
LISTNH
1000     REM -- STRING/ASCII CHANGE DEMO
1010     DIM X(3)
1020     LET A$ = "CAT"
1030     CHANGE A$ TO X
1040     X=X(0)+X(3)
         \ PRINT 'X=';X,'THE ARRAY X IS ';
                  X(0);X(1);X(2);X(3)
1050     !IN A CHANGE STATEMENT THE NUMERIC MATRIX
         !IS REFERENCED WITHOUT SUBSCRIPTS.  AS THIS
         !EXAMPLE SHOWS, HAVING A SINGLE-VALUE VARIABLE
         !WITH THE SAME NAME IN THE PROGRAM CAUSES
         !NO AMBIGUITY.
32767    END

Ready

RUNNH
X= 87          THE ARRAY X IS  3  67  65  84

Ready
```

X(1) through X(3) take on the ASCII values of the characters in the string variable A$. The first element of X, X(0), becomes the number of characters present in A$.

If more characters are present in the string variable than can be accommodated in the numeric list, the message SUBSCRIPT OUT OF RANGE is printed. The first element of the list becomes the number of characters in the entire string and is greater than the dimension of the list.

Notice that line 1010, above, created a 4-element array, X. A DIM statement must be used in this instance; otherwise, the system creates a default 121-element array, possibly causing unexpected results.

Another program which transforms a character string into a list of numeric values is shown below:

```
LISTNH
100     DIM A(65)
110     READ A$
120     CHANGE A$ TO A
130     FOR I=0  TO A(0)
        \ !A(0) GIVES LENGTH OF STRING
140     PRINT A(I);
        \ NEXT I
200     DATA ABCDEFGHIJKLMNOPQRSTUVWXYZ
32767   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
Ready
```

Notice that A(0) = 26

To change numbers into string characters, the CHANGE statement is used as follows:

```
LISTNH
100     REM --- ASCII NUMBER TO STRING CONVERSION DEMO
110     FOR I=0 TO 5
        \ READ A(I)
        \ NEXT I
120     CHANGE A TO A$
        \ PRINT A$
200 DATA 6,65,66,67,68,69,70
32767   END

Ready

RUNNH
ABCDE

Ready
```

This program prints ABCDE because the numbers 65 through 69 are the code numbers for A through E.

Before CHANGE is used in the matrix-to-string direction, the programmer must indicate the number of characters in the string as the zero element of the matrix. If element 0 has a value of zero, the CHANGE statement generates a zero-length string.

5.3 STRING INPUT

The READ, DATA and INPUT statements can be used to input string variables to a program. For example:

```
10        READ A$,B,C,D
150       DATA 17,14,13.4,CAT
```

causes the following assignments to be made:

A$ = the character string "17"
B = 14
C = 13.4
reading D as CAT causes the message ILLEGAL NUMBER AT LINE 10 to be printed.

Quotation marks are necessary around a string item in a DATA statement only when the string contains a comma, or when leading, trailing or embedded spaces within the string are significant, or when lower-case letters are to be preserved. Quotes (single or double) are always acceptable around string items, even though not always necessary. For example, the items in line 500 in the following program are all acceptable character strings.

```
LISTNH
100       READ A$,B$,C$,D$,E$
110       PRINT A$; B$; C$; D$; E$
120       PRINT A$,B$,C$,D$,E$
500       DATA 'MR. JONES', MISS SMITH, "MRS.BROWN", 'MISS','"MR"'
32767     END

Ready

RUNNH
MR. JONESMISSSMITHMRS.BROWNMISS"MR"
MR. JONES      MISSSMITH      MRS.BROWN      MISS           "MR"

Ready
```

Although the string MISS SMITH is acceptable without quotes, embedded spaces in the string are discarded. A READ statement can appear anywhere in a multiple statement line, but a DATA statement must be the last statement on a line. See also the MAT READ statement which reads matrices (either numeric or string), Section 7.2.

NOTE

The data pool composed of values from the programmed DATA statements is stored internally as an ASCII string list. Where a numeric variable is read, the appropriate ASCII to numeric conversions are performed. Where a string variable is read, the string is used as it appears in the DATA statement. If the item did not appear in quotes, leading, trailing and embedded spaces are ignored. If the item did appear in quotes, the string variable is equated to the entire string within the quotes.

The input statement is used to input character strings exactly as though accepting numeric values. For example:

```
100        INPUT 'YOUR NAME';N$;'YOUR AGE';A
```

is functionally equivalent to:

```
100        PRINT 'YOUR NAME';
110        INPUT N$
120        PRINT 'YOUR AGE';
130        INPUT A
```

5.3.1 Line Input

Another feature of the INPUT statement when used with character string input is the INPUT LINE statement of the form:

INPUT LINE <string variable>

For example:

```
INPUT LINE A$
```

which causes the program to accept a line of input from the terminal with embedded spaces, punctuation characters, or quotes. Any characters are acceptable in a line being input to the program in this manner. The program can then treat the line as a whole or in smaller segments as explained in Section 5.5 which describes string functions.

No text string can be output with the INPUT LINE statement; this facility is only available in the INPUT statement. For example:

```
10         INPUT LINE 'TEXT';A$
?Syntax error at line 10

Ready
```

An INPUT LINE statement reads the entire line as typed by the user, including the line terminating character. The line terminator is one of the following:

1. Carriage return/line feed, generated by typing the RETURN key (appends the ASCII values 13 and 10 to the character string);
2. Line feed, generated by typing the LINE FEED key (appends the ASCII values 10, 13, and 0 to the character string); or
3. ESCAPE, generated by typing the ESCAPE, ALT MODE or PREFIX key, depending upon the terminal (appends an ASCII 27 to the character string).
4. Form feed, generated by typing CTRL/L (appends an ASCII 12 to the character string).

Upon receipt of an INPUT or INPUT LINE statement, BASIC-PLUS issues a prompt symbol in the form of question mark and a space to signal that the next line typed will be treated as input. Depending on the present loading of the system, additional prompts may appear while entering long string input. These prompts are not included in the input.

After 256 characters are entered, the error message, LINE TOO LONG, is printed. When excessive input is entered to cause this message, the string variable will have unpredictable contents. For example:

```
LISTNH
100     PRINT 'ENTER LONG STRING'
        \ INPUT LINE A$
        \ PRINT
        \ PRINT A$
32767   END

Ready

RUNNH
ENTER LONG STRING
? ,./[';;']":]\POC\OIJKMNHFFFFFFFFFFFFFFFFFFF5555555566666777788890111122
233344RRFFCDSSWWEERGNNM,,KKLL:::::DDDD#####****..........+++++UUUIKOLBBBB
BBBFFFFFFFFFFFFFFFFRRRRRRRRR44444445555556? 6666677777788888899999IIIIIIIUUU
UYYYYFFFFFFFVVVVVBBBBNNNNNMMMVV? VVCCCSSSSAAAAQQQSSSSWWWEEDDD4444444444444
4444%%%%%%%%^^^^^^&&&&&&$$$$$$######@@@@!!!!!HHHHH666666777777778888?Line t
oo long at line 100
? HH

44444444444444444%%%%%%%%^^^^^^&&&&&$$$$$$#####@@@@!!!!!HHHHHHHH


Ready
```

The GET statement can also be used to input string information from a terminal in the form of I/O records. See Section 12.3 for further information.

5.4 STRING OUTPUT

When string values are included in a PRINT statement (either alone or in combination with other string and numeric values), no leading or trailing spaces are added to the string. Only the actual content of the string is printed, and leading or trailing spaces must be included in the string when it is defined. For example:

```
LISTNH
100     REM --  STRING OUTPUT DEMO
200     LET X=1.0
        \ Y=2.01
        \ A$="A="
210     PRINT A$;X" B="Y
220     PRINT 'DONE'
32767   END

Ready

RUNNH
A= 1  B= 2.01
DONE

Ready
```

Semicolons separating character string constants from other list items are optional. For example, in line 210 (above) note that the variable Y is not separated from the character string " B=" by a semicolon.

Character string output can also contain the string functions described in Section 5.5.

5.5 STRING FUNCTIONS

Besides intrinsic mathematical functions (e.g., SIN, LOG), BASIC-PLUS contains various functions for use with character strings. These functions allow the program to perform arithmetic operations with numeric strings, concatenate two strings, access part of a string, determine the number of characters in a string, generate a character string corresponding to a given number or vice versa, search for a substring within a larger string, and perform other useful operations. (These functions are particularly useful when dealing with whole lines of alphanumeric information input by a INPUT LINE statement.) The various functions available are summarized in Table 5-2.

Table 5-2
String Functions

Function Code	Meaning

LEFT(A$,N%)	Indicates a substring of the string A$ from the first character through the Nth character (the leftmost N characters of the string A$). For example:
	`10        A$='ABCDEFGHIJKLMNOPQRSTUVWXYZ'`
	`RUNNH`
	
	`Ready`
	
	`PRINT LEFT(A$,7%)`
	`ABCDEFG`
	
	`Ready`

RIGHT(A$,N%)	Indicates a substring of the string A$ from the Nth character through the last character in A$ (the rightmost characters of the string A$ starting with the Nth character). For example:
	`PRINT RIGHT(A$,20%)`
	`TUVWXYZ`
	
	`Ready`

MID(A$,N1%,N2%)	Indicates a substring of the string A$ starting with character N1, and N2 characters long (the characters between and including the N1 through N1+N2-1 characters of the string A$). For example:
	`PRINT MID(A$,15%,5%)`
	`OPQRS`
	
	`Ready`

LEN(A$)	Returns an integer that indicates the number of characters in the string A$ (including trailing blanks). For example:
	`PRINT LEN(A$)`
	` 26`
	
	`Ready`

Table 5-2
String Functions (Cont.)

Function Code	Meaning
+	Indicates a concatenation operation on two strings. For example "ABC"+"DEF" is equivalent to "ABCDEF". "12"+"34"+"56" is equivalent to "123456".
CHR$(N%)	Generates a 1-character string having the ASCII value of N (see Table 5-2). For example, CHR$(65) is equivalent to "A". Only one character can be generated.
ASCII(A$)	Generates an integer that is the ASCII decimal value of the first character in A$. For example, ASCII("X") is equivalent to 88, the ASCII equivalent of X. If B$ = "XAB", then ASCII(B$) = 88.
DATE$(N%)	Where N%=0, this function returns the current date in the form: <day>-<month>-<year> For example: 12-AUG-72 This quantity can be printed on output by simple reference to the function. It should be noted that dates are output using both upper and lower case letters. When the output device is not capable of generating lower case letters, the ASCII values still imply lower case. Where N%<>0 the function translates N% into a date string. (See Section 8.8.) If the run-time system was generated with the numeric data option, 12-Aug-72 is returned as 72.08.12.

INSTR(N1%,A$,B$)	Indicates a search for the substring B$ within the string A$ beginning at character position N1. Returns a value of 0 if B$ is not in A$. Returns the character position of B$ if B$ is found to be in A$ (character position is measured from the start of the string with the first character counted as character 1). For example:
	
	`PRINT INSTR(5%,A$,'OP')`
	` 15`
	
	`Ready`
	
	If B$ is a null string (B$ = ""), the INSTR function returns the value 1. The null string is a proper substring of any string and is treated conventionally as the first element of A$ in null string search operations. In addition, if both A$ and B$ are null strings, the INSTR function returns the value 1.

SPACE$(N%)	Indicates a string of N spaces, used to insert spaces within a character string.

NUM$(N)	Indicates a string of numeric characters representing the value of N as it would be output by a PRINT statement. (NUM$(n)=(space)n(space) if n>0 and NUM$(n)=-n(space) if n<0. For example:
	
	`PRINT NUM$(7465098702134)`
	` .74651E 13`
	
	`Ready`

Table 5-2
String Functions (Cont.)

Function Code	Meaning

NUM1$(N)	Yields a string of numeric characters numerically equal (or approximately equal) to the integer or floating point value N. This is similar to the NUM$ function, except that no spaces nor E-format results are returned. It may be used to convert an integer or floating point value for use as a string function operand. For example:
	`PRINT NUM1$(PI)`
	`3.14159265358979`
	`Ready`
	`PRINT NUM1$(97.5*30456.23+3.03^5.1)`
	`2969767.76154965`
	`Ready`

VAL(A$)	Computes the numeric value of the string of numeric characters A$ (may include digits, +, -, . and E) and returns it as a floating point number. If A$ contains any characters not acceptable as numeric input with the INPUT statement, an error results. For example:
	`PRINT VAL('14.3E-5')`
	` .000143`
	`Ready`

TIME$(N%)	Where N%=0, this function returns the current time of day as a string. For example: 01:30 PM Where N%<>0, the function translates N% into a time string (See Section 8.8). If the run-time system was generated using the 24-hour time option, 01:30 PM is returned as 13:30 followed by 3 spaces. TIME$ always returns an 8-character string.

STRING$(N1%,N2%)	Creates a string of length N1% and characters whose ASCII decimal value is N2%. For example, to create a string Y$ composed of 10 space (blank) characters CHR$(32%), execute the following statement:
	`Y$ = STRING$(10,32)`
	`Ready`
	See Section D.2 for the decimal values of ASCII characters.

Table 5-2
String Functions (Cont.)

Function Code	Meaning
CVT$$(S$,M%)	Converts the source character string $$ according to the decimal value of the integer M%. The bits of M% are interpreted as follows. 1% Trim the parity bit. 2% Discard all spaces and tabs. 4% Discard all carriage return (CR) line feed (LF), form feed (FF), escape (ESC), rubout (DEL), and fill or null (NUL) characters. 8% Discard leading spaces and tabs. 16% Reduce spaces and tabs to one space. 32% Convert lower case to upper case. 64% Convert square brackets to parentheses; i.e., [to (and] to). 128% Discard trailing spaces and tabs. 256% Disallow alteration of the string except parity bit trimming. These bits can be used in combination. If M% is given as 21%, for example, the result is the same as if the user had used three CVT$$ functions with M% values of 1%, 4%, and 16%. This function is described in detail in Section 12.5
XLATE(S$,T$)	Translates source string S$ from its existing storage code to a code indicated by the table string T$, and returns the translated form of string S$ as the target string. For a complete description of this function, see Section 12.7. **NOTE** The following functions are for use in connection with the string arithmetic feature. The arguments A$ and B$ can be either string variable names, string expressions, or string constants and should consist entirely of numeric characters with an optional decimal point. See Section 5.6 for further details on string arithmetic.

SUM$(A$,B$)	Yields the arithmetic sum A$ + B$ of the numeric strings A$ and B$. For example:
	`PRINT S1$`
	`12349.1789`
	
	`Ready`
	
	`PRINT SUM$(S1$,'89.4545454545')`
	`12438.6334454545`
	
	`Ready`

Table 5-2
String Functions (Cont.)

Function Code	Meaning

DIF$(A$,B$)	Yields the arithmetic difference, A$-B$, of numeric strings A$ and B$. For example:
	`PRINT B$`
	`9876.54321`
	
	`Ready`
	
	`PRINT DIF$(B$,'78.89')`
	`9797.65321`
	
	`Ready`

PROD$(A$,B$,P%)	Yields the product, A$ times B$, with rounding to P% places. For example:
	`PRINT A$,B$`
	`12345.6789     9876.54321`
	
	`Ready`
	
	`PRINT PROD$(A$,B$,6%)`
	`121932631.112635`
	
	`Ready`

QUO$(A$,B$,P%)	Yields the quotient, A$ divided by B$, with rounding to P% places. For example:
	`PRINT C$`
	`3.5`
	
	`Ready`
	
	`V9$=QUO$(C$,'1.7777',3%)`
	
	`Ready`
	
	`PRINT V9$`
	`1.969`
	
	`Ready`

Table 5-2
String Functions (Cont.)

Function Code	Meaning

PLACE$(A$,P%)	Rounds A$ to P% places. For example:
	`A1$=PLACE$(A$,INT(3.7))`
	`Ready`
	`PRINT A1$`
	`12345.679`
	`Ready`

COMP%(A$,B$)	Yields a truth value based on the result of a numeric comparison, as follows:
	-1 if A$ < B$
	0 if A$ = B$
	1 if A$ > B$
	For example:
	`T%=COMP%(A$,A1$)`
	`Ready`
	`PRINT T%`
	`-1`
	`Ready`
	`PRINT COMP%(A1$,A$)`
	` 1`
	`Ready`

5.5.1 User-defined String Functions

Character string functions can be written in the same way as numeric functions. (See Sections 3.7.3 and 8.1.) The function is indicated as being a string function by the $ character after the function name.

User-defined string functions return character string values, although both numeric and string values can be used as arguments to the function. For example, the following multiple-line function (see Section 8.1) returns the string which comes first in alphabetical order:

```
100     DEF FNF$(A$,B$)
        \ FNF$=A$
        \ IF A$>B$ THEN FNF$=B$
110     FNEND
```

The following function combines two strings into one string:

```
10        DEF FNC$(X$,Y$)=X$+Y$
```

Numbers cannot be used as arguments in a function where strings are expected or vice versa. Line 80 is unacceptable:

```
10        DEF FNA$(A$) = CHR$(LEN(A$)+1)


80        LET Z=FNA$(4)
```

The message:

```
?Arguments don't match at line 80
```

is printed.

The following code is a string function which returns the leftmost five characters from the sum of three arguments:

```
LISTNH
75        DEF FNA$(X,Y,Z)   = LEFT(NUM$(X+Y+Z),5)
80        PRINT FNA$(100,20,3)
32767     END

Ready

RUNNH
 123

Ready
```

NUM$(123) is a 5-character string, as follows:

"(space)123(space)"

5.6 STRING ARITHMETIC FEATURE

The optional string arithmetic feature comprises seven functions that treat numeric strings (i.e., strings consisting entirely of an optional leading sign, numeric characters, and an optional decimal point) as arithmetic operands. This feature offers greater arithmetic precision than floating point with large numbers or with fractions and eliminates the need for scaling. As with any string value, numeric string variable names must be suffixed with a dollar sign ($) character, and numeric string constants must be bounded by quotation marks (") or apostrophes ('). The seven string arithmetic functions are described in Table 5-2.

5.6.1 String Arithmetic Precision

The maximum size of a string arithmetic operand is 60 characters, including the sign and the decimal point.

P% is an integer expression. The value of P% determines the level of arithmetic precision in the result of a PROD$, QUO$, or PLACE$ function. P% can be positive or negative. A positive P% value less than 5000 rounds the result to P significant digits to the right of the decimal point. For example:

```
LISTNH
100       EXTEND
          !ALLOWS LONG VARIABLE NAMES
110       INPUT 'ENTER TWO NUMERIC STRINGS TO BE MULTIPLIED';
                  STRING.A$,STRING.B$
          \ INPUT 'TO HOW MANY DECIMAL PLACES';PNO%
120       PR$=PROD$(STRING.A$,STRING.B$,PNO%)
          \ PRINT 'ANSWER IS  ';PR$
32767     END

Ready

RUNNH
ENTER TWO NUMERIC STRINGS TO BE MULTIPLIED? 56453.346,'879.0004532'
TO HOW MANY DECIMAL PLACES? 12
ANSWER IS  49622516.7186564072

Ready

RUNNH
ENTER TWO NUMERIC STRINGS TO BE MULTIPLIED? .00009067543,0134.2340345
TO HOW MANY DECIMAL PLACES? 10
ANSWER IS  .0121717288

Ready
```

Negative P% causes the result to be effectively divided by 10^(P%) and rounded (P%) places to the left of the decimal point. For example:

```
100       PRINT 'ENTER A LARGE NUMBER'
          \INPUT A$
          \IF A$ = '0' THEN 150
110       LET B$ = SUM$(A$,B$)
          ! ACCUMULATE TOTAL OF INPUT STRINGS
120       GO TO 100
150       B$ = PLACE$(B$,-6%)
160       PRINT 'TOTAL IS APPROXIMATELY '; B$; ' MILLION'
32767     END

Ready
```

The following can be used in place of lines 150 and 160.

```
150       PRINT 'TOTAL IS APPROXIMATELY ';PLACE$(B$,-6%); ' MILLION'
```

The result is always rounded if P% is less than 5000.

To truncate a result instead of rounding, specify P% in the form:

P% + 10000

and the result will be truncated at positive P%.

5.6.2 Combining String Functions

String arithmetic functions can be nested (i.e., used as operands in other string arithmetic functions) to specify complex arithmetic or algebraic operations. For example, the following LET statement

```
X = A * B + C/D
```

might be written

```
X$ = SUM$(PROD$(A$,B$,10), QUO$(C$,D$,10))
```

in string arithmetic. The following statement

```
X$ = PROD$(A$,B$,10) + QUO$(C$,D$,10)
```

is legal, but concatenates the product and quotient strings instead of summing them.

CHAPTER 6
INTEGER AND FLOATING POINT OPERATIONS

Numbers on the system can be represented and manipulated in either integer, floating point, or string format. The implications of representing numbers in a certain format and the resultant benefits are described in this chapter. Certain operations involving integer numbers are more efficient if performed using a forced one-word integer format. The specification of a forced integer format and the possible integer operations are described in Section 6.1 through 6.6. The results of performing operations by mixing the formats are described in Section 6.7. Operations using standard floating-point arithmetic and floating-point scaled arithmetic are performed as described in Section 6.8.

6.1 INTEGER CONSTANTS AND VARIABLES

Normally, all numeric values (variables and constants) specified in a BASIC program are stored internally as floating-point numbers. If operations to be performed deal with integer numbers, significant economics in storage space can be achieved by use of the integer data type (which uses only one computer word per value). Integer arithmetic is also significantly faster than floating-point arithmetic. Integer variables (and constants) can assume values in the range -32768 to +32767.

A constant, variable or function can be specified as an integer by terminating its name with the % character. For example:

```
100%   A%    FNX%(Y)
  4%   A1%   FNL%(N%,L%)
```

The user is expected to indicate where an integer constant is to be generated by using the % character. Otherwise a floating-point value is normally produced.

When a floating-point value is assigned to an integer variable, the fractional portion of that number is lost. The number is not rounded to the nearest integer value. (A FIX function is performed rather than an INT function see Section 3.7.) For example:

```
A% = -1.1
```

or

```
A% = -1.9
```

causes A% to be assigned the value -1.

6.2 INTEGER ARITHMETIC

Arithmetic performed with integer variables is performed modulo 2^16. The number range -32,768 to +32,767 is treated as continuous, with the number after +32,767 equal to -32,768. Thus, 32767% + 2% = -32767% and so on.

Integer division forces truncation of any remainder: for example 5%/7%=0 and 199%/100%=1. Operations can be performed in which both integer and floating-point data are freely mixed. The result is stored in the format indicated as the resulting variable, for example:

```
125      LET X% = N% + FNA(R)*2
```

The result of the expression on the right is truncated to provide an integer value for X%. The result of mixing integer and floating-point data is explained in Section 6.7.

Where program size is critical, the use of the % character to generate integer values is encouraged, as it uses significantly less storage space. For example:

```
120      FOR I%=1% TO 10%
```

takes less storage space and executes faster than:

```
120      FOR I=1 TO 10
```

6.3 INTEGER I/O

Input and output of integer variables is performed in exactly the same manner as operations on floating-point variables. (Remember that in cases where a floating-point variable has an integer value it is automatically printed as an integer but is still stored internally as a floating-point number and hence takes more storage space.) It is illegal to provide a floating-point value for an integer variable through either a READ or INPUT statement. For example:

```
LISTNH
110      READ A,B%,C,D%,E
         \ PRINT A,B%,C,D%,E
600      DATA 2.7,3,4,5.7,6.8

Ready

RUNNH
%Data format error at line 110

Ready
```

when line 600 is changed to

```
600      DATA 2.7,3,4,5,6.8
```

the following is printed:

```
RUNNH
 2.7            3               4               5               6.8

Ready
```

6.4 USER-DEFINED INTEGER FUNCTIONS

Functions can be written to handle integer variables as well as floating-point variables (see Section 3.7.3 and 8.1). A function is defined to be of integer type by following the function name with the % character.

A function to return the remainder when one integer is divided by another is shown below:

```
110      DEF FNR%(I%,J%) = I%-J% * (I%/J%)
```

and could be called later in a program as follows:

```
200     PRINT FNR%(A%,11%)
```

Integer arguments can be used where floating-point arguments are expected and vice versa as the system performs the necessary conversions. However, strings cannot be used where numbers are required (or vice versa).

```
75      DEF FNA%(X%) = X% - 1%
80      LET Z% = FNA%(12.34)
```

is acceptable. Z equals 11 after line 80 has been executed.

6.5 USE OF INTEGERS AS LOGICAL VARIABLES

Integer variables or integer-valued expressions can be used within IF statements in any place that a logical expression can appear. An integer value of 0% corresponds to the logical value FALSE, and any non-zero value is defined to be TRUE. The logical operators (AND, OR, NOT, XOR, IMP, EQV) operate on logical (or integer) data in a bitwise manner. The integer −1% (which is represented internally as 16 binary 1's) is normally used by the system when a TRUE value is required.

Logical values generated by BASIC always have the values −1% (TRUE) and 0% (FALSE).

The following immediate mode sequence illustrates the use of integers in logical applications in an IF statement:

```
IF -1% THEN PRINT 'TRUE' ELSE PRINT 'FALSE'
TRUE

Ready


IF 4% AND 2% THEN PRINT 'TRUE' ELSE PRINT 'FALSE'
FALSE

Ready

IF -1% AND 0% THEN PRINT 1 ELSE PRINT 2
 2

Ready

IF -1% IMP -1% THEN PRINT 'T' ELSE PRINT 'F'
T

Ready

IF 1<0 XOR -1% THEN PRINT 'TRUE' ELSE PRINT 'FALSE'
TRUE

Ready
```

A relational expression or a logical operation on relational expressions (see Section 2.6.5) can be used in place of an integer expression. Consider the following expression, which assigns the value of a logical product of relational expressions to an integer variable.

```
LISTNH
100      X=5
         \ Y=-6
110      I%= X>Y AND X>-1*Y
         \ PRINT I%
32767    END

Ready

RUNNH
 0

Ready
```

6.6 LOGICAL OPERATIONS ON INTEGER DATA

BASIC-PLUS permits a user program to combine integer variables or integer valued expressions using a logical operator to give a bit-wise integer result.

An integer value is represented internally in 2's complement notation as a sign bit and 15 data bits. Refer to Appendix E for the description of the internal format of an integer. In a logical operation, the corresponding bits of two integer values are combined on a bit-by-bit basis determined by the logical operator used. The logical operators are defined in Section 2.5.5.

For the purpose of logical operations, A and B as defined in the truth tables shown in Section 2.5.5 are modified. A becomes the condition of one bit in one integer value, and B becomes the condition of the bit in the corresponding bit position of another integer value. The truth tables are as follows.

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

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

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

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

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

A	NOT A
1	0
0	1

The result of a logical operation is an integer value generated by combining the corresponding bits of two integer values according to the rules shown in the truth tables above. For example, the following command prints the logical product of the integers 85% and 28%.

```
PRINT 85% AND 28%
 20

Ready
```

Each bit in the internal representation of 85% is combined with each corresponding bit in the internal representation of 28% according to the rules in the AND truth tables. By consulting the AND (logical product) truth table, it can be seen that a bit is generated in the bit position of the result only if both bits are 1 in the corresponding bit position of the integer values 85% and 28%. The resultant value of 20 printed by BASIC is the integer value of the bits set in the internal representation of the logical product.

Note that two values that would yield "true" (<> 0) values if considered independently can yield a "false" (=0) result if combined in a logical product expression. For example:

```
2% AND 4%
```

yields a logical product of 0, because they have no set bits in common. This potential problem can be avoided by using -1% as one of the logical operands.

The following command prints the logical sum of 85% and 28%.

```
PRINT 85% OR 28%
 93

Ready
```

From the OR (logical sum) truth table, it can be seen that a bit is generated in the bit position of the result if either the corresponding bit of the internal representation of 85% or 28% is a 1. The resultant value of 93 printed by BASIC is the integer value of the bits set in the internal representation of the logical sum.

The following command prints the logical difference of 85% and 28%.

```
PRINT 85% XOR 28%
 73

READY
```

The result of any logical operation can be assigned to an integer variable. For example, the following statement assigns a logical product to an integer variable which, in turn, can be printed.

```
LISTNH
100       C% = 85% AND 28%
          \ PRINT C%

Ready

RUNNH
 20

Ready
```

The logical operation can be used to mask a particular bit pattern. For example, the following BASIC-PLUS statement is used to generate the value of the low order eight bits, L%, of an integer word, W%.

```
        100        L% = W% AND 255%
```

The internal representation of 255% is such that the low order eight bits (bits 0 through 7) are all 1, and the high order eight bits (bits 8 through 15) are all 0. The AND operation (logical product) generates a bit in L% only if a bit appears in the corresponding bit position of both W% and 255%. Since 255% is known to contain all 0's in the high order bits and all 1's in the low order bits, the result L% reflects the presence of bits set and cleared in the low order eight bits of W%. Such a use of a bit pattern is called masking, where the internal representation of 255% is such that it provides a mask to hide one portion of a bit pattern (the high order bits of W%) and reveals another portion of a bit pattern (the low order bits of W%).

In summary, integer values can be combined as described in Section 6.2 using arithmetic (mathematical) operators to give arithmetic results. Integer values can be compared using relational operators (see Section 2.5.4) and can be combined using logical operators (see Section 2.5.5) to give either a TRUE or FALSE result as described in Section 3.5 or to give 0% for false or -1% for true as described in Section 6.5. In any case, the results of all relational and logical operations are integer values. When a logical operation is performed in conjunction with arithmetic and relational operations, the priority scheme as described in Section 3.5 is used to determine the hierarchy of operations.

Thus, with the feature described in this section, integer variables and integer-valued expressions can be operated on by AND, OR, XOR, EQV, IMP and NOT to give a bit-wise integer result.

6.7 MIXED MODE ARITHMETIC

The user can perform arithmetic operations using a mix of integer and floating-point numbers. To force a floating-point representation of an integer constant, terminate it with a decimal point. Use the % character as described in Section 6.1 to force an integer representation of a constant. Constants without a decimal point or % character are termed ambiguous. The remainder of this section describes the results of arithmetic operations using a mix of numbers.

If both operands of an arithmetic operation are either explicitly integer or floating point, the system generates, respectively, integer or floating-point results. If one operand of an arithmetic operation is an integer and another is floating-point, the system converts the integer to a floating-point representation and generates a floating-point result. For example:

```
PRINT 1%/2%; 1./2.; 1%/2.;1./2%
 0  .5  .5  .5

Ready
```

In the first two operations, the system generates the explicit results; in the second two, the system converts the explicit integer and generates floating-point results.

When an ambiguous constant appears in an arithmetic expression (for example, 10 as opposed to 10% and 10.), the system represents it in integer format if an integer variable (for example, I%) or an integer constant (for example, 3%) occurs anywhere to the left of the constant in the expression. Otherwise, the system treats the ambiguous constant as a floating-point number. The system performs the operation according to the rules described above. For example,

```
PRINT 1%/2; 1/2%; 1/2
 0  .5  .5

Ready
```

In the first operation, the system treats the 2 as an integer because an explicit integer representation appears to the left in the expression. In the next two operations, the system treats the ambiguous constants as floating-point numbers since no explicit integer variable or constant appears to the left of the ambiguous constant in the expression.

Since the format of the results determines the results of many operations, the user must explicitly impose the correct format by use of the percent sign or the decimal point. For example, compare the following calculations, assuming A (2%) = 0 in each expression.

```
PRINT A(2%) + (32767+2); A(2%) + (32767. + 2)
-32767  32769

Ready
```

The result of the first expression is guided by the appearance of the percent sign and forces an integer result. The decimal point in the second expression forces results in floating-point format. The same principle applies in the following example.

```
PRINT 1% + 1/2; 1. + 1/2; 1 + 1/2
 1   1.5   1.5

Ready
```

The choice of an explicit percent sign or the decimal point determines the format of the result, thus enabling the user to control the result.

6.8 FLOATING-POINT AND SCALED ARITHMETIC

Floating-point numbers occupy either two 16-bit words or four 16-bit words of storage in memory. With the single precision package, two words are used; with the double-precision package, four words are used. Appendix E describes the internal format of the two packages.

With the 2-word format, the user can accurately represent numbers up to six decimal digits, and, with the 4-word format, numbers up to 15 decimal digits. Both formats allow numbers in the range 10^-38 to 10^38 approximately. An attempt to assign or compute a number outside the allowed range causes the FLOATING POINT ERROR condition (ERR = 48).

The system performs output of numeric results of floating-point calculations as described in Section 2.6.1. To perform output of numbers larger than six digits, the user can tailor the format as described in Section 10.3.3 for the PRINT USING statement.

Usually fractional numbers cannot be represented exactly in binary notation, and certain calculations in floating point result in an accumulative error. For example, the following calculation, run in standard 4-word floating point, results in an accumulative error because the floating point-fraction .01 is not represented internally as that precise value.

```
LISTNH
100     X=0.
110     X = X + .01 FOR I% = 1% TO 10000%
120     PRINT X - 100.
32767   END

Ready

RUNNH
-.177636E-11

Ready
```

If no accumulated error exists, the result is 0. Running the example code on a system using the 2-word format generates a much greater accumulated error (approximately .00295).

To perform decimal calculation on a system having the double-precision floating-point (4-word) math package, the user can employ the scaled-arithmetic feature to avoid or reduce accumulated error in the fractional part of a number. The user can specify the number of decimal places in fractional numbers by use of the SCALE command. Systems with 2-word precision do not have scaled arithmetic.

NOTE

Scale arithmetic is included with the double-precision package primarily for compatibility with existing programs. The string arithmetic feature provides a more flexible and generally easier-to-use method for improving arithmetic precision.

With the scaled-arithmetic feature, the user can select a scale factor of 0 to 6. The system uses the scale factor to preserve the accuracy of fractional numbers to that number of decimal places. The value 0 is a special scale factor which disables the scaled-arithmetic feature and allows the system to perform calculations using standard double-precision floating-point arithmetic. The scale factor is 0 at log-in time.

With a scale factor of n between 1 and 6 in effect, the system, upon input of a floating-point number, internally moves the decimal point n places to the right and truncates it to an integer. The system performs all subsequent calculations with the floating-point integers and, in turn, translates the result of each arithmetic operation into a floating-point integer with the scale factor n. On output, the system moves the decimal point to the left n places (descales) and passes the result to the PRINT or PRINT USING routines to format.

A scale factor between 1 and 6 determines the accuracy of fractional numbers. For example, with a scale factor of 2 in effect, the following statement, upon input, causes the system to move the decimal point two places to the right.

```
X = .01
```

If any rounding is necessary, the system does it at this point. The system then converts the result, 1, to a floating-point representation. Similarly, .1 becomes 10 internally and all numbers less than .005 become 0.

The scaled-arithmetic conversion thus avoids the loss of precision inherent in representing fractional numbers in binary notation since the system can represent the integer accurately in floating-point format. This feature, therefore, allows more predictable arithmetic results. For example, running the following calculation with a scale factor of 2 yields a 0 result.

```
LISTNH
100      X=0.
110      X = X + .01 FOR I% = 1% TO 10000%
120      PRINT X - 100.
32767    END

Ready

RUNNH
 0

Ready
```

The scaling factor of 2 eliminates the inaccuracy in representing a fraction two places to the right of the decimal point.

The range of integer numbers which can be represented accurately decreases according to the scale factor in effect. For example, with a scale factor of 2 in effect, two of the 15 digits must be used to represent the two digits of fraction. There remain 13 places to accurately represent the integer portion of the number.

With a scale factor in effect, the system handles output by PRINT and PRINT USING statements in the standard manner. The PRINT statement still handles six digits or less and uses the E format for numbers larger than six digits. The PRINT USING statement formats numbers according to the specified string.

The mathematical functions described in Section 3.7 can be used in conjunction with the scaled-arithmetic feature. With a non-zero scale factor in effect, the system automatically descales the number passed, computes the value of the function, and converts the value returned to an appropriately scaled floating-point integer. No rounding occurs; places outside the scale factor range are truncated.

6.8.1 The SCALE Command

The following description of the SCALE command is intended only to show briefly how to use it and how it works. For a more detailed description, refer to the RSTS/E System User's Guide.

To specify a scale factor, enter the following command:

```
SCALE <scale factor>
```

The scale factor must be a decimal integer from 0 to 6. That scale factor remains in effect until another SCALE command is given, or until log-off. When a program is compiled, the current job scale factor is established as the scale factor associated with the stored program.

SCALE is a RSTS/E command, not a BASIC-PLUS statement; a program cannot refer to or modify a scale factor.

Typing SCALE without a scale factor value causes the current job scale factor to be printed. If the current program has an associated scale factor that differs from the job scale factor, the program scale factor is also printed. For example:

```
SCALE
2,6
↑ ↑
│ └────────Program scale factor
│
└────────── Current job scale factor

READY
```

If the current job scale factor differs from the scale factor of the current program, the job scale factor takes precedence. A user might load a compiled program using the RUN command and then discover that the scale factors are different. To run under the program scale factor, the user must:

1. Change the current job scale factor to that of the program.
2. Use the OLD command to load the source version of the program, this time under the desired scale factor.

CHAPTER 7
MATRIX MANIPULATION

This chapter deals with BASIC-PLUS matrix manipulation commands. Matrices can be composed of variables of any type. A single matrix, however, is composed of a single type of data; floating point, integer, or character string. The MAT operations do not set the zero elements [A(0), or B(0, n) and B(n, 0)] of the specified matrix to conform with the requested operation.

7.1 BASIC-PLUS ARRAY STORAGE

A BASIC-PLUS program can define the size of a matrix in one of two ways: explicitly, by including the matrix in a dimension statement, or implicitly, where the matrix does not appear in any dimension statement. Implicitly dimensioned matrices are assumed to have ten elements in each dimension referenced (size 10 for a 1-dimensional matrix and size 10-by-10 for a 2-dimensional matrix, with each dimension also having a zero row and column). Implicitly dimensioning the matrix A(I, J), for example, has the same effect as explicitly including the following statement:

```
Ready

100       DIM A(10,10)
```

Dimensioning a matrix (explicitly or implicitly) establishes two quantities for the system: the default number of elements in each row and column and the maximum number of elements in the matrix. Through use of the MAT commands, described in this chapter, the program can alter the number of elements in each row and the number of columns in the matrix as long as the total number of elements does not exceed the number defined when the matrix was dimensioned. Changing the number of elements in either or both dimensions is termed redimensioning the matrix.

When a matrix is redimensioned, the user program should take care not to reference elements outside the currently dimensioned range of the matrix. For example, if the range of matrix A is 5 by 7, referencing A(3,8) is improper and, although no error is generated, generally results in some element elsewhere in the matrix being destroyed.

7.2 MAT READ STATEMENT

The MAT READ statement is used to read the value of each element of a matrix from DATA statements. The format of the statement is as follows:

MAT READ <list of matrices>

Each element in the list of matrices indicates the maximum amount of the matrix to be read (which cannot be greater than the dimensioned size of the matrix). The individual elements are separated by commas. If the matrix name is used without a subscript, the entire matrix is read. For example:

```
100       DIM A%(20,20)
110       MAT READ A%
```

The above lines read a 20-by-20 matrix of floating-point data. Data is read row by row; that is, the second subscript varies most rapidly. If line 110 had read:

```
110       MAT READ A%(5,15)
```

a 5-by-15 matrix would be read and the matrix A would be redimensioned.

7.3 MAT PRINT STATEMENT

The MAT PRINT statement prints each element of a 1- or 2-dimensional matrix. The statement is of the form:

MAT PRINT <matrix name> $\left\{ \begin{matrix} , \\ ; \end{matrix} \right\}$

If the matrix name consists of an unsubscripted matrix name, the entire matrix is printed. If the matrix name is subscripted, then the subscript indicates the maximum size of the matrix to be printed (but does not redimension the matrix). Only one matrix can be output by a single MAT PRINT statement.

If the matrix name is followed by a semicolon (;), the data values are printed in a packed fashion. If the matrix name is followed by a comma (,), the data values are printed across the line with one value per print zone. If neither character follows the matrix name (the null case), each element is printed on a separate line.

```
100      DIM A(10,10),B(20,20)
110      MAT PRINT A;
         !PRINT 10*10 MATRIX, PACKED FORMAT
120      MAT PRINT B(4,6),
         !PRINT 4-BY-6 MATRIX, 5 ELEMENTS PER LINE
```

One-dimensional arrays can be printed in either row or column format.

```
220      MAT PRINT V
```

where V is a singly-dimensioned array, prints the array V as a column matrix, and

```
220      MAT PRINT V,
```

prints the array V as a row matrix, five values per line.

```
220      MAT PRINT V;
```

prints the array V as a row matrix, closely packed. For example:

```
LISTNH
100      DIM A(7), X(5)
110      MAT READ A,X
120      MAT PRINT A;
         \PRINT
         \MAT PRINT X
200      DATA 21,22,23,24,35,36,37,51,52,53,54,55
32767    END

Ready
```

```
RUNNH
 21   22   23   24   35   36   37

 51
 52
 53
 54
 55

Ready
```

Two-dimensional arrays are printed in ascending format. For example:

```
LISTNH
100      DIM A(2,3)
110      FOR I% = 1% TO 2%
         \ FOR J% = 1% TO 3%
         \ LET A(I%,J%) = I%*100% + J%
         \ NEXT J%
         \ NEXT I%
         \ PRINT
120      MAT PRINT A;
32767    END

Ready

RUNNH

 101   102   103

 201   202   203


Ready
```

7.4 MAT INPUT STATEMENT

The MAT INPUT statement is used to input the value of each element of a predimensioned matrix. The statement is of the form:

line number MAT INPUT <list of matrices>

Input is read from the keyboard, as with a normal INPUT statement, and a ? character is printed when the program is ready to accept the input. The LINE FEED key can be used to continue typing data on succeeding lines. The RETURN or ESCAPE key is used to enter the data to the system. MAT INPUT does not affect row 0 or column 0 of the matrix.

The MAT INPUT statement allows input of integer, floating-point or character string values depending upon the variable names. Where more than one matrix is to be input by the same MAT INPUT statements, the names are separated by commas. For example:

```
100      DIM A%(20),B(15)
110      MAT INPUT A%,B
```

causes the program to input 20 integer elements for the array A% and 15 floating-point values for the array B.

Where an array or matrix element is specified, for example:

```
200     MAT INPUT N%(25)
```

only 25 elements of the array are input, regardless of the number of elements originally specified when the array was dimensioned. The array is then redimensioned. For example:

```
LISTNH
100     DIM A(20,20)
110     MAT INPUT A(4,3)
120     PRINT
        \ MAT PRINT A,
32767   END

READY

RUNNH
? 5,8,4.5,2,0,1,9.2,0,6.8,2.7,3.01,6.345

 5               8               4.5

 2               0               1

 9.2             0               6.8

 2.7             3.01            6.345


READY
```

The matrix A is redimensioned in line 100. The INPUT statement proceeds to accept input until the entire matrix has been read or the RETURN or ESCAPE delimiter is encountered. Several lines can be input by terminating the physical keyboard line with a line feed to indicate continuation on the following line.

Following the input of a matrix, the two variables NUM and NUM2 contain the number of elements input. NUM contains the number of rows input or, for a 1-dimensional matrix, the number of elements entered. NUM2 contains the number of elements in the last row. For example, the following example program inputs a variable size matrix (up to 10-by-10):

```
LISTNH
100     DIM A(10,10)
110     INPUT 'TYPE MATRIX DIMENSIONS UP TO 10,10';N,M
        \ MAT INPUT A(N,M)
120     PRINT 'NUM =';NUM, 'NUM2 =';NUM2
130     IF NUM*NUM2=N*M THEN PRINT 'MATRIX FILLED'
        \ GO TO 32767
140     PRINT 'MATRIX NOT FILLED'
32767   END

Ready
```

```
RUNNH
TYPE MATRIX DIMENSIONS UP TO 10,10? 4,8
? 123,456,345,909,765,456,123,1,2,3,4,5,6,7,8,9,0,0,9,8,7,45
NUM = 3        NUM2 = 6
MATRIX NOT FILLED

Ready
```

Checking the contents of the matrix

```
MAT PRINT A;
 123   456   345   909   765   456   123   1

 2   3   4   5   6   7   8   9

 0   0   9   8   7   45   0   0

 0   0   0   0   0   0   0   0


Ready
```

Unlike the INPUT statement, no text string can be output with the MAT INPUT statement. For example:

```
100       MAT INPUT 'CONTENTS OF MATRIX';A%
?Syntax error at line 100

Ready
```

7.5 MATRIX INITIALIZATION STATEMENTS

A matrix initialization statement allows the user to create initial values for the elements of a matrix. The statement is of the form:

```
MAT <name> = <value>    (DIM1, DIM2)
                        (DIM1)
```

The name specified is the name of a predimensioned matrix, and the optional DIM1 and DIM2 specifications indicate the size of the matrix to be initialized. When specified, DIM1 and DIM2 cause the matrix to be redimensioned. The value can be one of the following:

Value	Meaning
ZER	Sets all elements of the matrix to 0 (this is true of all matrices when they are first created). (Function does not set row 0 or column 0.)
CON	Sets all elements of the matrix to 1. (Function does not set row 0 or column 0).
IDN	Sets up an identity matrix (all elements are 0 except for those on the diagonal, A(I,I), which are 1). (Function does not set row 0 or column 0.)

If no dimensions are indicated (DIM1 and DIM2 are not specified) in a matrix initialization statement, the existing dimensions of the matrix are assumed to be unchanged. For example:

```
LISTNH
100      DIM A(10,10), B(15), C(20,20)
110      MAT A=ZER
         !SETS ALL ELEMENTS OF A EQUAL TO 0
120      MAT B=CON(10)
         !SETS FIRST 10 ELEMENTS OF B EQUAL TO 1
130      MAT C=IDN(10,10)
         !SETS UP AN IDENTITY MATRIX


32767    END

Ready

RUNNH

Ready

MAT PRINT C;
 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


Ready
```

Again, note that these instructions do not set row 0 or column 0.

7.6 MATRIX CALCULATIONS

Mathematical operators and two intrinsic functions are available for use with matrices.

7.6.1 Matrix Operations

The operations of addition, subtraction, and multiplication can be performed on matrices using the common BASIC mathematical symbols.

Each of the matrix operation statements is begun with the word MAT and is followed by the expression to be evaluated. Each of the matrices involved must be predefined in a DIM statement. The subscripts of the matrices need not be indicated on the statement. The matrices indicated for any operation must be conformable to that operation. A subset of one matrix cannot be indicated as part of an operation.

```
110      DIM A(50), B(25), C(50)
120      MAT C=A+B
RUNNH
?Matrix dimension error at line 120

Ready
```

In order for line 120 to execute properly, line 110 should read:

```
110      DIM A(50), B(50), C(50)
```

Multiplication of conformable matrices is indicated as follows:

```
100      DIM D(10,5), C(5,10), R(10,10)
110      MAT R = D*C
```

By conformable matrices is meant that the number of columns in matrix D is equal to the number of rows in matrix C. The dimensions of the matrix R must be large enough to contain the number of columns in D and the number of rows in C. The operation MAT A=A*B or MAT A=B*A is illegal.

Scalar multiplication of a matrix is performed as follows:

```
150      MAT C = (K)*A
```

Each element of matrix A is multiplied by the scalar value (constant, variable, or formula) K, indicated in parentheses.

The form MAT A=(K)*A is legal. Matrix A can be copied into matrix C (providing sufficient space is available in matrix C) as shown below:

```
160      MAT C = A
```

7.6.2 Matrix Functions

Functions exist for the performance of transposition and inversion of matrices.

```
150      MAT C = TRN(A)
```

causes matrix C to be set equal to the transpose of matrix A. That is, C(I,J)=A(J,I) for all I,J; matrix C is redimensioned if necessary. For example:

```
10       DIM X(15,25), N(5,10), M(5,5)
80       MAT X=TRN(N)
90       MAT N=INV(M)
```

causes N to be computed as the inverse of matrix M (M must be a square matrix). After the inversion is complete, the function DET is set to the value of the determinant of matrix M. (If the matrix being inverted is sufficiently singular to make it impossible to complete the inversion, the message CAN'T INVERT MATRIX is printed.) The value of DET, then, can be used in subsequent computations as a formula value, as with any other function. For example:

```
LISTNH
200      MAT A = INV(X)
         \D1=DET
210      MAT B = INV(A)
         \D2=DET
220      IF D1=1/D2 GO TO 340 ELSE PRINT 'RELATIONSHIP TRUE'

Ready
```

Matrix inversion, like the other BASIC-PLUS matrix operations, does not operate on the elements of the row 0 and column 0 of the matrix; however, inversion destroys the previous contents of these elements. The operation MAT A = INV(A) is legal.

CHAPTER 8
ADVANCED STATEMENT FEATURES

8.1 DEF STATEMENT, MULTIPLE LINE FUNCTION DEFINITIONS

In Chapter 3 the DEF statement is described as being able to create a 1-line function, which the user can call as an element in a BASIC statement. In BASIC-PLUS multiple-line user-defined functions are also possible. The format for a multiple-line function definition is as follows:

DEF FN<identifier> <(dummy arguments)>

<body of definition>

FNEND

The multiple-line DEF function differs from the 1-line user functions by the absence of an equal sign following the function name on the first line. (From zero to five arguments of any type or mixture of types can be used.) The value returned by the function is the value of FN<identifier>at the time the FNEND statement is encountered, somewhere within the multiple-line definition there should be a statement of the form:

{LET} FN<identifier> = <expression>

It is the value of this expression which is returned as the value of the function. (There may be more than one such statement, as in the example below.)

The function example below determines the larger of the two numbers and returns that number. The IF-THEN statement is frequently used in multiple line functions as follows:

```
LISTNH
100     DEF FNM(X,Y)
110     LET FNM=X
120     IF Y<=X THEN GOTO 50
130     LET FNM=Y
140     FNEND

Ready
```

As another example, the following is a recursive[1] function that computes N-factorial:

```
LISTNH
100      DEF FNF(M%)
105      IF M% = 0% THEN FNF = 1
         \ GO TO 120
110      IF M% = 1% THEN FNF = 1
                   ELSE FNF = M% * FNF(M%-1%)
120      FNEND
130      INPUT 'VALUE FOR FACTORIAL'; M%
140      PRINT M%; 'FACTORIAL EQUALS'; FNF(M%)
32767    END

READY

RUNNH
VALUE FOR FACTORIAL? 6
 6 FACTORIAL EQUALS 720

READY

RUNNH
VALUE FOR FACTORIAL? 0
 0 FACTORIAL EQUALS 1

READY
```

Any variable referenced in the body of a function definition which is not an argument of that multiple-line DEF function has its current value in the user program. Multiple-line DEF functions can be nested (one multiple-line definition can reference another multiple-line definition of itself.) A multiple-line user-defined function may contain transfers of control outside its boundaries (via GOSUB, ON ERROR . . . RESUME, nested function references, or GOTO, for example). However, once a function is entered, it must be exited via its FNEND statement. Line numbers within the function cannot be referenced from outside the function boundaries except to return to a function that is currently invoked. If the program encounters an FNEND statement whose companion DEF FN . . . statement has not been executed, an error occurs.

Generally, transfers out of function boundaries should be avoided, because they may not execute as expected on other BASIC systems.

[1]The term "recursive" refers to an inherently repetitive process in which the result of each cycle is dependent upon the result of the previous cycle.

The parameters with which a user-defined function is called are strictly formal; attempts by the program to modify them are cancelled when the function exits to its calling program:

```
LISTNH
100      DEF FNB(X)
110      X=0
         \FNB=10
120      FNEND
200      A=1
         \B=FNB(A)
         \PRINT A,B
32767    END

Ready

RUNNH
 1                10

Ready
```

A is not set to 0 by the function FNB(A). However, any variable referenced in the body of the function definition which is not one of the function arguments will retain, after exit from the function, any value assigned to that variable during the execution of the function.

Functions can be written in any type and can contain any variety of argument types. For example:

```
LISTNH
100      DEF FNA$(A,B,C%)
110      IF A>B GOTO 130
120      FNA$=CHR$(A+1)
         \ GOTO 140
130      FNA$=CHR$(A+C%)
140      FNEND
200      INPUT 'VALUES FOR A,B,C%'; A,B,C%
210      PRINT 'FNA$(A,B,C%) = ';FNA$(A,B,C%)
32767    END

Ready

RUNNH
VALUES FOR A,B,C%? 36,7.5,24
FNA$(A,B,C%) = <

Ready

RUNNH
VALUES FOR A,B,C%? 45.2,5.67,8
FNA$(A,B,C%) = 5

Ready
```

8.2 ON-GOTO STATEMENT

The simple GOTO statement allows the user to unconditionally transfer control of the program to another line number. The ON-GOTO statement allows control to be transferred to one of several lines depending on the value of an expression at the time the statement is executed. The statement is of the form:

ON <expression> GOTO <list of line numbers>

The expression is evaluated and the integer part of the expression is used as an index to one of the line numbers in the list. For example:

```
50        ON X GOTO 100,200,300
```

transfers control to:

1. line number 100 if 1. <= x < 2.
2. line number 200 if 2. <= x < 3.
3. line number 300 if 3. <= x < 4.

Values of x out of this range cause the error message:

```
?ON statement out of range at line 50
```

is printed (or the user can transfer to an ON ERROR GOTO routine with ERR = 58).

8.3 ON-GOSUB STATEMENT

The GOSUB and RETURN statements are used to allow the user to transfer control of his program to a subroutine and return from that subroutine to the normal course of program execution (see Section 3.8 for details). The ON-GOSUB statement is used to conditionally transfer control to one of several subroutines or to one of several entry points to one (or more) subroutine(s). The statement is of the form:

ON <expression> GOSUB <list of line numbers>

Depending on the integer value (truncated if necessary) of the expression, control is transferred to the subroutine which begins at one of the line numbers listed. Encountering the RETURN statement after control is transferred in this way allows the program to resume execution at the line following the ON-GOSUB line.

An example of the statement follows:

```
80        ON X%-Y% GOSUB 900,933,1014
```

When line 80 is executed, the value of X%–Y% being either 1, 2, or 3 causes control to transfer to line 900, 933 or 1014, respectively. If the quantity X–Y is not equal to 1, 2 or 3, the error message:

```
?ON statement out of range at line 80
```

is printed (or the user can transfer to an ON ERROR-GOTO routine with ERR = 58).

Since it is possible to transfer into a subroutine at different points, the ON-GOSUB statement could be used to determine which portion of the subroutine should be executed.

8.4 ON ERROR GOTO STATEMENT

Certain errors can be detected by BASIC during program execution. These errors fall into two broad areas: computational errors (such as division by 0) and input/output errors (reading an end-of-file code as input to an INPUT statement). Normally the occurrence of any of these errors causes termination of the user program execution and the printing of a diagnostic message.

Some applications may require the continued execution of a user program after an error occurs. In these situations, the user can execute an ON ERROR GOTO statement within his program. This statement tells BASIC that a user subroutine exists, beginning at the specified line number, which will analyze any I/O or computational error encountered in the program and possibly attempt to recover from that error.

The format of the ON ERROR GOTO statement is as follows:

ON ERROR GOTO {<line number>}

This statement is placed in the program prior to any executable statements with which the error handling routine deals. If an error does occur, user program execution is interrupted and the user-written error subroutine is started at the line number indicated. The variable ERR, available to the program, assumes one of the values listed in Appendix C, the complete RSTS/E Error Message Summary.

When an error is encountered in a user program, BASIC checks to see if the program has executed the ON ERROR GOTO statement. If this is not the case, then a message is printed at the user's terminal and the program proceeds (if the error does not cause execution to terminate). If the ON ERROR-GOTO statement was executed previously, then execution continues at the specified line number. Then the program can test the variable ERR to discover precisely what error occurred and decide what action is to be taken.

NOTE

An ON ERROR GOTO statement with an incorrect target statement number can cause error messages that are confusing or seemingly inappropriate.

8.4.1 RESUME Statement

After the problem is corrected (if this is both possible and desired by the programmer), execution of the user program can be resumed through use of the RESUME statement (which is placed at the end of the error handling routine, much like a RETURN statement in a normal subroutine). The RESUME statement causes the program line that originally caused the error to be reexecuted. If execution is to be restarted at some other point within the program (as might be the case for a non-correctable problem), the new line number can be specified in the RESUME statement at the end of the error handling routine.

The format of the RESUME statement is as follows:

RESUME {<line number>}

For example:

```
2000    RESUME
2001    RESUME 100
```

The line 2000 restarts the user program at the line in which the error was detected, and is equivalent to the statement:

```
2000    RESUME 0
```

A RESUME or RESUME 0 statement in an error handling routine passes control to the line containing the statement which caused the error. If the statement which caused the error is on a multiple statement line, control is passed to the DIM, DEF, FNEND, FOR, NEXT or DATA statement immediately preceding it on the line. If none of these six statements is present on the line, control passes to the first statement on the line. For example, consider the line:

```
50      A=A + 1 \ PRINT A \ FOR M% = 1% TO 3%
                \ INPUT X(M%)
                \ NEXT M%
```

If an error occurs in the INPUT statement, above, control is passed to the preceding FOR statement on the same line – not to the first statement of the line. The loop is not reinitialized, however, and the INPUT statement is retried without changing M%.

For this reason, a DIM, DEF, FNEND, FOR, NEXT or DATA statement on a multiple statement line with error handling should be the first statement on a line. Also, the first statement on a line should be the statement which may generate the trappable error. Such placement of the statement prevents logic errors and allows any further error to be handled. Any other placement of the statement causes logic errors because statements preceding the statement causing the error are executed as many times as control is passed back to the line. If the error handling routine must also handle errors, the program can pass control to a RESUME statement which, in turn, can pass control to the error handling routine.

Line 2001 above restarts the user program at line 100 (which can be used to print some terminal message for that particular operation).

A RESUME statement should always be included in the error handling routine.

8.4.2 Disabling the User Error Handling Routine

If there are portions of the user program in which any errors detected are to be processed by the system and not by the user program, the error subroutine can be disabled by executing the following statement:

```
ON ERROR GOTO 0
```

which returns control of error handling to the system. An equivalent form is:

```
ON ERROR GOTO
```

in which case line 0 is assumed. Executing this statement causes the system to treat errors as it would if no ON ERROR GOTO had ever been executed.

Generally, the error handling subroutine detects and properly handles only a few different errors; it is useful to have the RSTS system handle other errors, if they occur. For this reason, RSTS allows the ON ERROR GOTO 0 statement to be executed within the error subroutine itself. Special treatment is accorded this case, in that the disabling occurs retro-actively; the error which caused entry to the error subroutine is then reported and a message printed as though no ON ERROR GOTO statement had been in effect.

As an example of this feature, consider an application in which inexperienced users interact with a BASIC program. These users may not know what to type at the terminal, and the program may want to prompt them. The program tells the system to allow up to 60 seconds for the user to respond (via the WAIT function, described in Section 8.8) and then to alert it that the user has not replied. The program then prints additional information for the user.

The program below requests the user's name with the INPUT statement on line 120. The ON ERROR GOTO statement is previously executed on line 100.

```
LISTNH
100       ON ERROR GOTO 1000
          !SET UP ERROR ROUTINE
110       WAIT(60)
          !WAIT 60 SECONDS FOR REPLY
120       INPUT 'YOUR NAME';N$
          !GET STUDENT NAME
150       STOP



1000      !THIS IS THE ERROR HANDLING ROUTINE
1010      IF ERR<>15 THEN ON ERROR GO TO 0
          !WAIT ERRORS ONLY
1020      PRINT
          !SKIP TO NEW LINE
1030      PRINT 'PLEASE TYPE YOUR NAME'
          \ PRINT 'AND THEN HIT "RETURN" KEY'
1050      RESUME
          !TRY AGAIN
32767     END

Ready
```

In this example, if the call to the error subroutine was caused by some error other than the KEYBOARD WAIT EXHAUSTED error, the program would exit via the ON ERROR GOTO 0 in line 1010. This permits the appropriate error message to be printed on the user's terminal. Note that exiting via the RESUME at line 1050 causes the INPUT statement to be restarted.

8.4.3 The ERL Variable

It is sometimes useful to be able to recognize the line number at which an error occurred. Following an error detection, the integer variable ERL contains the line number of the error.[1]

ERL would be used, for example, to indicate which of several INPUT statements caused an END OF FILE error.

Care must be taken in use of the ERL variable since changing or resequencing the line number field of all or some statements within the program can alter the value of the ERL variable as it appears within an expression context. For example:

[1] The one exception to this rule is the ?PROGRAMMABLE CTRL/C TRAP error (ERR = 28) as described in the *RSTS/E Programming Manual.* In this case ERL is not set, but the LINE Variable is set to the line number executing when CTRL/C was typed.

```
LISTNH
100      ON ERROR GOTO 1000
110      INPUT 'TYPE TWO NON-ZERO NUMBERS'; A,B
120      LET X=A/B
130      LET X=X+B/A
140      PRINT X
150      STOP
           .
           .
           .
1000     IF ERR<>61 THEN ON ERROR GOTO 0
1010     PRINT 'FIRST NUMBER WAS 0' IF ERL=130
1020     PRINT 'SECOND NUMBER WAS 0' IF ERL=120
32767    END

Ready

RUNNH
TYPE TWO NON-ZERO NUMBERS? 5,10
 2.5
Stop at line 150

Ready

CONT

Ready

RUNNH
TYPE TWO NON-ZERO NUMBERS? 6,0
SECOND NUMBER WAS 0

Ready

RUNNH
TYPE TWO NON-ZERO NUMBERS? 0,7
FIRST NUMBER WAS 0

Ready
```

If the LET statements in line 120 and 130 were moved to some other line numbers, lines 1010 and 1020 would also require a change.

8.5 IF-THEN-ELSE STATEMENT

The IF-THEN statement allows the program to transfer control to another line or execute a specified statement depending upon a stated condition.

The IF-THEN-ELSE statement is the same as the IF-THEN statement, except that rather than executing the line following the IF statement, another line number or statement can be specified for execution where the condition is not met. The statement is of the form:

$$\text{IF} \langle\text{condition}\rangle \quad \begin{bmatrix} \text{THEN} \langle\text{line number}\rangle \\ \text{THEN} \langle\text{statement}\rangle \\ \text{GOTO} \langle\text{line number}\rangle \end{bmatrix} \begin{Bmatrix} \text{ELSE} \langle\text{line number}\rangle \\ \text{ELSE} \langle\text{statement}\rangle \end{Bmatrix}$$

where the condition is defined as one of the following:

> <relational expression>
> <logical expression>

A relational expression is defined as:

> <expression> <relational operator> <expression>

as described in Section 3.5.

A logical expression is one of the following:

1. An integer expression (FALSE if 0, TRUE if <> 0)
2. A set of relational expressions, corrected by logical operators
3. A set of integer expressions, or logical expressions, or both, connected by logical operators.

The condition is tested; if it is true the THEN/GOTO part of the statement is executed. If the condition is false, the ELSE part of the statement is executed, Following the word ELSE is either a statement to be executed or a line number to which control is transferred.

As an example of an IF-THEN-ELSE statement:

```
175       IF X>Y THEN PRINT 'GREATER' ELSE PRINT 'NOT GREATER'
```

An IF statement can follow either the THEN or ELSE clause in the above statement, making it possible to nest IF statement to any desired level. For example:

```
LISTNH
100       INPUT 'ENTER THREE NUMBERS';A,B,C
110       IF A>B THEN
                  IF B>C THEN PRINT 'A>B>C'
                          ELSE IF C>A
                                  THEN PRINT 'C>A>B'
                                  ELSE PRINT 'A>C>B'
                  ELSE IF A>C THEN PRINT 'B>A>C'
                          ELSE IF B>C
                                  THEN PRINT 'B>C>A'
                                  ELSE PRINT 'C>B>A'
32767     END

Ready

RUNNH
ENTER THREE NUMBERS? 4.6,-.01,-3.5
A>B>C

Ready

RUNNH
ENTER THREE NUMBERS? 2005,2.8,3006
C>A>B

Ready
```

```
RUNNH
ENTER THREE NUMBERS? 3,2,4
C>A>B

Ready
```

The use of line continuation and TAB characters greatly improves the legibility of complex program statements such as line 110 above.

The IF-THEN-ELSE statement can appear anywhere in a multiple-statement line. However, if this statement is followed by any other statements, the following rules apply:

1. The physically last THEN or ELSE clause is considered to be followed by the next statement on the line:

```
210       IF A=1.0 THEN GOTO 100 ELSE PRINT A \ PRINT 'ABNORMAL'
```

   where A≠1, the value of A and the text string ABNORMAL are printed.
2. All other THEN or ELSE clauses are considered to be followed by the next line of the program:

```
20        IF A>B THEN IF B>C THEN PRINT 'B<C'
          \ GOTO 30
25        PRINT 'A<B'
```

   Only in the case where "B<C" is printed is the statement GOTO 30 seen and executed. If either A<B or B>C, the line "A<B" is printed.

8.6 CONDITIONAL TERMINATION OF FOR LOOPS

In the simple FOR-NEXT loop described in Section 3.6.1, the format of the FOR statement is given as:

FOR <variable> = <expression> TO <expression> STEP <expression>

There are many situations in which the final value of the loop variable is not known in advance and what is really desired is to execute the loop as many times as necessary to satisfy some condition. In evaluating a function, for example, this condition might be the point at which further iterations contribute no further accuracy to the result. BASIC-PLUS provides a convenient way of specifying that a loop is to be executed until a certain condition is detected or while some condition is true. These statements take the forms:

FOR <variable> = <expression> STEP <expression> WHILE <condition>

and

FOR <variable> = <expression> STEP <expression> UNTIL <condition>

The condition has the same structure as specified in an IF statement (see Section 3.5) and can be just as elaborate, if necessary. Before the loop is executed and at each loop iteration the condition is tested. The iteration proceeds if the result is true (FOR-WHILE) or false (FOR-UNTIL).

The difference between a FOR loop specified with a WHILE or UNTIL and one specified with a terminal value for the loop variable is worth noting, in order to avoid potential pitfalls in the usage of each. Consider the two loops in the program below:

```
LISTNH
10       FOR I=1 TO 10
         \ PRINT I;
         \ NEXT I
20       PRINT 'I=';I
50       FOR I=1 UNTIL I>10
         \ PRINT I;
         \ NEXT I
60       PRINT 'I=';I
32767    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
```

Each of these loops prints the numbers from 1 to 10. When the loop at line 10 is done, however, the loop variable is set to the last value used (that is, 10). In the second loop beginning at line 50, the loop variable is set to the value which caused the loop to be terminated (that is, 11).

Next consider the two loops following:

```
LISTNH
10       X=10
20       FOR I=1 TO X
         \ X=X/2
         \ PRINT I,X
         \ NEXT I
30       PRINT
         \ X=10
40       FOR I=1 UNTIL I>X
         \ X=X/2
         \ PRINT I,X
         \ NEXT I
32767    END

Ready

RUNNH
 1              5
 2              2.5
 3              1.25
 4              .625
 5              .3125
 6              .15625
 7              .078125
 8              .390625E-1
 9              .195313E-1
 10             .976563E-2

 1              5
 2              2.5

Ready
```

In the case of the loop beginning with line 20, the iteration stops when I exceeds the initial value of X (that is, 10). Even though the value of X changes within the loop, the initial value of X determines the performance of the loop. In the second loop, the current value of X determines when the iteration ceases. Thus, after three iterations, I is greater than X in the second loop and the loop is terminated. (The STEP value when omitted, is still assumed to be 1.)

These forms of loop control are particularly useful in iterative applications where data generated during the loop execution determines loop completion.

Consider the problem of scanning a table of values until two successive elements are both 0 or the end of the table is reached:

```
100       FOR I=1 UNTIL I=N OR X(I)=0 AND X(I+1)=0
          \ NEXT I
```

The following two programs also illustrate the FOR-UNTIL and FOR-WHILE constructions:

```
LISTNH
100       INPUT 'LETTER IS'; Y$
110       X$=''
120       FOR I%=0% UNTIL X$=Y$ OR X$ = 'ZZZ'
          \ READ X$
          \ NEXT I%
130       IF X$ = 'ZZZ' THEN PRINT 'IMPROPER INPUT'
          \ GO TO 32767
140       PRINT 'LETTER IS NUMBER';I%; 'IN ALPHABET'
500       DATA A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P,Q,R,S,T,U,V,W,X,Y,Z,ZZZ
32767     END

Ready

RUNNH
LETTER IS? J
LETTER IS NUMBER 10 IN ALPHABET

Ready

RUNNH
LETTER IS? 9
IMPROPER INPUT

Ready

RUNNH
LETTER IS? X
LETTER IS NUMBER 24 IN ALPHABET

Ready
```

```
RUNNH
LETTER IS? CC
IMPROPER INPUT

Ready


LISTNH
100      INPUT 'WORD';Y$
110      X$=''
120      FOR I% =0% WHILE X$<=Y$
         \ READ X$
         \ NEXT I%
130      IF I% = 1% THEN PRINT 'IMPROPER INPUT'
         \ GO TO 32767
140      PRINT 'WORD BEGINS WITH LETTER';I%-1%
500      DATA A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P,Q,R,S,T,U,V,W,X,Y,Z,ZZZ
32767    END

Ready

RUNNH
WORD? SECOND
WORD BEGINS WITH LETTER 19

Ready

RUNNH
WORD? /MESSAGE
IMPROPER INPUT

Ready
```

8.7 STATEMENT MODIFIERS

To increase the flexibility and ease of expression within BASIC-PLUS, five statement modifiers are available (IF, UNLESS, FOR, WHILE, and UNTIL). These modifiers are appended to program statements to indicate conditional execution of the statements or the creation of implied FOR loops.

8.7.1 The IF Statement Modifier

The form

    <statement> IF <condition>

is analogous to the form:

    IF <condition> THEN <statement>

For example:

```
100      PRINT X IF X <> 0.
```

is the same as:

```
        100       IF X<> 0. THEN PRINT X
```

The statement is executed only if the condition is true.

When a statement modifier appears to the right of an IF-THEN statement, then the modifier operates only on the THEN clause or the ELSE clause, depending on its placement to the left or right of ELSE. For example:

```
        100       IF 1=1 THEN PRINT 'HELLO' ELSE PRINT 'BYE' IF 2=2
```

will print:

```
        HELLO
```

since the test 1=1 is true. The modifier IF 2=2 is also true, but as it applies only to the ELSE clause, it is never tested.

It is not possible to include an ELSE clause when using the modifier form of IF.

Several modifiers may be used within the same statement. For example:

```
        100       PRINT X(I,J) IF I=J IF X(I,J) <> 0
```

prints the value of X(I,J) only if the value of X(I,J) is non-zero and if I equals J. When there is more than one modifier on a line, the modifiers are executed in a right-to-left order. That is, the rightmost one is executed first and the leftmost is executed last. This situation is described by the term "nested modifiers."

An additional operational advantage of IF modifiers is illustrated in the discussion of FOR modifiers in Section 8.7.3.

8.7.2. The UNLESS Statement Modifier
The form:

<statement> UNLESS <condition>

causes the statement to be executed only if the condition is false. This particular form simplifies the negation of a logical condition. For example, the following statements are all equivalent:

```
        100       PRINT A UNLESS A=0.
        110       PRINT A IF NOT A=0.
        120       IF NOT A=0. THEN PRINT A
        130       IF A <> 0 THEN PRINT A
```

8.7.3 The FOR Statement Modifier
The forms:

<statement> FOR <variable> = <expression> TO <expression> {STEP <expression>}

and

<statement> FOR <variable> = <expression> {STEP <expression>} $\left\{\begin{array}{l}\text{WHILE <condition>}\\ \text{UNTIL <condition>}\end{array}\right\}$

can be used to imply a FOR loop on a single line. For example (using none of the optional elements):

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

This statement is equivalent to the following FOR-NEXT loop:

```
100       FOR I=1. TO 10.
          \ PRINT I, SQR(I)
          \ NEXT I
```

In cases where the FOR-NEXT loop is extremely simple, the necessity for both a FOR and a NEXT statement is eliminated. Notice that this implied FOR loop will only modify (and hence execute iteratively) one statement in the program. Any number of implied FOR loops can be used in a single program.

As in the case with all modifiers, a FOR modifier in an IF statement operates only on the THEN or ELSE clause with which it is associated, and never on the conditional expression to the left of the THEN. Thus, if it was desired to print all non-zero values in a matrix X(100), the following program would not operate properly:

```
10        DIM X(100%)
15        READ X(I%) FOR I%=1% TO 100%
20        IF X(I%)<>0 THEN PRINT I%;X(I%),  FOR I%=1% TO 100%
```

since the implied FOR loop at line 20 applies only to the THEN PRINT . . . part of the statement, and not to the IF . . . part. The first value of X tested is X(100), since I remained at 100 from statement 15. To achieve the desired effect, it is only necessary to state line 20, not as an IF statement, but rather as a PRINT statement with nested modifiers; for example:

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

When expressed in the latter form, the nested modifier rule takes effect, and all the values of X(I) are tested and printed as appropriate.

The WHILE and UNTIL clauses are explained in Section 8.6.

8.7.4 The WHILE Statement Modifier
The form:

<statement> WHILE <condition>

is used to repeatedly execute the statement while the specified condition is true. For example:

```
10        LET X=X^2 WHILE X^2 < 1E6
```

is equivalent to:

```
LISTNH 10
10        LET X=X^2
          \ IF X < 1E6 THEN 10
```

The WHILE modifier (and the UNTIL modifier in Section 8.7.5) operates usefully only in iterative loops where the logical loop structure modifies the values which determine loop termination. This is a significant departure from FOR loops, in which the control variable is automatically iterated; a WHILE statement need not have a formal control variable. The following infinite loops never terminate:

```
10      X=X + 1 WHILE I<1000
20      PRINT I, A(I) WHILE A(I) <> 0
```

In both cases, the program fails to alter the values which are used to determine when the loop is done.

A successful application of the WHILE modifier is shown below:

```
LISTNH
100     ! TEST OF SQUARE ROOT ROUTINE
110     X = 1 +(X*2%) WHILE X = SQR(X^2%)
120     PRINT X
32767   END

Ready

RUNNH
%Floating point error at line 110
 .184467E 20

Ready
```

8.7.5 The UNTIL Statement Modifier
The form:

<statement> UNTIL <condition>

is used to repeatedly execute the statement until the condition becomes true; which is to say, while the condition is false. For example:

```
100     X = 1 + X*2% UNTIL X <> SQR(X^2%)
        \ PRINT X
```

is the same as:

```
100     IF X = SQR(X^2%) THEN GO TO 110 ELSE GO TO 130
110     X = X*2% + 1
120     GO TO 100
130     PRINT X
```

8.7.6 Multiple Statement Modifiers
More than one modifier can be used in a single statement. Multiple modifiers are processed from right to left. For example:

```
100     LET A=B IF A>0 IF B>0
```

is equivalent to any of the following:

```
100       IF B>0 THEN IF A>0 THEN A=B
```

or

```
100       IF B>0 AND A>0 THEN LET A=B
```

or

```
100       IF B<=0 THEN 150
110       IF A<=0 THEN 150
120       LET A=B
150       ! TEST OF A AND B COMPLETE
```

A 2-dimensional matrix (m by n) can be read one row at a time as follows:

```
150       READ A(I,J) FOR J=1 TO M FOR I=1 TO N
```

which is equivalent to:

```
150       MAT READ A(N,M)
```

and to:

```
150       FOR I=1 TO N
160       FOR J=1 TO M
170       READ A(I,J)
180       NEXT J
190       NEXT I
```

Also see Section 8.7.3 which described the interaction of FOR and IF modifiers.

8.8 SYSTEM FUNCTIONS AND STATEMENTS

RSTS/E has several system functions which allow the user to obtain certain information about or perform operations with the system. The functions are described in Table 8-1. (Several system functions used frequently in user programs are also described in earlier sections.)

Table 8-1
System Functions

Function	Meaning	Sample Usage

DATE$ (0%)	Returns the current day, month and year, in the form:
	
	22-Mar-76
	
	Note that the date contains both upper and lower case characters
	
	If the run time system is generated with the numeric data option, DATES returns the date as year.month.day; for example:
	
	76.03.22
		`PRINT DATE$(0)`
		`07-Nov-76`
		
		`Ready`

Table 8-1 (Cont.)
System Functions

Function	Meaning	Sample Usage

DATE$ (N%)	Returns a character string corresponding to a calendar date. The formula used to translate between N and the date is as follows:
	
	(day of year) + [(number of years since 1970)*1000]
	
	DATES(1%)="01-Jan-70"
	DATES(2060%)="29-Feb-72"
	
	If the run time system is generated with the numeric data option, DATES return the data as year.month.day; for example:
	
	76.02.29
		`PRINT DATE$(6278)`
		`04-Oct-76`
		
		`Ready`

TIME$ (0%)	Returns the current time of day as a character string as follows:
	
	TIME(0)="05:30 PM"
	or "17:30 "
		`175 IF TIME$(0) >= '06:00 PM'`
		`THEN PRINT 'DINNERTIME'`

TIME$ (N%)	Returns a string corresponding to the time at N minutes before midnight. For example:
	
	TIME$(1%)="11:59 PM"
	or "23:59 "
	
	TIME$(1440%)="12:00 AM"
	or "00:00 "
	
	TIME$(721%)="11:59 AM"
	or "11:59 "
	
	N% must be less than 1441 to return a valid string. The format of the string (i.e., 02:40 PM or 14:40) is determined at system generation time.
		`PRINT TIME$(1)`
		`11:59 PM`
		
		`Ready`
		
		`PRINT TIME$(1400)`
		`12:40 AM`
		
		`Ready`

Table 8-1 (Cont.)
System Functions

Function	Meaning	Sample Usage

TIME (0%)	Returns the clock time in seconds since midnight.	`25 IF TIME(0)>43200`
		`THEN PRINT 'AFTERNOON'`

TIME (1%)	Returns the central processor (CPU) time used for the job in 0.1 second quanta.	`10 IF TIME(1)>30`
		`THEN STOP`
		`!3 SECONDS ALLOWED`

TIME (2%)	Returns the connect time (time during which the user has been logged into the system) for the job in minutes.	`340 IF TIME(2)>1000 THEN STOP`
TIME (3%)	Returns the number of kilo-core ticks (KCT's) used by the job. (See the RSTS/E System User's Guide for an explanation of KCT's.)	`180 PRINT TIME(3)`

TIME (4%)	Returns the device time for the job in minutes.	`220 IF TIME(4)/60>2.5`
		`THEN GOTO 90`

SWAP% (I%)	Causes a byte swap operation to occur on the integer variable I%; returns the value of I% with the bytes swapped.	`300 PRINT CHR$(SWAP%(I%))`
RAD$(I%)	Converts an integer to a 3-character string. This function is used to convert a value (expression in Radix-50 format) back into ASCII. Radix-50 is explained in Appendix D of the RSTS/E Programming Manual.	`280 PRINT RAD$(I%)`

There are also two special system statements that can be used within a BASIC-PLUS program: SLEEP and WAIT. Both statements allow the user to suspend his program for a stated interval.

The SLEEP statement is of the form:

SLEEP <expression>

SLEEP is used to dismiss the currently running program for the number of seconds indicated by the expression. At the end of this period the program is again runnable. Thus, the user is guaranteed at least this number of seconds idle time, possibly slightly more depending upon the number of jobs currently active on the system.

To awaken a job from a sleep before the specified number of seconds has expired, type a delimiter (RETURN, LINE FEED, FORM FEED or ESCAPE) at any of the job's terminals. The program segment shown below, however, can be used to override line terminating delimiters and provide a continuous SLEEP for a specified time.

```
100       T=TIME(0)
110       SLEEP T+30-TIME(0)
          \ IF TIME(0)-T<30 GOTO 110
120       INPUT X
```

In the above program, the INPUT statement is executed only if the time elapsed is equal to or greater than 30 seconds. Otherwise, if a delimiter is typed, the SLEEP is executed again for the length of time remaining in the original 30 seconds or until another line terminating character is typed.

A job is also awakened when it has declared itself a receiver and a message is queued for it through the SEND/RECEIVE system function calls. (The SEND/RECEIVE system function calls are documented in the RSTS/E Programming Manual.

The WAIT statement is of the form:

WAIT <expression>

WAIT is used to set a maximum period for the system to wait for input from the user keyboard. If no delimiter is typed at the keyboard (RETURN, LINE FEED, ESCAPE) within the number of seconds specified by the expression, the program is restarted and a KEYBOARD WAIT EXHAUSTED error (ERR=15) occurs, which can be detected using ON ERROR-GOTO. The WAIT statement is used in conjunction with the INPUT statement. As an example:

```
LISTNH
10        ON ERROR GOTO 100
20        WAIT 15
30        INPUT '16 + 16 =';A
40        WAIT 0
50        IF A=32 THEN PRINT 'RIGHT!'
                  ELSE PRINT 'NO, TRY AGAIN'
                  \ GOTO 10
60        STOP
100       IF ERR<>15 THEN ON ERROR GOTO 0
110       PRINT 'WAKE UP!'
120       RESUME 30
32767     END

Ready

RUNNH
16 + 16 =? WAKE UP!
16 + 16 =? 30
NO, TRY AGAIN
16 + 16 =? 32
RIGHT!
Stop at line 60

Ready
```

In this example line 100 is executed only if the user fails to respond within 15 seconds. The use of WAIT 0 restores the terminal to its normal state in which no timeout occurs, but rather the system waits until a line is entered, however long that may take.

PART III

BASIC-PLUS DATA HANDLING

Part III contains a complete description of all BASIC-PLUS data handling operations. A brief review is made of the simple forms of READ, DATA, PRINT, RESTORE and INPUT along with the more advanced forms of these statements. Formatted ASCII files, virtual core matrices, and record I/O operations are described.

Advanced users are advised to consult the *RSTS/E Programming Manual* for a description of advanced data handling techniques and device dependent operations.

CHAPTER 9
DATA STORAGE CAPABILITIES

9.1 FILE STORAGE

Previous chapters have described techniques for entering data into a program as the program is written (via READ and DATA statements) or from the user terminal while the program is executing (via the INPUT statement). Either technique is inefficient when more than a few items are to be read or written. Thus BASIC-PLUS also provides facilities to define and manipulate permanent data files.

A BASIC-PLUS data file consists of a sequence of data items transmitted between a BASIC program and an external input/output device. The external device can be the user terminal, some other terminal, disk, line printer, card reader, magnetic tape device, DECtape, or high-speed paper tape equipment.

Each data file and device has both an external name by which it is identified within the RSTS/E system (the name of the file on a disk storage device, for example) and a channel number used to reference the file. An OPEN statement associates and external file specification with an internal file channel.

This chapter describes briefly the key concepts and terminology of data storage as it concerns a BASIC-PLUS user. More detailed information is available in the *RSTS/E Programming Manual* and the *RSTS/E System User's Guide.*

9.2 OPEN STATEMENT

The OPEN statement associates a file on a file-structured device or some non-file structured device with an I/O channel number internal to the BASIC program. (The I/O channel is a logical entity, having no relationship to a hardware channel.) BASIC-PLUS permits up to 12 files to be open at a given time. Channel numbers 1% to 12% specify device channels, and channel number 0% specifies the user's terminal.

The general form of the OPEN statement is as follows:

```
OPEN <string>   [FOR INPUT ]   AS FILE <expression>
                [FOR OUTPUT]
```

The string field is a character string constant, variable or expression that contains a filename. It can also contain a device designation (logical or physical), a project-programmer code, a file extension specifying the file type, and a protection code; these optional fields are described in the *RSTS/E System User's Guide.* After opening a file or device, the program performs input and output by referring to the channel number, as indicated by the expression following the key word FILE. The expression must be an integer in the range 1 to 12.

Protection codes are normally specified only in the NAME-AS statement, which changes the name and protection code of an existing file (see Section 9.4). However, protection codes can be specified as an optional part of any filename. For example.

```
330      OPEN 'FILE.EXT<40>' FOR INPUT AS FILE 1%
```

The file FILE.EXT is created under the current account with a protection code of <40>. The AS FILE expression corresponds to the internal channel number on which the file is being opened; channel 1 in this case.

One or more of the following specifications can be appended to the end of the statement. They are described in Sections 9.2.1 through 9.2.4.

{,RECORDSIZE <expression>} {,CLUSTERSIZE <expression>}

{,FILESIZE <expression>} {,MODE <expression>}

Options must be in the exact order shown. If options are out of order, the message

?MODIFIER ERROR

appears. Omitting an option is equivalent to specifying that option with a parameter of 0% except for the MODE option on non-file-structured magtape.

There are three distinct forms for the OPEN command:

OPEN <string> FOR INPUT
OPEN <string> FOR OUTPUT
OPEN <string>

The form of the OPEN statement used determines whether an existing file is to be opened or a new file created.

1. An OPEN FOR INPUT statement causes a search for an existing file (since the statement indicates the file is an input file). If no file is found, the CAN'T FIND FILE OR ACCOUNT error (ERR=5) occurs.

```
150     OPEN 'FILE.DAT' FOR INPUT AS FILE 2%
```

2. An OPEN FOR OUTPUT statement causes a search for an existing file which, if found, is deleted. A new file is then created.

```
270     OPEN 'DATA.01<40>' FOR OUTPUT AS FILE 3%
```

3. An OPEN statement without an INPUT or OUTPUT designation attempts to perform an OPEN FOR INPUT operation as described above. If this fails, a new file is created.

```
180     OPEN 'MATR.TER' AS FILE 7%
```

The OPEN statement does not control whether the program attempts to perform input or output on the disk file or whether read and/or write access to the file is granted[1]; these privileges are controlled by the file protection code and MODE value.

If, for some reason, the program cannot gain access to the file or device, an error is returned. Table 9-1 summarizes some of the more common errors that occur on attempted file access.

[1] Magtape and DECtape are exceptions to this rule. See the *RSTS/E System Programming Manual.*

Table 9-1
Open Statement Errors

Value of ERR	Message	Explanation
1	?Bad directory for device	The directory of the referenced device is in an unreadable or illegal format.
2	?Illegal file name	The filename specified is not acceptable. It contains unacceptable characters or the filename specification format has been violated.
4	?No room for user on device	Directory space for the current user of the specified device has been exceeded, or the device as a whole is too full to accept further data.
5	?Can't find file or account	The file or account number was not found on the specified device.
6	?Not a valid device	The device specification supplied is not valid for one of the following reasons: 1. Either the unit number or its type is not in the system configuration. 2. The logical name has no associated physical device and is thus untranslatable.
8	?Device not available	The specified device exists on the system but a user's attempt to ASSIGN or OPEN it is prohibited for one of the following reasons: 1. The device is currently reserved by another job. 2. The user lacks necessary access privileges for the device. 3. The device is disabled. 4. The device is a keyboard line for pseudo-keyboard use only.
10	?Protection violation	The user does not have the necessary access privileges for the file.
14	?Device hung or write locked	User should check hardware condition of device requested. Possible causes of this error include a line printer out of paper or a high-speed reader being off-line.
17	?Too many open files on unit	Only one open DECtape output file is permitted per DECtape drive. Only one open file per magtape drive is permitted.
32	?No buffer space available	The user accessed a file and the monitor requires one small buffer to complete the request. No small buffer is currently available.
39	?Magtape select error	When access to a magtape drive was attempted, the selected unit was found to be off-line.
46	?Illegal I/O channel	An I/O channel number was specified outside the range of integers 1 through 12.

When used with disk files, an OPEN FOR INPUT or OPEN FOR OUTPUT allows either read or write operations on the opened file. The system allows write access to a file if the protection code permits and if no other user has write access to the file. For example, if user 1 opens a file, he has read and write access. When user 2 opens the same file, he has read access only; a PROTECTION VIOLATION error occurs when he attempts to write on that file. When user 1 subsequently closes the file, no user has write access until the next open operation. User 3 can now open the file and obtain both read and write access, because no other user currently has write access to that file. On DECtape and magnetic tape devices, the FOR INPUT and FOR OUTPUT clauses restrict operations on that file to the type of operation specified.

NOTE

Only one user can have write access to a file at a time unless MODE 1% (that is, update mode) is used for disk files.

The next four sections in this manual describe generally the RECORDSIZE, CLUSTERSIZE, FILESIZE and MODE options of the OPEN statement. See the RSTS/E Programming Manual for device-dependent details. As these are sophisticated file-handling tools, it is suggested that the novice user initially skip these sections and continue with Section 9.2.5.

9.2.1 RECORDSIZE Option

When any file is opened, the system creates a buffer area in the user's memory space to buffer all I/O to and from the file. Normally the amount of space reserved is determined by the device, because each device has a default device buffer size as described in Table 9-2.

Table 9-2
Default Device Buffer Size

Device	Default Device Buffer Size (in characters or bytes)
All disks	512[1]
Floppy disk	512
DECtape	510[1]
Magtape (DOS or ANSI)	512[2]
High-speed reader	128
High-speed punch	128
Line printer	128
Card reader	160
User terminal	128

[1] The default buffer size may differ when the device is used as a non-file structured device.

[2] For ANSI magtape, the system reads a value from the header label to establish the buffer size.

With the RECORDSIZE option, the user program can allocate more buffer space than is provided by the default case. However, in some cases the particular device driver may not permit additional space to be used.

The buffer size must be an even number. If odd, BASIC uses the next lower integer. To learn the size of the buffer on any channel n, the user can invoke the BUFSIZ function. Table 9-3 shows the buffer size alterations for specific devices.

Table 9-3
Use of RECORDSIZE

Device	Possible Buffer Alterations
Disk	The disk drivers permit use of any buffer size that is an integral multiple of 512 bytes.
DECtape	The DECtape driver uses only the first 510 bytes of the available buffer space (512 bytes for non-file structured DECtape).
Magtape (DOS or ANSI)	The magtape driver uses only enough bytes for one physical magtape record. Each physical record must be at least 14 bytes, and be no larger than the buffer size.
High-speed reader High-speed punch Line printer User Terminal	These non-file structured devices can use any selected buffer size greater than the default size.
Card reader	The card reader driver uses only enough bytes for one card's data.
Floppy disk	The floppy disk driver permits use of any buffer size that is an integral multiple of 128 bytes.

The RECORDSIZE option has significant advantages when used with magtape and disk files. On a disk file, total throughput can be improved by using a larger buffer size, as this permits a single disk transfer to read a large quantity of data. Specify only an even number of bytes in the RECORDSIZE expression. For example:

```
100     OPEN 'MASTER.DAT' FOR INPUT AS FILE 1%, RECORDSIZE 2048%
```

If the file MASTER.DAT were on an RF11 disk and occupied a contiguous area on that disk, a 2048-byte transfer would take about 33ms while four 512-byte transfers would take about 83ms (on the average). If the file did not reside in a contiguous disk area, the RSTS Monitor would break the 2048-byte transfer into four 512-byte transfers. Even in this last case, the system overhead to perform the transfer would be less.

This example raises the question of how to ensure that a file occupies a contiguous disk area. This can be done by means of the MODE option (see the *RSTS/E Programming Manual*) or the CLUSTERSIZE option described in the following section.

BASIC-PLUS establishes the default buffer size when a value less than the default is given. To obtain a buffer size that is less than the default, specify the desired buffer size, plus 32767%, plus 1%. Smaller buffer sizes are often useful when performing I/O operations on alternate channels.

Thus, to open the paper-tape reader with a buffer size of two bytes for use with alternate channel I/O, write

```
OPEN "PP:" FOR INPUT AS FILE 1%,
        RECORDSIZE 32767% + 1% + 2%
```

9.2.2 CLUSTERSIZE Option

The CLUSTERSIZE option applies only to disk and ANSI magtape files created with an OPEN or OPEN FOR OUTPUT statement. The CLUSTERSIZE specification is ignored in all other cases.

The following description applies to disk files. CLUSTERSIZE is also legal for ANSI magtape. See the *RSTS/E Programming Manual* for further information.

The RSTS system divides each disk into a number of 256-word blocks. Each block is assigned a unique logical block number starting at 1.[1] Logical block numbers are assigned such that block n is physically contiguous with blocks n + 1 and n - 1.

A number of contiguous blocks taken together as a unit are called a cluster. RSTS permits clusters to be 1, 2, 4, 8, 16, 32, 64, 128 or 256 blocks long. When the disk is initialized (the process by which the disk is cleared for use on RSTS), a minimum cluster size can be established. This minimum cluster size (called the pack cluster size) for the disk can be 1, 2, 4, 8, or 16 blocks.

For each file on the system, an entry is made in the owner's file directory (User File Directory or UFD) containing the retrieval information for the file: filename, cluster size for the file, and a sequential list of clusters belonging to that file.

A UFD has a fixed maximum size which is determined when the UFD is created.[2] A UFD on any one disk cannot exceed 112 (decimal) blocks (28,672 words). If all files were a minimum size (7 or fewer clusters long) a UFD clustered as 16 would have room for a maximum of 1157 files. To keep the list of blocks belonging to the file as short as possible, the UFD contains a 1-word entry for the first block of each cluster. Knowing the first block number of the cluster and the cluster size is sufficient to determine all of the blocks in the cluster.

Because of the size limit on the UFD, large files benefit from the specification of large cluster sizes. In an extreme example, the UFD would be completely filled by a single file of 24,283 blocks where the file cluster size is one block. However, with a cluster size of 256 blocks, only 128 words of the UFD are required to describe this file.

Since most user files are not extremely large, omitting the CLUSTERSIZE option when creating the file makes little practical difference. Omitting the CLUSTERSIZE option is equivalent to specifying CLUSTERSIZE 0% and has the effect of assigning a cluster size equal to the pack cluster size for the disk on which the file resides. An attempt to create a file with a cluster size less than the pack cluster size or not a power of 2 causes the ILLEGAL CLUSTER SIZE error message (ERR=23).

A negative cluster size may be used; this suppresses the ILLEGAL CLUSTER SIZE error when the program uses a scratch disk whose maximum cluster size is not known. If the cluster size is negative, the system attempts to create the file at the absolute value of the specified cluster size or at the pack cluster size, whichever is larger.

Once a file is opened on an internal I/O channel, all I/O requests by the BASIC program are handled by means of a read or write call from BASIC-PLUS to the Monitor, directed to the nth virtual block of the file. The RSTS system translates the virtual block number into a logical block number. This is done by reading the file's retrieval information and finding the entry corresponding to the nth virtual block. To minimize the overhead involved in reading the UFD, which is stored on the disk, part of this list of clusters belonging to a file is kept in memory. This part of the list is called the file window. The file window is composed of seven entries from the list of file clusters. Since each entry corresponds to one cluster of the file, with a file cluster size of one block, seven blocks (or 3584 bytes) of the file are described by the in-memory file window. These seven blocks can then be read or written without accessing the complete list from the UFD stored on the disk. Similarly, with a file cluster size of 256 blocks, the file window describes the location of 1792 blocks of the file, or over 900,000 bytes. When performing random access I/O to virtual memory arrays and RECORD I/O files, any of the 1792 blocks would be read or written without referencing the UFD.

[1] Block 0 of each disk is reserved for a bootstrap record and is not used by any file.

[2] The maximum size of a UFD is seven times the cluster size for that UFD, which is established when the account is created, and may be 1, 2, 4, 8 or 16 blocks. The figures given in the text assume a UFD cluster size of 16.

As an example of the use of the CLUSTERSIZE option:

```
    100       OPEN 'MAT.DAT' FOR OUTPUT AS FILE 1%, CLUSTERSIZE 128%
```

In this case the file MAT.DAT is created with a cluster size of 128 blocks. Note that the file is initially 0 blocks long and is extended as needed in 128-block increments.

Since files with large cluster sizes must be extended by a whole cluster at a time and since clusters are always contiguous blocks, it may not always be possible to find sufficient contiguous free blocks to extend the file. If not, the NO ROOM FOR USER ON DEVICE error message is printed (ERR=4). The user should be aware of this possibility whenever he creates a file with a cluster size larger than the pack cluster size (the minimum cluster size for that disk).

As another example (using line continuation as described previously):

```
    100       OPEN 'DATA' FOR OUTPUT AS FILE 1%,
                   RECORDSIZE 2048%,
                   CLUSTERSIZE 4%
```

The RECORDSIZE option improves disk throughput when multiple blocks can be read or written in a single transfer (see Section 9.2.1). By creating the file with a cluster size of 4 (2048 bytes per cluster) the user guarantees that virtual blocks 1–4, 5–8, etc., of his file are physically contiguous on the disk.

9.2.3 FILESIZE Option

A disk file (and only a disk file) can be pre-extended by using the FILESIZE option in an OPEN statement, eliminating the need for a PUT statement. The format for the FILESIZE option is:

OPEN <string> [FOR OUTPUT] AS FILE <expr> , FILESIZE <expr>

For example:

```
    100       OPEN 'VALUES' FOR OUTPUT AS FILE 3%, FILESIZE 50%
```

The data file, VALUES is opened and automatically pre-extended to 50 256-word blocks.

FILESIZE is also used on ANSI magtape, but for a different purpose than described above. See the *RSTS/E Programming Manual* for further information.

9.2.4 MODE Option

The OPEN statement allows another option: the MODE field. The format of the OPEN statement, including the MODE field, is as follows:

$$\text{OPEN <string>} \left[\begin{matrix}\text{FOR INPUT}\\ \text{FOR OUTPUT}\end{matrix}\right] \text{AS FILE <expr>}$$

$$\left\{\text{,RECORDSIZE <expr>}\right\}$$
$$\left\{\text{,CLUSTERSIZE <expr>}\right\}$$
$$\left\{\text{,MODE <expr>}\right\}$$

The MODE option is used to establish device-dependent properties of the file. See the *RSTS/E Programming Manual* for device-dependent features.

9.2.5 File-Structured Vs. Non-File-Structured Devices

RSTS/E distinguishes between file-structured (disk, DECtape and magtape) devices and non-file-structured devices. When a file is to be found or created on a file-structured device, the file specification string in the OPEN statement must include both a device designation (or default public structure) and a filename. On non-file-structured devices, the device name alone identifies a file (filename and extension, if specified, are ignored). For example:

DT0:	Insufficient information to specify a file.
DT0:FRED	Specifies the file FRED on DECtape unit 0.
PP:	Uniquely specifies the high-speed punch.
PP:FILE	Same effect as PP:; the filename is ignored.
DX1:	Uniquely specifies floppy disk unit 1.

File specification syntax is such that the default device (the public disk storage area) need not be specified.

It is also possible to open a file-structured device in non-file-structured mode. For example:

```
160       OPEN 'DK2:' AS FILE 5%
```

is sufficient to open a disk cartridge in non-file-structured mode. Device-dependent features are described in the *RSTS/E Programming Manual.*

9.3 CLOSE STATEMENT

The CLOSE statement is used to terminate I/O between the BASIC program and a peripheral device. Once a file has been closed, it can be reopened for reading or writing on any internal file designator. On a CLOSE the system reclaims buffer space assigned to the file.

All files must be closed before the end of program execution. Execution of a CHAIN statement automatically closes any open files, but does not cause the output of the last blocks to output files. (The CHAIN statement is described in Section 9.6.) The format of the CLOSE statement is as follows.

CLOSE <expression>{, <expression> . . .}

The expression indicated has the same value as the expression in an OPEN statement and indicates the internal channel number of the file to close. Any number of files can be closed with a single CLOSE statement; if more than one file is to be closed, the expressions are separated by commas. For example:

```
240       CLOSE 10%
250       CLOSE 2%,4%
```

Line 250 above closes the files opened on internal I/O channels 2 and 4. Line 240 closes the file open on internal I/O channel 10.

9.4 NAME-AS STATEMENT, FILE PROTECTION AND RENAMING

The NAME-AS statement is used to rename and/or assign protection codes to a disk or DECtape file, and can only be used on a given file by someone logged into the system under an account number that has write privilege for that file. The format of the statement is as follows:

NAME <string> AS <string>

The specified file (the first string indicated) is renamed (as the second string indicated). When the file resides on a device other than the default device (system disk), the device must be specified in the first string and may optionally be specified in the second string. No filename extension assumptions are made by NAME-AS; the filename extension must be specified in both strings if any extension is present in the old filename or desired in the new filename. For example:

```
110      NAME 'DT0:OLD.BAS' AS 'NEW.BAS'
```

This is equivalent to:

```
110      NAME 'DT0:OLD.BAS' AS 'DT0:NEW.BAS'
```

However, the statement:

```
190      NAME 'FILE1.BAS' AS 'FILE2'
```

is not advised because FILE2 has no extension for automatic recognition by the system.

A file protection code can be specified within typed angle brackets as part of the second <string> although it is not required. If a new file protection code is specified, it is reflected in the protection assigned to the renamed file. If no new protection code is specified, the old protection code is retained. See the *RSTS/E System User's Guide* for a complete description of protection codes.

The statement:

```
200      NAME 'FILE.EXT' AS 'FILE.EXT<40>'
```

changes only the protection code of the file FILE.EXT stored on the system disk.

The statement:

```
200      NAME 'DT0:ABC.BAS' AS 'XYZ.BAS'
```

changes the name of the file ABC.BAS on DECtape unit 0. Since no transfer of the file from one device to another can be performed with the NAME-AS statement, it is not necessary to mention DT0: twice; that is, the device of the new filename need not be specified. However, an error is generated if a device other than the old device is specified.

```
120      NAME 'NEW' AS 'NEW1'
```

changes only the name of the disk file NEW.

9.5 KILL STATEMENT

The KILL statement is of the form:

KILL <string>

and causes the file-named string to be deleted from the user's file area. (The file can no longer be opened, but if it is already open the file remains available until it is closed.) For example, when the user has completed all work with the file XYZ (note that the filename has no extension) on the system disk, he could remove the file from storage by executing the following statement:

```
460      KILL 'XYZ'
```

A user is not allowed to KILL a file is write-protected against him.

9.6 CHAIN STATEMENT

If a user program is too large to be loaded into memory and run in one operation, the user can segment the program into two or more separate programs. Such programs are called into core for execution by means of a CHAIN statement. Each program section is assigned a name and control can be transferred between any two programs. A CHAIN statement is of the form:

CHAIN <string> {<line number>}

and causes the program named by the string to be called, compiled (if necessary), and executed. The line number, if specified, designates the line at which the program is to be started. If the line number is omitted, the program is started at the lowest numbered line (as though a RUN command has been used). The CHAIN statement is the last statement executed in each program segment (except the last segment). For example:

```
1000     CHAIN 'MAIN.BAC' 2000
```

causes the program MAIN.BAC to be loaded, beginning execution at line 2000.

Notice that a filename extension is not required. The compiled form of the program is searched for and, if found, run. If the compiled form is not found, the non-compiled form is searched for and, if found, compiled and run. If neither form of the program is found, an error occurs.

On the other hand, if a filename extension is specified, and not found, an error occurs; in this case, no other form of the program is searched for.

NOTE

If a CHAIN statement in a nonprivileged program names a privileged program, the CHAIN statement should not include a line number. The entire chained program must be executed, or the system will not retain the chained program's privileges.

Chaining to precompiled program files (.BAC files) is considerably more efficient than chaining to BASIC source program files since .BAS files require compilation upon each call.

Communication between chained programs is performed by means of user's files or memory common.

When the CHAIN statement is executed, all open files for the current program are closed, the new program segment is loaded, and execution continues. Any files to be used in common by several programs should be opened in each program.

The CHAIN statement implicitly closes all open I/O channels, which is slightly different from the actions performed as a result of a CLOSE statement. For example, the line printer drivers perform two page ejects when the line printer is closed with a CLOSE statement. To continue printing on the same piece of paper after chaining, do not direct a CLOSE statement to the line printer channel. The CHAIN statement is sufficient to close the printer without unwanted page ejects.

CHAIN does a fast close on all open files in the chaining program. This saves the data in the file, but any partially filled output buffer or modified virtual array elements may be lost. Thus it is generally good programming practice to include an explicit CLOSE statement in the program for all open files before chaining or exiting.

Similarly, an explicit close of the paper tape punch causes a trailer to be punched; the implicit close does not.

When a program is entered via a CHAIN statement, the STATUS variable is set. For more information see the ***RSTS/E Programming Manual.***

CHAPTER 10
BASIC-PLUS INPUT AND OUTPUT OPERATIONS

10.1 READ AND DATA STATEMENTS

A READ statement is used to assign to a list of variables values obtained from a data pool composed of one or more DATA statements. The two statements are of the form:

READ <list of variables>
DATA <list of values>

The list of variables can include floating-point, integer, subscripted, or character string variables. Data values must correspond in type with their respective variables, but the % character should not be included in integer values. Integer and floating-point values are interchangeable, although they are stored according to the type of the variable. The use of quote marks is discussed below.

The data pool consists of all DATA statements in a program. Values are read starting with the DATA statement having the lowest line number and continuing to the next higher, etc. The location of DATA statements in a program is irrelevant, although for simplicity they are usually kept together toward the end of the program. (The DATA statements must occur in the proper numeric sequence, however.) A DATA statement must be the last or only statement on a line, although a READ statement can occur anywhere on a line. Comments are not permitted at the end of a DATA statement. If a READ statement is unable to obtain further data from the data pool, an error message is printed and program execution is terminated. (This error can be treated through the ON ERROR GOTO statement, Section 8.4.)

Quotes are necessary in DATA statements only around string items that contain a comma, significant spaces or tabs, or lower-case letters that are to be preserved. The data pool, composed of values from the program's DATA statements, is stored internally as an ASCII string list. When a numeric variable is read, the appropriate ASCII to numeric conversions are performed. When a string variable is read, the string is used as it appears in the DATA statement. If the item does not appear in quotes, then spaces and tabs in the string are ignored. If the item appears in quotes, the string variable is equated to the entire string within the quotes.

Matrices are read from DATA statements via the MAT READ statement of the form:

MAT READ <matrix>

This reads the value of each element of a predimensioned matrix from the data pool. Each element in the list of matrices indicates the maximum dimension of the matrix to be read (which cannot be greater than the dimensioned size of the matrix). Individual elements are separated by commas. For example:

```
100     DIM A(20,20), B(50)
110     MAT READ A
120     MAT READ B(35)
```

The above lines read values for the 20-by-20 matrix A and 35 out of the possible 50 values for the B matrix (remaining elements are 0). Data is read row by row; that is, the second subscript varies most rapidly.

10.2 RESTORE STATEMENT

The RESTORE statement reinitializes the data pool of the program's DATA statements. This makes it possible to recycle through the DATA statements beginning with the lowest numbered DATA statement. The RESTORE statement is of the form:

RESTORE

For example:

```
     85          RESTORE
```

causes the next READ statement encountered after executing line 85 to begin reading data from the first DATA statement in the program, regardless of where the last data value was found. See Section 3.3.1 for an example program using the RESTORE statement.

The RESTORE statement can be placed in any position on a multiple-statement line.

10.3 PRINT STATEMENT

In its simplest form, the PRINT statement:

PRINT

causes a carriage return/line feed to be performed on the user terminal. The format:

PRINT <list>

causes the printing of the elements in the list on the user terminal. An element in the list can be any legal expression. When an element is not a simple variable or constant, the expression is evaluated before a value is printed. The list can also contain character strings between quotes which are printed exactly as typed between quotes.

NOTE

If a character string is enclosed in a PRINT statement with an initial quote and no terminating quote, a terminating quote is considered to follow the last character of that PRINT statement's terminal line (up to the carriage return, line feed, form feed, or escape character). For example:

```
10          PRINT 'NAME IS A$
10          PRINT "NAME IS A$"
20          PRINT 'NAME IS ' A$
```

Line 10 is shown in two equivalent forms. Line 20 is the correct form to generate the printed line:

```
NAME IS JOHN DOE
```

where A$ = "JOHN DOE".

Elements in the list are separated by commas or semicolons. For example:

```
100         A% = 1%
            \ B% = 2%
            \ C% = 3%
110         PRINT A%; A% + B% + C%, C% - A%, 'END'
```

when executed causes the following line to be printed:

```
 1  6            2              END
```

A terminal line is considered to be divided into print zones[1] of 14 spaces each. Use of these zones involves the comma character which causes the print head to move to the next available print zone (from 1 to 14 spaces away). If the right-most print zone on a line is filled, the print head moves to the first print zone on the next line.

The semicolon character functions as follows:

1. If an integer or floating-point variable, function, or expression is followed by a semicolon, the value is printed with a preceding minus sign if the number is negative, or a preceding space if it is positive. The number is then followed by a single space.
2. Character strings and string variables followed by a semicolon are printed with no preceding or trailing spaces.

Any PRINT statement which does not end with a semicolon or comma character causes a skip to the next line after printing the elements in the list. The presence of the punctuation character at the end of the PRINT list causes the next PRINT statement to continue on the same line under the conditions already defined.

In general, the output rules for the PRINT statement are:

1. Leading 0's and trailing 0's to the right of a decimal point are suppressed. Where a number can be represented as an integer, printing of the decimal point is also suppressed.
2. At most six significant digits are printed, unless the PRINT-USING statement is used.
3. Most numbers are printed in decimal format. Numbers too large or too small to be printed in decimal format are printed in exponential format.
4. Character string constants are printed without leading or trailing spaces.
5. Extra commas cause print zones to be skipped.
6. Semicolons separating character string constants from other list items are optional; omitting punctuation has no effect on the output format in this case.

10.3.1 Formatted ASCII I/O

BASIC-PLUS permits access to data files by three methods:

1. Formatted ASCII
2. Virtual memory arrays, described in Chapter 11
3. Record I/O, described in Chapter 12

Formatted ASCII data files are the simplest method of data storage, involving a logical extension of the PRINT and INPUT statements to be used in conjunction with the OPEN statement.

The formats for INPUT and PRINT statements to be used with the OPEN statement are:

```
line number INPUT #<expression>,<list>
line number PRINT #<expression>,<list>
```

where the expression has the same value as the expression in the OPEN statement (the internal file designator) and the list is a list of variable names, expressions, or constants as explained in the sections describing the PRINT and INPUT statements.

[1] The actual number of print zones is INT (n/14), where n is the size of the print line.

10.3.2 Output to Non-Terminal Devices

In order to direct output to a device other than the user terminal, the PRINT command is formatted as follows:

```
PRINT #<expression>,<list>
```

where the expression is the internal channel number (the internal file designator) of a previously opened output file (see Section 9.2). The list of information to be output can include any of the output information described as applicable to the PRINT statement. For example:

```
100     OPEN 'DATA1' FOR OUTPUT AS FILE 7%
110     PRINT #7%, 'START OF DATA FILE"
```

The above lines open a file called DATA1 on the disk with internal channel number 7 (of 12 possible open files available to the user). The first line in that file reads: START OF DATA FILE.

To output a table of square roots on the line printer, the following program could be used:

```
LISTNH
100     LET I$='LP:'
        \ OPEN I$ FOR OUTPUT AS FILE 1%
110     PRINT #1%, I, SQR(I) FOR I= 1. TO 5.
32767   END

Ready

RUNNH

Ready

PRINT I, SQR(I) FOR I=1. TO 5.
 1             1
 2             1.41421
 3             1.73205
 4             2
 5             2.23607

Ready
```

The user terminal can be addressed by referring to channel 0, or by associating a filename with the keyboard, as shown in the following two examples.

```
LISTNH
100     PRINT #0%, 'TEXT'

Ready

RUNNH
TEXT

Ready
```

```
LISTNH
100     OPEN 'KB:' AS FILE 1%
        \ PRINT #1%, 'TEXT'

Ready

RUNNH
TEXT

Ready
```

10.3.3 PRINT-USING Statement

In order to perform formatted output, the following statement is used:

PRINT{#<expression>,}USING<string>,<list>

where the expression (which is optional) indicates the internal channel number of the file or device which is the destination of the output; the string is either a string constant, string variable, or string expression which is an exact image of the line to be printed. This string is called a format field. The list is a list of items to be printed in the format specified by the format field. All characters in the string are printed as they appear except for the special formatting characters and character combinations described on the following pages. The string, or portions of the string, are repeated until the list is exhausted. The string is constructed according to the following rules.

10.3.3.1 Exclamation Point – An exclamation point in the format field identifies a 1-character string field. The variable string is specified in the <list> within the PRINT statement. For example:

```
LISTNH
100     PRINT USING '!!!', 'AB', 'CD', 'EF'
32767   END

Ready

RUNNH
ACE

Ready
```

The first character from each of the three string constants or variables is printed. Any other characters beyond the first are ignored.

10.3.3.2 String Field – A variable string field of two or more characters is indicated in the format field by spaces enclosed between backslashes. The backslash character (\) is produced by typing SHIFT/L on some keyboards. Enclosing no spaces indicates a field two columns wide, one space is equivalent to a field three columns wide, etc. For example:

```
100     PRINT USING '\\\  \', 'ABCD', 'EFGHI'
```

causes

```
ABEFGH
```

to be printed at the user's terminal. The first two backslashes have no spaces enclosed, hence permit the printing of two characters (AB). The second two backslashes enclose two spaces and permit the printing of four characters (EFGH). No spaces are printed unless specifically planned.

10.3.3.3 Numeric Field – Numeric fields are indicated with the # character in the format field. Any decimal point arrangement can be specified and rounding is performed as necessary (not truncation). For example:

```
100      PRINT USING '###.##',12.345
```

causes

```
12.35
```

to be printed on the user's terminal. Also consider the following:

```
100      PRINT USING '####', 12.345
110      PRINT USING '####.', 12.345
120      PRINT USING '##', 100
RUNNH
  12
  12.
% 100

Ready
```

Numeric fields are right justified; that is, if a number does not fill the allotted space, leading blanks precede the number. When the field specified is too small for a constant or variable to be printed, the % character is printed to indicate the error. The number is then printed without reference to the format field. On the other hand, when the format field specified is more than 20 character spaces larger than required for a constant or variable to be printed, a PRINT-USING BUFFER OVERFLOW non-recoverable error may occur.

If the format field specifies a digit as preceding the decimal point, at least one digit is always output before the decimal point. If necessary, that digit is 0.

10.3.3.4 Asterisks – If a numeric field designation in the format field begins with **, any unused spaces in the format field are filled with asterisks. For example:

```
100      A=27.95
         \ B=107.50
         \ C=1007.50
         \ PRINT USING '**##.##', A,B,C
RUNNH
**27.95
*107.50
1007.50

Ready
```

Notice that the ** characters act as two additional # characters as well as allowing asterisk fill.

Exponential format (see below) cannot be used in a field with leading asterisks. Negative numbers cannot be output using asterisk fill unless the sign is output following the number (see below).

10.3.3.5 Exponential Format – When the exponential form of a number is desired, the numeric format field is followed by the string (four characters) which allocates space for E-xx. Any arrangement of decimal points is permitted. For example:

```
LISTNH
100       F$='##^^^^######'
110       A=10000.
120       PRINT USING F$,A,A

Ready

RUNNH
10E 03 10000

Ready
```

All format positions are used to output a number with an exponent. The significant digits are left justified and the exponent is adjusted.

10.3.3.6 Trailing Minus Sign – If a numeric format field designation is terminated with a minus sign, the sign of the output number is printed following the number, rather than preceding it. A blank is printed to indicate a positive number. For example:

```
LISTNH
100       A=-10.5
110       PRINT USING '##.##- ####.##', A,A

Ready

RUNNH
10.50-   -10.50

Ready
```

Note that if the trailing minus is not used, space must be reserved in the numeric format field for the sign to precede the number.

10.3.3.7 Dollar Signs – If a numeric format field begins with $$, a dollar sign is printed immediately preceding the first digit of the number. For example:

```
LISTNH
100       A=77.44
          \ B=304.55
          \ C=2211.40
110       PRINT USING '$$###.##', A,B,C

Ready

RUNNH
 $77.44
$304.55
% 2211.4

Ready
```

Note that the $$ characters provide for the printing of two additional characters in the number. Since one character is a $, the effect is to allow for one additional # designation beyond those typed by the user.

Exponential format (see above) cannot be used in a field with leading dollar signs. Negative numbers cannot be output using the floating dollar character unless the sign is output following the number (see above).

10.3.3.8 Commas – If one or more commas appear to the left of the decimal point (if any) in a numeric format field, then commas are inserted every three digits to the left of the decimal point. A comma to the right of the decimal point is considered a printing character. For example:

```
LISTNH
100      PRINT USING '#,######.## ####.#,#', 12345.5, 123.456, 1

Ready

RUNNH
  12,345.50   123.5,1

Ready
```

10.3.3.9 Insufficient Format – If insufficient format characters are present in a field when a number is output, a % character is printed in the first position of the field followed by the number in standard format, usually causing the field to be widened to the right. The user is guaranteed his entire number. For example:

```
LISTNH
100      PRINT USING '##.## ##.##', 12.345, -12.5

Ready

RUNNH
12.35 %-12.5

Ready
```

Rounding occurs when digits are dropped at the right of numbers. If rounding causes the number to exceed the format allowed, the % character is used. For example:

```
100      PRINT USING '.##       .##', .125, .999
RUNNH
.13      % .999

Ready
```

10.3.3.10 Format Too Large – If a numeric format field results in an attempt to output more significant digits then are available for the number, 0's are substituted for all digits following the last significant digit. Six significant digits are available with the 2-word, single precision math package and 15 digits with the 4-word, double-precision math package. Up to 29 formatting characters for single precision and 19 formatting characters for double precision are permitted with the PRINT USING statement. An attempt to print fields larger than 29 or 19 results in the following error message:

?PRINT USING BUFFER OVERFLOW

In certain cases, a number larger than the format field is truncated without an error message. Good programming practice is to check for overflow before attempting to output numbers near the maximum size.

10.3.3.11 PRINT Statement Punctuation – When the PRINT-USING statement is used, the usual PRINT statement punctuation characters (commas and semicolons) have no effect on the output format, except that a comma or semicolon at the end of the PRINT list inhibits termination of the printed line. For example:

```
LISTNH
100      PRINT USING '##     ##  ##',1;2,3

Ready

RUNNH
 1      2    3

Ready
```

As another example:

```
LISTNH
100      PRINT USING '#.##', 2.5;
110      PRINT 'X'

Ready

RUNNH
2.50X

Ready
```

As another example:

```
100      A=1.32519
         \ B=2.45457
         \ LET F$ = '   A=##.##   B=##.##'
110      OPEN 'LP:' FOR OUTPUT AS FILE 4%
120      PRINT #4%, USING F$, A, B
```

would cause:

```
   A= 1.33   B= 2.45

 Ready
```

to be printed on the line printer.

10.3.4 MAT PRINT Statement

The MAT PRINT statement allows for easy printing of a predimensioned matrix. The statement is of the form:

MAT PRINT {#<expression>,} <matrix name>

For example:

```
100       DIM A(16)
110       MAT PRINT A(15)
```

If the specified matrix name is unsubscripted, the entire matrix is printed. If the matrix specification is subscripted, the subscript(s) indicates the maximum size of the matrix to be printed.

The matrix name can be followed by a semicolon to indicate that the values are to be printed in a packed fashion, or by a comma to indicate that each element is printed in its own zone. For example:

```
LISTNH
100       DIM A(10,10),B(10,20)
110       MAT PRINT A;
          !PRINT MATRIX A IN PACKED FORMAT
120       MAT PRINT B(10,10),
          !10*10 MATRIX PRINTED 5 VALUES PER LINE

Ready
```

Row and column matrices can also be printed. For example:

```
LISTNH
10        DIM A(5),B(10)
20        MAT PRINT A;
          ! PRINT MAT A ON ONE LINE
30        MAT PRINT B
          ! PRINT IN COLUMN FORMAT

Ready
```

Line 30 causes A to be printed as a row matrix, closely packed; line 40 causes B to be printed as a column matrix. The form:

MAT PRINT A,

would cause the matrix A to be printed as a row matrix, five values per line (at the user terminal).

10.3.5 PRINT Functions

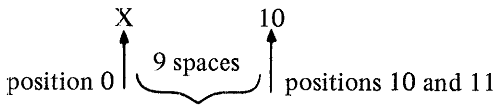
In order to aid in formatting simple and complex PRINT statements the following functions are provided:

Function	Meaning
POS(X)	Returns the current position on the output line; where X is the I/O channel number. POS(0%) returns the value for the user's terminal.
TAB(X)	Tab to position X in the print record. For example, a standard terminal has 72 printable columns numbered 0 through 71. TAB (4%) causes sufficient spaces to be output to move the print head to column 4. If the print head is currently past position 4, no spaces are output.

For example:

```
     100       PRINT 'X' \ \; TAB(10);POS(0)
```

causes the following to be printed:



10.4 INPUT STATEMENT

The INPUT statement allows data to be entered to a running program from an external device, the user's keyboard, disk, DECtape, paper tape reader, etc. The full form for this statement is:

INPUT [#<expression>,] <variable list>

In many cases the simpler form:

INPUT <variable list>

is used. This last form causes a ? to be printed at the terminal and the system then waits for the user to respond with the appropriate values of string or numeric variables. If a sufficient number of values are not typed, the system prints another ?; if too many values are typed, separated by commas, excess values are ignored. The user can also insert printed messages between the variables to be input. For example:

```
     100       INPUT 'YOUR NAME IS';N$, 'ACCOUNT NUMBER';A,'THANK YOU'
```

when executed would allow the following interaction at the terminal (the underlined characters are typed by the user):

```
     RUNNH
     YOUR NAME IS? JOE
     ACCOUNT NUMBER? 347654
     THANK YOU
     Ready
```

ON ERROR GOTO statements can be used in a program to trap recoverable errors which occur during input statement execution. The errors shown below occur most frequently when an INPUT statement is executed.

Error	Meaning	Examples
%DATA FORMAT ERROR (ERR = 50)	Data input in an illegal form	3.4.5 or $2 or #16 or 2;3 or LORA input for a numeric variable; X" or "HELLO" "THERE" input for a string variable
?ILLEGAL NUMBER (ERR = 52)	Overflow or underflow	3E+66 or --23
?END OF FILE ON DEVICE (ERR = 11)	Input CTRL/Z	^Z

The system assigns values to variables as they are input. Multiple variables can be assigned by separating them in the INPUT variable list by commas. Similarly, use commas or the RETURN key to separate values as they are input from the keyboard. For example:

```
100      INPUT X,Y,Z
110      PRINT X,Y,Z
RUNNH
? 3.14
? 14,92
 3.14          14            92

Ready
```

Do not use commas within a single number; the system ignores all characters input beyond a comma unless another variable is to be assigned. For example:

```
          Right                              Wrong

LISTNH                             LISTNH
100      INPUT R                   100      INPUT R
         \ PRINT R                          \ PRINT R

Ready                              Ready

RUNNH                              RUNNH
? 25902                            ? 25,902
 25902                              25

Ready                              Ready
```

Quotation marks (") should be used with string variables when embedded commas and spaces are to be preserved. For example:

```
          Right                              Wrong

LISTNH                             LISTNH
100      INPUT M$                  100      INPUT M$
         \ PRINT M$                         \ PRINT M$

Ready                              Ready

RUNNH                              RUNNH
? 'MOUSE, MICKEY'                  ? MOUSE, MICKEY
MOUSE, MICKEY                      MOUSE

Ready                              Ready
```

The format:

INPUT #<expression>,<string variable>

causes input to be read from the file or device indicated in the expression, by the internal file designation number given when the file was opened. (See Section 9.2 for a description of the OPEN statement.) If the value of the expression is non-zero and the specified file is the user terminal, open as an input device, then no ? character is printed at the terminal when input is requested.

For example:

```
LISTNH
100      OPEN 'KB:' FOR INPUT AS FILE 2%
110      INPUT #2%,A
120      PRINT A

Ready
```

The system then pauses while the user types a numeric value for the variable A, although no prompting ? or character string message is printed on the terminal.

```
RUNNH
567.8
 567.8

Ready
```

Another format of the INPUT statement allows the user to enter an entire line of data as a single character string entity, regardless of embedded spaces or punctuation. This is different from the normal mode of string input, where the comma, apostrophe, single quote and double quote characters have special significance. The format is:

INPUT LINE {#<expression>,} <string variable>

For example:

```
150      INPUT LINE A$
```

pauses to allow the user to enter a line followed by the RETURN, FORM FEED, LINE FEED or ESCAPE key (see also Section 5.3). Every character input, including quotation marks and commas, is present in A$, above. The end of the line being input is the carriage return/line feed sequence (or line feed/carriage return/null or ESCAPE, (see Section 5.3) which is appended to the data typed by the user. To remove the end-of-line sequence, use the CVT$$ function, described in Section 12.5.

END OF FILE ON DEVICE (ERR=11) occurs when CTRL/Z is input.

As another example:

```
100      OPEN 'F2.DAT' FOR INPUT AS FILE 7%
110      INPUT LINE #7%, B$
```

These lines cause the system to open a file F2 on the system disk on channel 7 (of 12 possible channels) and to read a string of characters up to the next LINE FEED character.

10.4.1 MAT INPUT Statement

The MAT INPUT statement is used to input the values of a pre-dimensioned matrix from a specified input device. Where no device is specified, the input is accepted from the user terminal. For example:

```
200       MAT INPUT A(20)
```

causes 20 floating-point values to be accepted as elements of the matrix A. A statement of the form:

MAT INPUT {#<expression>,}<variable list>

causes the input to be read from a file or device previously opened on the internal channel indicated by the expression.

```
140       DIM B(10,25)
200       OPEN 'DT1:DATA1' FOR INPUT AS FILE 1%
210       MAT INPUT #1%, B(10,25)
```

The above lines cause the file DATA1 on DECtape 1 to be opened for input on channel 1 (of 12 possible channels) and a matrix of values for the elements of B to be read to fill B(10,25). The zero elements are not assigned a value. When input is from channel 0 (i.e., the user terminal), ? is printed; however, reference to another channel does not cause the printing of the prompting character. Depending upon the name of the matrix, the MAT INPUT statement allows input of floating-point, integer, or character-string values.

10.4.2 Opening the User Terminal as an I/O Channel

The internal file designator (following the # character in the INPUT or PRINT statements) is always in the range 1 to 12. File designator 0 is, by definition, always open as the user's terminal. Internal file designator 0 cannot be closed or opened. Use of file #0 is indicated below (no OPEN #0 statement is necessary or allowed).

```
100       INPUT #0,A$
```

is equivalent to:

```
100       INPUT  A$
```

It is sometimes useful to be able to request keyboard input without having the "?" prompting character printed first. This can be accomplished by opening the user's terminal ("KB:") on some internal file designator other than 0. The ? character is only generated for input requests on channel #0, shown in the following example:

```
LISTNH
100       OPEN 'KB:' AS FILE 1%
110       PRINT 'WITH USE OF INTERNAL FILE DESIGNATOR'
          \ PRINT 'TYPE YOUR NAME, FOLLOWED BY RETURN KEY'
120       INPUT #1%, A$; 'THANK YOU'
          \ PRINT
          \ PRINT
          \ PRINT 'FOR COMPARISON, WITHOUT FILE DESIGNATOR'
          \ PRINT 'TYPE YOUR NAME, FOLLOWED BY RETURN KEY'
          \ INPUT A$; 'THANK YOU'
32767     END

Ready

RUNNH
WITH USE OF INTERNAL FILE DESIGNATOR
TYPE YOUR NAME, FOLLOWED BY RETURN KEY
J. P. JONES
THANK YOU

FOR COMPARISON, WITHOUT FILE DESIGNATOR
TYPE YOUR NAME, FOLLOWED BY RETURN KEY
? J. P. JONES
THANK YOU
Ready
```

CHAPTER 11
VIRTUAL ARRAY STORAGE

Many applications require a capability to individually address and update records on a disk file in a random (non-sequential) manner. Other applications may require more memory for data storage than is economically feasible. BASIC-PLUS fills both these requirements with a simple random-access file system called virtual memory.

The BASIC-PLUS virtual array facility provides a mechanism for the programmer to specify that a particular data matrix is not to be stored in the computer memory, but within the RSTS-11 disk file system instead. Data stored in disk files external to the user program remain, even after the user leaves his terminal, and can be retrieved by name at a later session. Items within the file are individually addressable. In fact, it is the similar treatment of arrays both in memory and on random-access files that leads to the term, "virtual array."

With the virtual array facility, BASIC-PLUS programs can operate on data structures that are too large to be accommodated in memory at one time. The disk file system is used for storage of data arrays, and only portions of these files are maintained in core at any given time.

With virtual data storage, the user can reference any element of one or more matrices within the file, no matter where in the file that element resides. This random access of data allows the user non-sequential referencing of the data for use in any BASIC statement. The virtual memory matrices are read into memory automatically by the system.

The order in which array elements are referenced can have a significant effect on the program execution time. This section therefore describes the algorithms used in the virtual array processor, in order that users concerned with efficiency can optimize their use of this facility.

Each disk file appears to the user program as a contiguous sequence of 512-byte records. Any position in a file can be specified internally with a 2-component address; the first part being the relative record within the file, and the second being the position of the item within the block. One of the functions of the virtual array processor is to transform, or map, each virtual array reference into its corresponding file address.

Virtual arrays are stored as unformatted binary data. This means that no I/O conversions (internal form-to-ASCII) need to be performed in storing or retrieving elements in virtual storage. Thus, there is no loss of precision in these arrays and no time wasted performing conversions.

11.1 VIRTUAL MEMORY DIM STATEMENT

In order for a matrix of data to exist in virtual memory, it must be declared in a special form of the DIM statement. This special DIM statement is as follows:

DIM# <integer constant>,<list>

where the integer constant is between 1 and 12 and corresponds to the internal file designator on which the program has opened a disk file. The variable list appears as it would in a DIM statement for a matrix in main memory. Thus, a 100-by-100 matrix could be defined as:

```
100     DIM #12%, A(100,100)
```

Floating-point constants, integer constants and strings can be stored in virtual memory matrices. More than one matrix can be specified in one virtual memory file. For example:

```
    250        DIM #1%, A(1000), B%(2000), C$(2500)
```

allocates space for 1000 floating-point numbers, 2000 integer numbers and 2500 character strings (16 characters long). However, if a virtual array is defined in this fashion, future references should always dimension the arrays to the same size.

11.2 VIRTUAL ARRAY STRING STORAGE

One of the few differences in data handling between memory and disk matrices occurs in the storage of strings within string matrices in virtual memory. Strings in the computer memory are of variable length from zero characters to any arbitrary length. Strings in virtual memory matrices are allocated with a maximum length and may vary from zero characters to the specified maximum length (all elements of a single string array have the same maximum length). This maximum length can be defined by the program and varies from two characters to 512 characters. The system requires the maximum length to be one of the following powers of 2:

2, 4, 8, 16, 32, 64, 128, 256, 512

Each element in the virtual memory string need not use the maximum length available, even though space is reserved for each element to be the maximum size. If the user indicates other than one of the values above, he receives the next higher size. Thus:

```
    100        DIM #1%, X$(10) = 65
```

is equivalent to:

```
    100        DIM #1%, X$(10) = 128
```

If no length is specified, a default length of 16 characters is assumed. The maximum length of virtual memory strings is specified as an expression in the DIM statement, using the form:

DIM# <integer constant>,<string>(<dimension>{, <dimension>}) {= <integer constant>}

For example:

```
    150        DIM #1%, A$(100)=32%, B$(100)=4%, C$(100)
```

where:

A$ consists of 101 strings of 32 characters each, maximum.
B$ consists of 101 strings of 4 characters each, maximum.
C$ consists of 101 strings of 16 characters each, maximum.

If a length attribute is given in a DIM statement for a memory string matrix, it is ignored, since memory is allocated dynamically to hold a string of any length.

11.3 OPENING AND CLOSING A VIRTUAL MEMORY FILE

In order for the user to reference his virtual core file, he must first associate a disk file (by name) with an internal channel designator from 1 to 12 (which is also used in the virtual DIM declaration). This is done with an OPEN, OPEN FOR INPUT, or OPEN FOR OUTPUT statement:

$$\text{OPEN <string>} \left\{ \begin{array}{l} \text{FOR INPUT} \\ \\ \text{FOR OUTPUT} \end{array} \right\} \text{AS FILE <expression>}$$

where the string is the name of a disk file and the expression specifies an internal file designator (this is the same format described in Section 9.2); thus:

```
350       OPEN 'ACCT' AS FILE 1%
```

associates the file named ACCT with internal channel 1. If ACCT already exists, then the existing file is used. If there is no file named ACCT, one would be created. If the user wishes to destroy an old file named ACCT and create a new file of the same name, he can use the statement:

```
350       OPEN 'ACCT' FOR OUTPUT AS FILE 1%
```

which causes the file to be deleted if it already exists and a new file created. If the user wants to be alerted that the file ACCT is not present, he could write:

```
350       OPEN 'ACCT' FOR INPUT AS FILE 1%
```

which would cause an error message to be printed if ACCT is not found.

NOTE

Virtual memory arrays do not permit internal buffers larger than 512 characters; therefore, the RECORDSIZE option is not used when opening a virtual core array file.

11.3.1 Pre-Extending a Virtual Array

The system overhead for extending a file by a single data element and by many elements is nearly the same. Thus it is much more efficient to extend a newly created file immediately to its final length than to extend it many times in increments of a single data element. Whenever the eventual size of a file is known, the file should be extended to its full size in a single operation.

For example:

```
100       OPEN 'DATA' FOR OUTPUT AS FILE 1%
110       DIM#1%, A(10000%)
120       A(10000%)=0.
```

This extends the virtual array A to its final length. Virtual memory arrays, however, are not initially zeroed by the system. In the example given above, A(0) through A(9999) contain indeterminate values. Unless the user is careful these values could cause a program failure. The user is advised to first zero the virtual memory array. This could be done as follows:

```
300       A(I%) = 0.0 FOR I% = 0% TO 10000%
```

which both zeros and extends the file to its maximum size. However, this uses the more time-consuming method of extending the file. A more optimal approach would be:

```
    300       A(I%) = 0.0 FOR I% = 10000% TO 0% STEP -1%
```

which immediately extends the file to its maximum and then zeros it sequentially. These techniques have frequent practical application.

11.3.2 Closing a Virtual Array File

The CLOSE statement must be used to terminate I/O between the BASIC program and the virtual array. Once a virtual array has been closed, it can be reopened for reading or writing on any internal file designator.

All virtual arrays must be closed before the end of program execution. The CLOSE statement causes the output of the last data element to a virtual memory file. Execution of a CHAIN statement automatically closes any open arrays, but does not cause the output of the last data elements to the array. The format of the CLOSE statement is as follows.

CLOSE <expression>{,<expression> . . .}

The expression indicated has the same value as the expression in the OPEN statement and indicates the internal channel number of the array to close. Any number of arrays can be closed with a single CLOSE statement; if more than one array is to be closed, the expressions are separated by commas. The CLOSE statement writes the current contents of the I/O buffer to virtual memory before closing it. This frees memory space for the program to open other arrays or files (a maximum of 12 depending upon available space). For example:

```
    255       CLOSE 2%,4%
    345       CLOSE 10%
```

Line 255 above closes the virtual arrays opened on internal I/O channels 2 and 4. Line 345 closes the array open on internal I/O channel 10.

11.4 VIRTUAL ARRAY PROGRAMMING CONVENTIONS

Recoverable errors occur when using virtual memory arrays if the user program does any of the following:

1. References a virtual array without first opening the file.
2. References a non-disk file (for example, DECtape or the line printer) as a virtual array.
3. Exceeds virtual memory, that is, defines a matrix that is bigger than the amount of available disk storage on the system.

Remember that a virtual memory file must be closed before stopping the program or chaining to another program.

11.4.1 Array Storage

Any data element in a virtual array is completely contained within a single block (512 bytes) of disk storage. This restriction has no effect on integers and floating-point items, where the size of data items is fixed, but does limit the maximum length of a virtual string to 512 characters. The number of data elements stored in each disk block is a function of the size of each element. For virtual strings, the number of elements is also related to the maximum string length specified in the DIM# statement. The size of a virtual string defaults to 16 characters and can be specified as: 2, 4, 8, 16, 32, 64, 128, 256, or 512. Table 11-1 indicates the number of array elements stored in each block of a virtual file.

Table 11-1
Virtual Array Storage Capabilities

Data Type	Number of Elements per Block
Integer (%)	256
2-Word Floating Point	128
4-Word Floating Point	64
String ($) (where the maximum length = N)	512/N

Strings in virtual memory occupy pre-allocated space in the virtual file, and thus differ from strings in memory, where space is allocated dynamically. A disk block containing virtual strings can be considered to be a succession of fields, each of the maximum string length. When a virtual string is assigned a new value, it is stored left-justified in the appropriate field. If the new string value is shorter than the maximum length, the remainder of the field is filled with 0's. When the string is retrieved, its length is computed as the maximum string length minus the number of zero-filled bytes.

11.4.2 Translation of Array Subscripts into File Addresses

In order to translate an array subscript into a file address, RSTS/E computes (1) the relative distance from the specified item to the first item in the array, and then adds (2) the relative distance from the first element of the array to the first item in the file. The first quantity (1) is computed from the array subscript and the number of elements per block, as shown in Table 11-1. The second number (2) is a constant for each array in a file, and is computed from the parameters specified in the DIM# statement.

Since the DIM# statement contains the only information used to define the structure of a file, it is possible for the user to specify different accessing arrangements for the same file in one or more programs. For example, the user can reference the same data as either a series of 32-byte strings (A2$) or 16-byte strings (A1$), with the following statements:

```
10      DIM #1,A1$(1000) = 16           !16-CHARACTER STRINGS
20      DIM #1,A2$(500) = 32            !32-CHARACTER STRINGS
30      OPEN 'FIL1' AS FILE 1%          !VIRTUAL ARRAY FILE
```

The user should keep in mind that in BASIC-PLUS array subscripts begin with 0, not 1. An array with dimension n, or (n, m) actually contains n+1, or [(n+1)*(m+1)] elements.

User programs may define 2-dimensional virtual arrays as well as singly dimensioned ones. Two-dimensional arrays are stored on disk (and in memory) linearly, row-by-row. Thus, in the case of an array X(1, 2), the array appears logically as:

X(0, 0)	X(0, 1)	X(0, 2)
X(1, 0)	X(1, 1)	X(1, 2)

while physically it is stored as:

X(0, 0)	lowest address
X(0, 1)	
X(0, 2)	
X(1, 0)	
X(1, 1)	
X(1, 2)	highest address

If a virtual array is to be referenced sequentially, it is preferable to reference the rows, rather than the columns, in sequence. Consider the case in which it is necessary to compute the sum of each row and column in a 2-dimensional array. Program 1 does this far more efficiently than program 2.

Program 1

```
LISTNH
100      REM - PROGRAM 'ONE' TO COMPUTE SUMS EFFICIENTLY
                 'ARRAY' CONTAINS VIRTUAL ARRAY
                 R(I%) IS SUM OF ROW I
                 C(J%) IS SUM OF COLUMN J
110      DIM #1%,A(10%,50%)
         ! 10 ROWS, 50 COLUMNS
120      DIM R(10%), C(50%)
130      OPEN 'ARRAY' AS FILE 1%
         ! OPEN VIRTUAL FILE AND INITIALIZE SUMS WITH MAT
140      MAT R = ZER
         \ MAT C = ZER
150      FOR I% = 1% TO 10%
         ! OPERATE ROW-BY-ROW
160      FOR J% = 1% TO 50%
         ! DO EACH COLUMN IN ROW
170      R(I%) = R(I%) + A(I%,J%)
         ! TOTAL ACROSS ROW
180      C(J%) = C(J%) + A(I%,J%)
         ! TOTAL DOWN COLUMN
190      NEXT J%
         \ NEXT I%
         ! COLUMN SUM IS INSIDE LOOP
200      MAT PRINT R;
         \ MAT PRINT C;
         ! PRINT ROW TOTALS, THEN COLUMN TOTALS
210      CLOSE 1%
32767    END

Ready
```

Program 2

```
LISTNH
100      REM - PROGRAM 'TWO' USES VIRTUAL MEMORY INEFFICIENTLY
110      DIM #1%,A(10%,50%)
         ! 10 ROWS, 50 COLUMNS
120      DIM R(10%), C(50%)
130      OPEN 'ARRAY' AS FILE 1%
         ! OPEN VIRTUAL FILE AND INITIALIZE SUMS WITH MAT
140      MAT R = ZER
         \ MAT C = ZER
150      FOR J% = 1% TO 50%
         ! OPERATE ONE COLUMN AT A TIME
160      FOR I% = 1% TO 10%
         ! AND ACROSS ROW
170      R(I%) = R(I%) + A(I%,J%)
         ! TOTAL ACROSS ROW
180      C(J%) = C(J%) + A(I%,J%)
         ! TOTAL DOWN COLUMN
190      NEXT I%
         \ NEXT J%
200      MAT PRINT R;
         \ MAT PRINT C;
         ! PRINT ROW TOTALS, THEN COLUMN TOTALS
210      CLOSE 1%
32767    END

Ready
```

In virtual arrays two (or more) arrays can share the same file. That is, the following DIM# statement is legal.

```
100      DIM #1, A(1000), B%(999), C(1000)
```

The matrix B% begins immediately after the 1000th element of A and the matrix C begins immediately after B%(999). Therefore, the disk layout is as shown in Figure 11-1.

There is, however, an exception to this rule. Elements in string arrays are allocated a fixed number of bytes in the disk file. This is either 2, 4, 8, 16, 32, 64, 128, 256 or 512 bytes of storage. A single string element must not cross a disk block boundary (where each disk block contains 512 bytes or 256 words). Consider the following case:

```
100      DIM A%(2), B$(1000)=4
```

The first three words of the disk block are allocated to A%. If the arrays B$ were to begin immediately after A%, one of the elements of B$ would cross a block boundary. Hence, B$ begins at the start of the second block in the file rather than immediately after A%.

The rule can be stated as follows: When more than one array is assigned to a single virtual array file, each array begins immediately following the last element of the preceding array unless such an allocation would cause an element of the array to be split across two disk blocks, in which case the array begins at the start of the next block of the file, and the remaining words of the current block are unused.

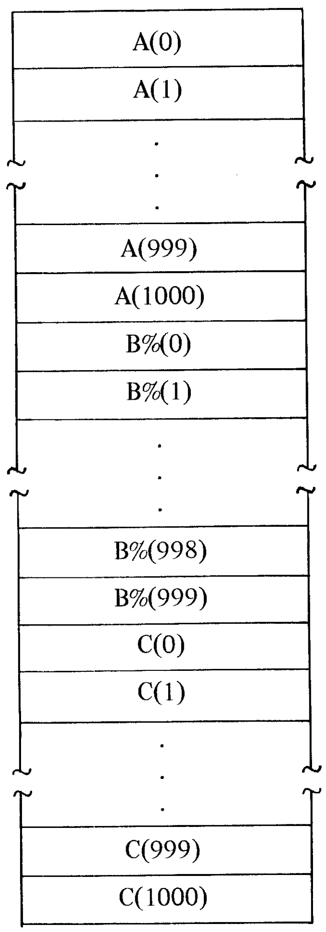


Figure 11-1 Virtual Array File Layout

11.4.3 Access to Data in Virtual Arrays

Only a portion of a virtual array is in memory at any given time. This data is transferred directly between the disk and an I/O buffer in the user's area, created when the OPEN statement is executed. This buffer must be 512 bytes (one block) long, and may not be specified as several blocks with the RECORDSIZE option in the OPEN statement. For each virtual array file, RSTS/E notes (1) the block of the file in the buffer, and (2) whether the data in the buffer has been modified since it was read into memory.

After RSTS/E translates a virtual array address into a file address, it checks whether the block containing the referenced item is currently in the buffer. If the necessary block is present the reference proceeds; but if not, another portion of the file is read into the buffer. If the current data in the buffer has been altered, it is necessary to rewrite this data on the disk prior to reading new data into the buffer.

The referencing algorithm, which minimizes the number of disk memory accesses generated when handling virtual arrays, is flow-charted in Figure 11-2.

All references to virtual arrays are ultimately located via file addresses relative to the start of the file. No symbolic information concerning array names, dimensions, or data types is stored within the file. Thus, different programs may use different array names to refer to the data contained within a single virtual array file. The user must be cautious in such operations, since it is his responsibility to ensure that all programs referencing a given set of virtual arrays are referencing the same data. Consider the following example:

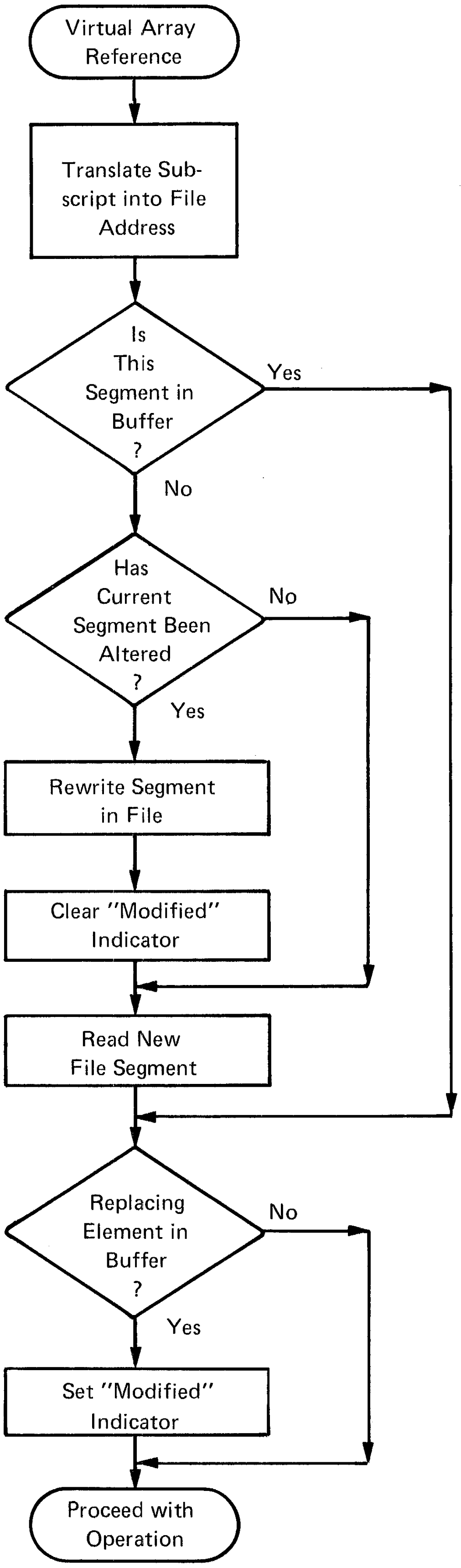

Virtual Array Reference
Translate Subscript into File Address
Is This Segment in Buffer ?
Yes
No
Has Current Segment Been Altered ?
No
Yes
Rewrite Segment in File
Clear "Modified" Indicator
Read New File Segment
Replacing Element in Buffer ?
No
Yes
Set "Modified" Indicator
Proceed with Operation

Figure 11-2 Virtual Array Addressing Algorithm

Program 1 contains:

```
100     !PROGRAM ONE
110     DIM #1, X(10), Y(10)
120     OPEN 'FILE' AS FILE 1%
```

Program 2 contains:

```
100     !PROGRAM TWO
110     DIM #1, Z(10), X(10)
120     OPEN 'FILE' AS FILE 1%
```

whenever program 2 references the array Z, it is using the data known to program 1 as array X. Both X and Z are the first arrays in their declaration, both contain floating-point data, and both are 11 elements (X(0), . . . , X(10)) long, These two arrays, then, correspond in position, type, and dimension.

References to the array X (in 1) and to the array X (in 2) do not refer to the same data, even though both are using the same virtual file (FILE). (The concept of using relative position, rather than name, to identify data items is familiar to users of the Fortran COMMON facility.)

Within a single BASIC-PLUS program it is possible to open a single virtual core array file twice on the same channel for the purpose of reallocating the data within the file. For example:

```
100     OPEN 'DATA' FOR INPUT AS FILE 1%
110     DIM #1, A$(10%)=4%
120     DIM #1, B$(4%)=16%
```

The program now has access to the file DATA through both the array A$ and the array B$. Each element of B$ contains four elements of A$ (B$(0) is equivalent to the elements A$(0) through A$(3), etc.). Note that the two DIM# statements reference that file on a single channel number (#1 in this case).

Note also that the two statements:

```
100     DIM #1, A(10%)
110     DIM #1, B(10%)
```

are not equivalent to the statement:

```
100     DIM #1, A(10%), B(10%)
```

In the first case the arrays A and B are equivalent to each other and constitute the first array in the file open on channel 1. In the second case the arrays A and B are defined as both existing in the file open on channel 1.

NOTE

The user is advised not to open a single file under two different channel numbers. For example:

```
150     OPEN 'VALUES' AS FILE 1%
        \ OPEN 'VALUES' AS FILE 2%
                •
                •
                •

200     DIM #1%, X$(20%)
210     DIM #2%, Y$(20%)
```

causes two buffers to be created for the storage of input to/from channel 1 and to/from channel 2. If changes are made to the same block of the file in both buffers, only the changes made in one of the buffers is added to the file. The buffer written first is overwritten by the other buffer; consequently any data changed in the first buffer is lost.

11.4.4 Allocating Disk Storage to Virtual Files

The dimensions indicated in a DIM# statement set maximum allowable values for subscripts, and are not used to compute the initial size of the virtual file to be allocated on disk. Instead, the file is created with an initial length of zero blocks, and blocks are appended to the file to accommodate the highest referenced file address in the array. This permits a user to specify array dimensions larger than required at the time the program is written; such programs may eventually operate on larger arrays without modification, and without tying up disk storage unnecessarily.

Areas of unallocated disk storage are found only at the end of a file.

As blocks are appended to a file, their contents are not initialized to zero. The data previously recorded in a block (when it was part of another file) is available to the new owner of the block. Users whose files contain confidential information should explicitly overwrite all data in such files, prior to file deletion. In order to protect data contained therein.

To override the dynamic virtual array allocation, the user can reference the last element in the virtual array file. This causes all blocks in the file, up to and including the last, to be allocated. As noted above, the contents of these blocks as appended to the file are unknown. Using the MAT ZER command is advisable if the program depends on array values being initialized to a known (zero) quantity.

11.4.5 Simultaneous Access of a Virtual Memory Array by Several Programs

As mentioned in Section 9.2, only the first program to open a file (array) is given write privileges. When a second program attempts to modify an array which is already open, the appropriate block is read from the disk but changed only in the second user's buffer – not on the disk. When the second program references this array and attempts to read another block from the disk, a PROTECTION VIOLATION error occurs. This is because the system attempts to update the disk with the new information in the current block before the required block is read into memory. Since the second program has no write privileges, the disk cannot be updated. A CLOSE operation at this point also results in a PROTECTION VIOLATION error for the same reason. Once the job returns to BASIC-PLUS command level and a NEW, OLD or RUN command is executed, a CLOSE is performed on all channels. In this case, no write is attempted so the CLOSE is successful.

The best way to avoid simultaneous write accessing of a virtual core array is to determine whether the user program has write privileges. Do this with the STATUS variable (see Section 12.3.5) as shown below.

```
100       OPEN 'ARRAY' AS FILE 1%
110       IF (STATUS AND 1024%)
                    THEN PRINT 'NO WRITE ALLOWED ON ARRAY'
                    \         STOP

Ready
```

MODE 1% should not be used for updating an array by several programs simultaneously. This is because a user's buffer is modified when an array is opened with the MODE 1% option – the disk is not updated at this time. (Even when the first program unlocks the file, allowing other programs to access the array, the first program's modifications exist only in the first user's buffer.) The array is updated only when the first user accesses data from another block, as explained above.

11.5 PROGRAMMING EXAMPLE

As an example of virtual core usage, consider the problem of generating a large array of random numbers. Since a physical disk block is 256 words, the most efficient array would contain a multiple of 256 elements. The virtual core file, ARRAY1.DAT, in this example, contains 5120 data elements in a 2-by-2560 array. The zero row and zero column are used, so this array is dimensioned V%(1%,2559%). Twenty physical blocks are used to store this array. The program shown below creates the virtual array V% by assigning a random value between 0 and 1000 to each element in the array.

```
LISTNH
1000      OPEN 'ARRAY1.DAT' AS FILE 3%
1010      DIM #3%, V%(1%,2559%)
1020      FOR I% = 0% TO 1%
1030      V%(I%,J%) = RND(1) * 1000.% FOR J% = 0% TO 2559%
1040      NEXT I%
1050      CLOSE 3%
32767     END

Ready
```

Now that the file ARRAY1.DAT has been created, the virtual array can be accessed simply by specifying the elements by their subscripts. The program shown below prints every 256th value. Notice that the format of the array in the DIM statement, below, must be identical to the original format for predictable results. The file's internal channel number and the array's name can change, but the array must be formatted the same way every time it is accessed.

```
LISTNH
1000      OPEN 'ARRAY1.DAT' AS FILE 3%
1010      DIM #3%, V%(1%,2559%)
1020      FOR I% = 0% TO 1%
1030      PRINT V%(I%,J%); FOR J%=0% TO 2559% STEP 256%
1040      NEXT I%
1050      CLOSE 3%
1060      PRINT
32767     END

Ready

RUNNH
 204   909   954   839   65   131   537   784   371   798   565   173   122   910   39
  8   318   468   958   289

Ready
```

Values of array elements can be changed simply by redefining them in assignment statements (e.g., LET, INPUT, READ). For example, the program below changes the value of specified data elements, once they are defined by subscripts.

```
LISTNH
1000     OPEN 'ARRAY1.DAT' AS FILE 3%
1010     DIM #3%, V%(1%,2559%)
1015     ON ERROR GOTO 1050
1020     INPUT 'ENTER THE I AND J LOCATION OF THE ELEMENT'; I%,J%
1030     N% = V%(I%,J%)
1040     INPUT 'ENTER THE NEW VALUE'; V%(I%,J%)
1045     PRINT
         \ PRINT 'OLD VALUE WAS: ' ;N%; '  NEW VALUE IS: ';V%(I%,J%)
         \ PRINT
         \ GO TO 1020
1050     CLOSE 3%
32767    END

Ready

RUNNH
ENTER THE I AND J LOCATION OF THE ELEMENT? 0,9
ENTER THE NEW VALUE? 600

OLD VALUE WAS:  678   NEW VALUE IS:  600

ENTER THE I AND J LOCATION OF THE ELEMENT? 1,255
ENTER THE NEW VALUE? 333

OLD VALUE WAS:  937   NEW VALUE IS:  333

ENTER THE I AND J LOCATION OF THE ELEMENT? 0,2225
ENTER THE NEW VALUE? 9999

OLD VALUE WAS:  424   NEW VALUE IS:  9999

ENTER THE I AND J LOCATION OF THE ELEMENT? 9,9

Ready
```

Some thought should be given to access methods of virtual arrays. In the above examples, ARRAY1.DAT was allocated as follows:

Block 1	V(0, 0) – V(0, 255)
Block 2	B(0, 256) – V(0, 511)
Block 3	V(0, 512) – V(0, 767)
.	.
.	.
.	.
Block 10	V(0, 2304) – V(0, 2559)

```
Block 11            V(1, 0) – V(1, 255)
   .                         .
   .                         .
   .                         .
Block 20            V(1, 2304) – V(1, 2559)
```

Notice that the second subscript varies from 0 to 2559 for each of the two values (0 and 1) of the first subscript. Since the system transfers an entire physical record (that is, a block) from the disk to memory at one time, only one disk access is performed for each 256 consecutive data elements (e.g., V(0, 256) – V(0, 511)). It is far more efficient to access data elements within a given block than to access data elements in different blocks.

The two programs shown below access, but do not print, each element in the virtual array. The first access method transfers a new block to memory for each data element accessed, resulting in 5, 120 disk accesses. The second method, howevery, transfers a new block to memory only once per 256 data elements, resulting in only 20 disk accesses. The difference in execution time between both methods is quite significant, as shown below.

Program 1 (Inefficient)

```
LISTNH
1000     OPEN 'ARRAY1.DAT' AS FILE 3%
1010     DIM #3%, V%(1%,2559%)
1015     T = TIME(0)
1020     FOR J% = 0% TO 2559%
1030     D% = V%(I%,J%) FOR I% = 0% TO 1%
1040     NEXT J%
1045     PRINT 'THIS ACCESS TOOK ' TIME(0) - T 'SECONDS.'
1050     CLOSE 3%
32767    END

Ready

RUNNH
THIS ACCESS TOOK  422 SECONDS.

Ready
```

Program 2 (Efficient)

```
LISTNH
1000     OPEN 'ARRAY1.DAT' AS FILE 3%
1010     DIM #3%, V%(1%,2559%)
1015     T = TIME(0)
1020     FOR I% = 0% TO 1%
1030     D% = V%(I%,J%) FOR J% = 0% TO 2550%
1040     NEXT I%
1045     PRINT 'THE SECOND ACCESS TOOK' TIME(0) - T ' SECONDS.'
1050     CLOSE 3%
32767    END

Ready

RUNNH
THE SECOND ACCESS TOOK 2  SECONDS.

Ready
```

CHAPTER 12
RECORD I/O

There are three methods of performing I/O operations in BASIC/PLUS. Formatted ASCII I/O is simple and flexible, but requires conversion of numbers by the system from an internal form to an externally usable ASCII representation and does not permit random access to files. I/O to virtual memory arrays permits high-speed random access to files but can be used only on disk files and does not allow true intermixing of string and numeric elements or use of the RECORD-SIZE specification.

The third type of I/O, Record I/O, permits the user program to have complete control of I/O operations. Properly used, Record I/O is the most flexible and efficient technique of data transfer available under BASIC-PLUS. It does, however, sacrifice the simplicity of formatted ASCII and virtual array I/O. Less experienced users should first experiment with the simpler I/O techniques before attempting Record I/O.

12.1 OPENING A RECORD I/O FILE

Opening a file for Record I/O requires an OPEN statement, described in Section 9.2. The format of the OPEN statement is as follows:

```
              ( FOR INPUT  )
OPEN<string> <              >  AS FILE<expr>
              ( FOR OUTPUT )

        {,RECORDSIZE<expr>} {,CLUSTERSIZE<expr>}

        {,FILESIZE<expr>} {,MODE<expr>}
```

The RECORDSIZE and CLUSTERSIZE options can be specified for Record I/O files as described in Sections 9.2.1 and 9.2.2.

The use of all optional clauses depends on the particular device being accessed. The optional clauses are described extensively in the RSTS/E Programming Manual.

12.2 CLOSING A RECORD I/O FILE

Each Record I/O file must be closed once I/O operations on that file are completed. Files are closed with the CLOSE statement, as described in Section 9.3. The CLOSE statement is of the form:

```
CLOSE<expr>{,<expr>}
```

where the value of each expression specifies one of the 12 I/O channels.

Remember, the CLOSE statement for formatted ASCII and virtual array files causes the final record of the file to be written before closing the file. However, all I/O operations to Record I/O files are explicitly performed (with GET and PUT statements). For this reason, be sure the user program explicitly writes the last record onto a Record I/O file before executing a CLOSE.

12.3 THE GET AND PUT STATEMENTS

Input and output operations to Record I/O files are performed directly between the device channel and the I/O buffer created by the OPEN statement. All I/O is specified in terms of single records, using the GET and PUT statements. GET and PUT are of the form:

$$\begin{matrix} \text{GET\#<expr1>} \\ \text{PUT\#<expr1>} \end{matrix} \left\{\text{,RECORD<expr2>}\right\} \left\{\text{,COUNT<expr3>}\right\}$$

If the RECORD option (see Section 12.3.3) is not used, the GET statement reads the next sequential record from the file open on the channel designated by <expr1>. The record is placed in the I/O buffer that was associated with the channel by the OPEN statement. The size of the record depends upon the characteristics of the device on which the file resides, as described in Table 12-1. In Record I/O, the RECORD option refers to a sector whose length is device-specific, not to a logical data record.

When the RECORD option is used in a GET or PUT statement, a specific record, or sector, is accessed. For example:

```
100     GET #4%, RECORD 8%
```

reads the eighth sector of the file opened on channel 4 into the user I/O buffer. Notice that the preceding seven sectors of the file need not be read.

Table 12-1
Device Record Characteristics

Device	Input Record Characteristics
disk	Records (sometimes called blocks or segments) are normally 512 bytes long. When the RECORDSIZE option is specified in the OPEN statement, and a buffer longer than 512 bytes is created, the system reads as many full records as possible. If several disk blocks are read with a single GET statement, the next sequential record is that record immediately following the last block read. RECORDSIZE must be an even number. If RECORDSIZE is not a multiple of 512 bytes, the last block in the transfer is only partially transferred. The remainder of the block is discarded. In non-file-structured operation, the default record size is dependent on the device cluster size.
DECtape	For file-structured DECtape, records are always 510 bytes long. For non-file-structured DECtape, records are always 512 characters.
magtape	When performing file-structured I/O, magtape records are normally 512 characters. With non-file-structured I/O, magtape records can be of any length; only one record can be read per GET statement; and the record length can not exceed the buffer size as determined by the RECORDSIZE option.
keyboard	The GET #0 statement (or a GET on any channel associated with the keyboard) obtains one line from the keyboard, up to the first line delimiter (CTRL/Z, RETURN, LINE FEED, ESCAPE, FORM FEED or CTRL/D).
card reader	A record consists of a single card. The RECORDSIZE option has no effect on card reader input.
paper tape	RSTS/E reads a full buffer of input from the paper tape reader unless an end-of-tape is detected.

Similarly, if the COUNT and RECORD (see Sections 12.3.2 and 12.3.3) options are not used, the PUT statement writes the contents of the I/O buffer for the specified I/O channel onto the next sequential record of the file. The expression <expr1> specifies the internal channel number on which the file was opened. The PUT statement writes a single record on the device, with the exception of disk files which permit several records to be written with one PUT statement (when the RECORDSIZE option in the OPEN statement is used to increase I/O buffer size).

To avoid having to move the contents of an input buffer to an output buffer (as when copying from one file to another, for example) the alternate buffer I/O technique is recommended. In this technique, <expr1> of the GET and PUT statements is as follows:

SWAP%(B%) + I%

where B% is the channel number of the buffer to be used, and I% is the channel number on which the I/O activity occurs.

The following example shows a fast copy, using the alternate buffer technique.

```
100     ON ERROR GOTO 9000
110     INPUT 'ENTER INPUT CHANNEL NUMBER'; I%
        \ INPUT 'ENTER OUTPUT CHANNEL NUMBER'; O%
120     OPEN 'INPUT' FOR INPUT AS FILE I%
130     OPEN 'OUTPUT' FOR OUTPUT AS FILE O%
140     GET #I%
        \ PUT #SWAP%(I%) + O%
        \ GOTO 140
160     CLOSE #I%, #O%
        \ PRINT 'FILE COPIED'
        \ GOTO 32767
9000    IF ERR=11% AND ERL=30%
                THEN RESUME 160
                ELSE ON ERROR GOTO 0
        ! ACCEPT END-OF-FILE ON INPUT
32767   END
```

12.3.1 The RECOUNT Variable

Non-file-structured devices, as can be seen in the description of the GET statement, can read less than a full buffer of data. To permit the program to determine how much data was actually read, a system variable, RECOUNT, contains the number of characters read following every input operation. RECOUNT is used primarily for non-file-structured input; however, it may also be used with file-structured devices.

RECOUNT is set by every input operation on any channel (including channel 0). It is, therefore, essential that the RECOUNT value be tested or copied immediately following the GET statement. RECOUNT is not properly set if any error occurs in the operation.

12.3.2 The COUNT Option

The COUNT option used in a PUT statement with a non-file-structured device specifies the number of characters to write in the current record. However, the COUNT expression cannot be greater than the size of the I/O buffer.

For example, where internal channel 1 is opened as magtape unit 0 (non-file-structured magtape), the following statement could be used to write an 80-character record:

```
100     PUT #1%, COUNT 80%
```

When COUNT is not used, the PUT statement writes an entire buffer, regardless of whether the buffer contains data.

COUNT must be used when writing a disk file that was opened with RECORDSIZE not a multiple of 512.

12.3.3 The RECORD Option

With disk files, the user has the capability of performing random access I/O to any record of the file. Records in a disk file are always 512 characters long and are logically numbered within the file from 1 to n, where n is the size of the file.

The RECORD expression provides the logical record number of the file to the GET or PUT statement. For example, assuming a disk file opened on internal channel 1, the following statement writes the contents of the I/O buffer associated with channel 1 on records 10 through 99 of that disk file:

```
200     PUT #1%, RECORD I% FOR I%=10% TO 99%
```

More than one physical record or block can be read or written by assigning a large I/O buffer to the file with the RECORDSIZE option in the OPEN statement. (The size of the buffer does not affect the numbering of the records within the file.)

If the disk file on channel 1 were opened with a RECORDSIZE of 1024 characters (which would cause two 512-character records to be written with each PUT) the PUT statement would be written as follows:

```
200     PUT #1%, RECORD I% FOR I% = 10% TO 98% STEP 2%
```

After performing a random access GET or PUT on a disk file, the next GET or PUT statement on that channel accesses the next sequential record if no RECORD number is specified. For example:

```
290     OPEN 'DATA' AS FILE 1%, RECORDSIZE 512%
300     GET #1%, RECORD 99%
310     PUT #1%
```

The PUT statement at line 310 writes record 100 of the disk file.

12.3.4 BUFSIZ Function

In certain applications, it is important for a program to determine the buffer size of an open channel, especially if the OPEN statement specifies a logical device name. The user program can execute the integer function BUFSIZ to extract this information.

The BUFSIZ function returns an integer value telling the size of the buffer for a specified open channel. For example:

```
Y% = BUFSIZ(N%)
```

The statement returns to Y% the size of the buffer in number of bytes for channel N%. If the channel is closed, the function returns 0 to Y%.

12.3.5 STATUS Variable

The variable STATUS contains information concerning the last channel on which a user program executed an OPEN statement. The variable is a 16-bit word, each bit of which the user program can test to determine status. Table 12-2 shows the information, the tests, and the meaning of each bit.

Table 12-2
RSTS Variable STATUS

Bit	Test	Meaning
0-7	(STATUS AND 255%)	The first eight bits of the word contain the handler index. The following values apply for various devices.
		0 Disk — 12 Card Reader
		2 Keyboard — 14 Magtape
		4 DECtape — 16 PK: device (pseudo-keyboard)
		6 Line Printer — 18 DX: device (floppy disk)
		8 Paper Tape Reader — 20 RJ: device (2780 remote job entry)
		10 Paper Tape Punch — 22 Null device NL:
		24 DMC11
8	(STATUS AND 256%)<>0%	The device is open in non-file structured mode or is characteristically non-file structured.
9	(STATUS AND 512%)<>0%	The job does not have read access to the device.
10	(STATUS AND 1024%)<>0%	The job does not have write access to the device.
11	(STATUS AND 2048%)<>0%	The device maintains its own horizontal position. Such devices are keyboard and line printer type.
12	(STATUS AND 4096%)<>0%	The device accepts modifiers. Such devices use the record number as a modifier word rather than a physical position of the device. Keyboard, line printer, and card reader are such devices.
13	(STATUS AND 8192%)<>0%	Device is a character device.
14	(STATUS AND 16384%)<>0%	Device is an interactive type (keyboard).
15	(STATUS <0%)	Device is a random access blocked device, such as disk and non-file-structured DECtape.

12.4 WORKING WITH RECORD I/O FILES

Techniques for opening, closing, reading and writing Record I/O files have been described. But these techniques apply only to indivisible I/O buffers associated with internal channels; no mention has been made of manipulating data within these buffers. Techniques for moving data into or out of a buffer are provided by extensions to the BASIC language. The FIELD, LSET and RSET statements permit the program to access and modify the contents of an I/O buffer, character by character. These statements are discussed in the following sections.

12.4.1 Extending Disk Files

A disk file that is created by an OPEN FOR OUTPUT (or OPEN) statement has a length of 0. As records are written, the file progressively grows in length; this growth is called extending the file.

A more exact description of file extending is as follows:

1. Is there room in the last cluster[1] of the file for the new record?
2. If so, then the file length is increased and previously unused space in that cluster is used.
3. If not, then a new cluster is appended to the file. There is then room in the newest last cluster for the new record so condition 2 applies.

The amount of space actually allocated by the system to a file may be greater than the file length. For example, if the file clustersize is 4 and the first 6 records of that file have been written, the file is of length 6 but is actually allocated 8 blocks (2 clusters) of space.

A file is extended by attempting to write beyond the current end-of-file. Hence, a program must have write privileges in order to be able to extend a file. There is an exception to the rule that having write access to a file permits a program to extend the file. When a file is opened for update (see Section 12.4.5) several programs can have simultaneous write privileges on a single file. Nonetheless, if a program opens a file in update mode, that file can not be extended. A file can be extended only when open in normal (non-update) mode.

It is possible to extend a file by a number of records at one time. For example:

```
100      OPEN 'DATA' FOR OUTPUT AS FILE 1%
110      PUT #1%, RECORD 100%
```

creates a file DATA and (when line 200 is executed) extends it immediately to 100 records. Since the system overhead for extending a file by a single record and by many records is nearly the same, it is much more efficient to immediately extend a newly created file to its final length than to extend it many times in increments of a single record. Whenever the final size of a file is known, the file should be extended to its full size in a single operation.

12.4.2 The FIELD Statement

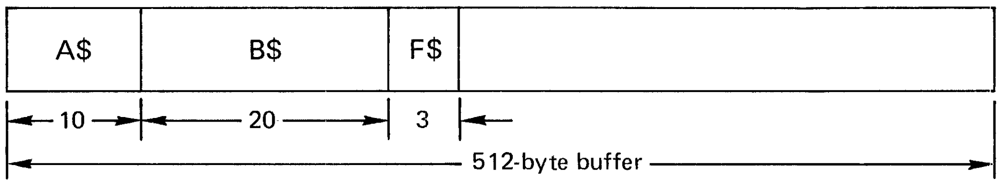
The FIELD statement is used to dynamically associate string names with all or part of an I/O buffer. The FIELD statement has the form:

FIELD #<expr>, <expr1> AS <stringvar1>

[,<expr2> AS <stringvar2> . . .]

where <expr> is an internal channel number associated with some file by an OPEN statement; <expr1> is the length, in characters, of the associated string variable; and <stringvar1> is a unique string variable name. The names are associated from left to right with successive characters in the I/O buffer assigned to the designated internal channel number. For example:

```
75       FIELD #2%, 10% AS A$, 20% AS B$, 3% AS F$
```



[1]The CLUSTERSIZE defines the minimum increment by which a file can be extended on the disk. A file need not occupy all blocks within the cluster.

As shown in the previous diagram, statement 75 associates three strings, A$, B$, and F$ in the I/O buffer, with lengths of 10, 20, and 3 characters, respectively. The total number of characters represented in this statement is 33. The total number of characters must be less than or equal to the actual I/O buffer size (which is dependent on the device and the RECORDSIZE option, as described in Section 9.2.1).

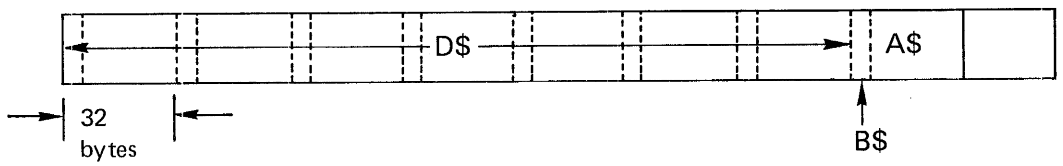
FIELD statements do not move data but rather permit direct access to sections of the I/O buffer via string variables. The effect upon a string variable is temporary and is nullified by any attempt to assign a value to the variable (other than the LSET and RSET, described in Section 12.4.3). For example:

```
100        OPEN 'FILE' AS FILE 2%
110        FIELD 3\3\#2%, 5% AS A$
120        LET A$ = 'ABCDE"
```

Line 120 causes the string variable A$ to be removed from the I/O buffer. The string ABCDE is not stored in the I/O buffer by line 120.

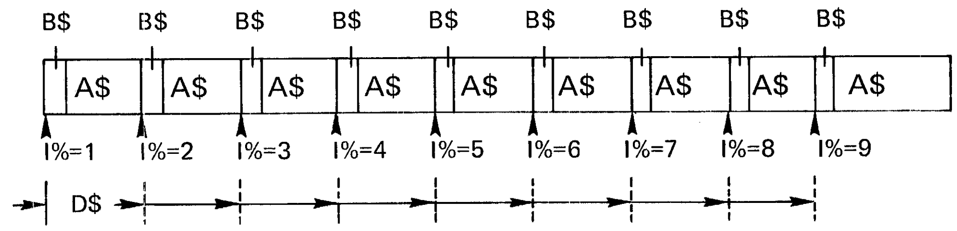
A FIELD statement is an executable statement, rather than a compiler directive (such as a DIM statement). To illustrate: suppose that each block of a disk file contains sixteen 32-character subrecords and that each record consists of one 5-character field and one 27-character field. In order to extract the eighth subrecord from the I/O buffer, the following statement could be executed:

```
200        FIELD #1%, 224% AS D$, 5% AS B$, 27% AS A$
```



Line 200 causes the string variables B$ and A$ to point to the desired subrecord. The string D$ is created to permit the first seven subrecords (7*32=224) to be skipped. An even more general statement could be used to obtain any of the subrecords in the I/O buffer, as follows:

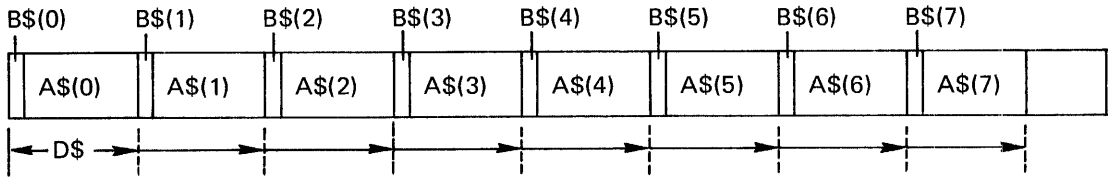
```
190        FOR I% = 0% TO 15%
200        FIELD #1%, (I%-1%)*32% AS D$, 5% AS B$, 27% AS A$
210        NEXT I%
```



When the statement above is executed, I% should contain the number of the subrecord that B$ and A$ are to contain, as an integer from 1 to 16. When I%=1, for example, the expression (I%- 1%)*32% equals 0, so B$ points to the first subrecord in the buffer. When I%=2, however, the expression (I%- 1%)*32% equals 32, so B$ now points to the first subrecord beyond the 32nd character of the buffer. Each single increment of I% moves B$ 32 characters further into the buffer.

Subscripted string variables can also be used in FIELD statements. For example, the following statements could be used to allocate the subrecords, described in the previous example, to two string arrays:

```
300     DIM A$(15%), B$(15%)
310     FOR I% = 0% TO 15%
320     FIELD #1%, I%*32% AS D$, 5% AS B$(I%), 27% AS A$(I%)
330     NEXT I%
```



With each iteration of the FIELD statement at line 320 the dummy string D$ increases by 32 characters, making the displacement from the start of the I/O buffer to the string B$(I%) equal to 32 times I% characters. Once this loop is executed, the position of each string in the arrays A$ and B$ is fixed, A$(0) and B$(0) pointing to the first subrecord and A$(15) and B$(15) to the last.

However, virtual array strings must not be defined as string variables in a FIELD statement. When strings are defined as virtual arrays they are required to be in a fixed place in both a disk file and the I/O buffer for that file. Attempting to specify a virtual array string variable in a FIELD statement has no effect on the virtual array string.

12.4.3 LSET and RSET Statements

Once the strings have been defined as part of the I/O buffer by a FIELD statement, values in these strings can be stored without moving them from the I/O buffer. The LSET and RSET statements store values in a string without redefining the string position. These statements are of the form:

LSET <stringvar> {,<stringvar> . . .} = <string>

RSET <stringvar> {,<stringvar> . . .} = <string>

where <stringvar> represents any legal existing string variable name (multiple string variable names can be separated by commas) and <string> represents any legal string expression.

The LSET and RSET statements store the value of the string expression into the designated string or strings. The string previously stored in the variable is overwritten. The length of the string is not changed; if the new string is longer than the existing string, the new value is truncated. If the new string is shorter than the existing string, it is either padded with spaces on the right by LSET or padded with spaces on the left with RSET. LSET, then, causes the string to be left-justified in the field and RSET causes the string to be right-justified.

The normal use of LSET and RSET, as described in this section, is to store data in strings allocated within an I/O buffer by a FIELD statement. LSET and RSET can be used to assign a value to any string variable within a BASIC-PLUS program.

12.4.4 Differences Between the LET Statement and the LSET/RSET Statement

The LET statement cannot be used to place string values into an I/O buffer because it causes the string to be redefined elsewhere. Another restriction on LET occurs when that statement is used to equate two strings, as follows:

```
100     LET A$ = B$
```

To avoid unnecessary character manipulation, this operation causes A$ and B$ to reference the same string in memory. Normally, any operation which alters B$ causes that string to be moved, so no conflict arises. However, LSET and RSET do not move strings; they alter existing strings in a fixed position.

Therefore, if the value of B$ in line 50 above were altered by an LSET or RSET statement, the value of A$ also changes. For example:

```
    400        B$ = 'ABC'
    410        A$ = B$
    420        LSET B$ = 'XYZ'
```

Both A$ and B$ contain "XYZ" following the execution of line 420.

This phenomenon has another ramification; if the string B$ in this example had been defined by a field statement as being in some I/O buffer, the string A$ would also be in the I/O buffer (being identical to B$). Executing a GET statement to read another record into the I/O buffer would then change the value of A$ as well as B$. For this reason, LSET and RSET should be used only for Record I/O operations; using these statements for other purposes may cause peculiar results.

When the strings A$ and B$ should not be physically identical, the string B$ can be moved into the string A$ as follows:

```
    300        LET A$ = B$ + ''
```

Line number 300 appends a null string to B$, which has no effect on the string A$ but causes the two strings to occupy different storage areas.

12.5 CVT CONVERSION FUNCTIONS

The FIELD, LSET, and RSET statements allow a program to store or retrieve string data directly from an I/O buffer. To permit floating-point and integer values in Record I/O files, four conversion functions are provided as described in Table 12-3. A fifth conversion function facilitates character string manipulation.

Four of the functions do not affect the value of the data, but rather its storage format. Each character in a string requires one byte of storage (8 bits); hence, characters may assume (decimal) values from 0 through 255 and no others. A 16-bit quantity can be defined as either an integer or a 2-character string; 2-word floating-point numbers can equally be defined as 4-character strings.

The CVT functions that change storage format perform two important functions: first, they permit dense packing of data in records. For example, any integer value between -32768 and 32767 can be packed in a record in two characters using CVT%$; this would only be true for integers between -9 and 99 if the data were stored as ASCII characters. Second, converting the internal numeric representation to an ASCII string (as with the NUM$ function) is a more time-consuming process than that performed by the CVT functions. Thus, the CVT functions speed the processing of a large amount of data within a file.

The **CVT$$** function manipulates a character string and generates a new character string. This action is unlike other CVT functions because it does not change the internal format of the data, but rather alters the contents of the string. The output string is converted according to an integer value given by the user program and can be any value or sum of any values listed in Table 12-3.

The value 1% in the CVT$$ function removes the parity bit (most significant bit) from each character in the string. Under RSTS/E, characters are usually represented with no parity. All comparison of characters assume no parity. The value 2% removes all space characters (CHR$(32)) and horizontal tab characters (CHR$(91)) from the string while values 8%, 16%, and 128% remove only selective occurrences of space and horizontal tab characters. The terminating and excess characters removed by the value 4% in the CVT$$ function usually have no informational value in a string.

Table 12-3
CVT Conversion Functions

Function Form	Operation
A$ = CVT%$ (I%)	Maps an integer into a 2-character string.
I% = CVT$% (A$)	Maps the first two characters of a string into an integer. If the string has fewer than two characters, null characters are appended as required.
A$ = CVTF$ (X)	Maps a floating-point number into a 4- or 8-character string (depending upon whether the 2-word or 4-word math package, respectively, is being used on the system). The current math package can be determined by examining LEN(CVTF$ (0)).
X = CVT$F (A$)	Maps the first four or eight characters (depending upon whether the 2-word or 4-word math package, respectively, is being used on the system) of a string into a floating-point number. If the string has fewer than the required number of characters, null characters are appended.
T$ = CVT$$ (S$,M%)	Converts the source character string S$ to the string referenced by the variable T$. The conversion is performed according to the decimal value of the integer represented by M% as follows: 1% Trim the parity bit. 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. 256% Do not alter characters inside quotes.

The value 32% converts all lower-case characters in a string to upper-case. This feature is valuable since some terminals transmit both forms of alphabetic characters. The lower-case characters are between CHR$(97) and CHR$(122) and upper-case characters are between CHR$(65) and CHR$(90).

The value 64% in the CVT$$ function enables BASIC-PLUS programs to accept the parenthesis and square bracket characters ad delimiters of a project-programmer number. This action is desirable when handling account numbers from terminals not having the square bracket characters since most terminal devices have the parenthesis characters.

The value 256% in the CVT$$ function forbids any alteration of characters inside quotes, except parity bit trimming -- set by M%=1%. Regardless of other values in the parameter M%, when 256% is included no operations are performed in the source string on characters within quotes.

Generally, the precedence of operations performed on the string is in increasing order of the individual values in the parameter M%. (The 256% value, however, is the exception; its precedence ranks between 1% and 2%.) This order implicitly determines which subsequent operations are performed on the string. For example, if the characters in the source string have their parity bit set and the parity trimming option is not selected, subsequent comparisons required by other options might not be successful because comparisons are made against ASCII characters with no parity. For example, a space (SP) character, which is CHR$(32) in no parity or odd parity form, does not compare with a space (SP) character which is CHR$ (160), its even parity form.

Keeping the parity bit in the input character of the string is important in text processing applications where the parity bit of each character is possibly a flag rather than a parity bit. As a result, such flagged characters are not changed or discarded if the parity trimming option is not selected.

The precedence of operations affects the result of values given in the CVT$$ function. If the values 2%, 8%, 16%, and 128% (154% or greater) are given in the CVT$$ function, the values 8%, 16%, and 128% have no effect on the output string since the first option performed (2%) removes all space and tab characters from the string and the remaining values dealing with space and tab characters have no effect. In like manner, the value 16% applies to all space and tab characters not discarded by the 2% and 8% options. Accordingly, to maintain at least a single space interval in a string, the user program must give the 16% value and omit the 2% and 8% values.

The use of the CVT$$ function in general eliminates the need for special code in BASIC-PLUS programs handling string input. For example, the following code at lines numbered 110 through 150 manipulates an input string.

```
LISTNH
50       DIM A6%(128%)
60       N1% = 1%
100      PRINT 'TYPE THE INPUT STRING';
         \ INPUT LINE A6$
110      T$ = FNC6$(A6$)
120      DEF FNC6$(A6$)
130      CHANGE A6$ TO A6%
         \ J6% = 0%
140      FOR X6% = N1% TO A6%(0%)
         \ IF A6%(X6%) <= 32%   OR   A6%(X6%) > 93%
                 THEN GOTO 150
                 ELSE J6% = J6% + N1%
         \ A6%(J6%) = A6%(X6%)
150      NEXT X6%
         \ A6%(0%) = J6%
         \ CHANGE A6% TO A6$
         \ FNC6$ = A6$
         \ FNEND
160      PRINT 'T$ = '; T$
         \ GO TO 100
32767    END

Ready
```

```
RUNNH
TYPE THE INPUT STRING?    DEV:    FILE.EXT   [100 ,   100    ]
T$ = DEV:FILE.EXT[100,100]
TYPE THE INPUT STRING? 'THE LIFE OFA\A\ A PEBBLE IS QUICK;
 LIKE UNTO THE ESSENCE OF A GREEN ARROW.'
T$ = 'THELIFEOFAPEBBLEISQUICK;LIKEUNTOTHEESSENCEOFAGREENARROW.'
TYPE THE INPUT STRING? ^C

Ready
```

Lines 110 through 150 can be replaced by a single CVT$$ function statement at line 110 as shown in the sample code below.

```
LISTNH
100      PRINT 'TYPE THE INPUT STRING';
         \ INPUT LINE A6$
110      T$ = CVT$$(A6$,7%)
160      PRINT 'T$ = '; T$
         \ GO TO 100
32767    END

Ready
```

The value 7% in the CVT$$ function is the sum of 1%, 2%, and 4%. The CVT$$ function with a value 7% causes the same results as the code of the user-defined function FNC6$. The following sample dialog shows the effect of the value 7% at line 110.

```
TYPE THE INPUT STRING?   DEV:   FILE .EXT   [100,1 0 0   ]
T$ = DEV:FILE.EXT[100,100]
TYPE THE INPUT STRING? ^C
```

The value 255% in the CVT$$ function at line 110 produces the results shown by the following sample dialog.

```
TYPE THE INPUT STRING? DEV    :  FILE   .EXT   [100,     100]
T$ = DEV:FILE.EXT(100,100)
TYPE THE INPUT STRING? ^C
```

The following sample dialog shows the effect of the value 189% (1%+4%+8%+16%+32%+128%).

```
RUNNH
TYPE THE INPUT STRING? HE  SAID,   "I   AM SURE
  I . . . DON'T    KNOW."
T$ = HE SAID, "I AM SURE I . . . DON'T KNOW."
TYPE THE INPUT STRING? ^C
```

12.6 EXAMPLES OF RECORD I/O USAGE

In Figure 12-1, the device KB: is opened with the default size (128 characters) buffer length by the OPEN statement at line 10.

```
LISTNH
10       OPEN "KB:" FOR OUTPUT AS FILE 1%
20       FIELD #1%, 10% AS A$, 10% AS B$, 10% AS C$
30       LSET A$ = '12345'
         \ RSET B$ = '67890'
         \ RSET C$ = 'VWXYZ'
40       PUT #1%, COUNT 30%
32767    END

Ready
```

Figure 12-1 Record I/O Example #1

The FIELD statement at line 20 defines three 10-character segments of the buffer as A$, B$, and C$. LSET at line 3(positions "12345" in the leftmost 5 of the first 10 characters of the buffer via the pointer A$. Similarly the second and third 10-character pieces of the buffer are set by RSET statements. When run, this program generates:

```
RUNNH
12345              67890       VWXYZ
Ready
```

Note that no carriage return/line feed was output by the PUT statement. (The Monitor outputs a CR/LF sequence as the first part of the READY message.)

Figure 12-2 is a program to move data from a file named "SNOOPY.BAS" in the system library (note the $ in the filename) onto the line printer. Both the line printer and the disk file buffers are initialized to 512 characters. The FIELD statements at lines 140 and 150 set A$ and B$ to refer to these buffers. Data read at line 160 is transferred to the line printer buffer by the LSET statement (RSET would also be acceptable in this one case, since both A$ and B$ are the same length) at line 170. Then, at line 180, this data is output to the line printer. The loop terminates on end-(file on attempting to read past the last block of the SNOOPY.BAS file via the ON ERROR GOTO mechanism. Note that the example at the end of Section 12.3 shows a more efficient technique to do this.

```
LISTNH
110      OPEN '$SNOOPY.BAS' AS FILE 1%
120      ON ERROR GOTO 200
130      OPEN 'LP:' FOR OUTPUT AS FILE 2%, RECORDSIZE 512%
140      FIELD #1%, 512% AS A$
150      FIELD #2%, 512% AS B$
160      GET #1%
170      LSET B$ = A$
180      PUT #2%
190      GOTO 160
200      CLOSE 1%, 2%
32767    END

Ready
```

Figure 12-2 Record I/O Example #2

FIELD statements can be used to perform blocking and deblocking of records where appropriate, as in Figure 12-3.

```
100       GET #2%
110       FOR X%=0% TO 420% STEP 80%
120       FIELD #2%, X% AS A$, 80% AS B$
             •
             •
             •
180       NEXT X%
190       PUT #2%
```

Figure 12-3 FIELD Statement Example

Figure 12-4 illustrates the use of the CVT functions to store numerical data in compact form as strings of binary types. The tape punched by this program has each integer represented on two frames of tape. A similar program could be written to read this binary tape.

```
LISTNH
100       DIM A$(99%)
110       OPEN 'PP:' FOR OUTPUT AS FILE 1%, RECORDSIZE 200%
120       FIELD #1%, 2.*I AS Z$, 2. AS A$(I) FOR I = 0. TM 99.
130       LSET A$(I%) = CVT%$(I%) FOR I%=0% TO 99%
140       PUT #1%
150       CLOSE 1%
32767     END

Ready
```

Figure 12-4 CVT Function Example

12.7 THE XLATE FUNCTION

The XLATE function is provided to translate a string from one storage code into another. For example, while reading a magtape file, it might be necessary to translate from EBCDIC code to ASCII code so that data could be processed by the PDP-11. The XLATE function is of the form:

XLATE (<string1>,<string2>)

For example:

X$ = XLATE(A$,B$)

The first argument, <string1>, is the source string, the second argument <string2>, is the table string; the string value returned by XLATE is called the target string. Characters are taken sequentially from the source string, and the value of each character (0 to 255) is used as an index into the table string (that is, 0 means the first character of the table string, 1 means the second, etc.). The character value from the table string is appended to the target string unless the selected character in the table string has a value of 0 or the table string is shorter than the index value. This means that the target string is equal to or shorter than the source string.

For example, the following program removes all characters except "0" to "9" and changes the characters "8" and "9" into "A" and "B":

```
LISTNH
100     T$ = '01234567AB'
110     T$ = CHR$(0%) + T$ FOR I% = 0% TO 47%
        ! LINE 110 PUT 0'S CORRESPONDING TO CODES 0 TO 47
120     INPUT S$                 ! GET STRING TO TRANSLATE
130     PRINT XLATE(S$,T$)
32767   END

Ready

RUNNH
? 1234567890 DIGITS  ---  ABCDEFGHIJKLMNOPQRSTUVWXYZ
1234567AB0

Ready

RUNNH
? 12XXX34ABCDE097JJKKLMN56789123DEFGHI1357986
12340B7567AB1231357BA6

Ready
```

APPENDICES

The following pages contain a summary of the BASIC-PLUS language, the commands described in the *RSTS/E System User's Guide*, and error messages.

APPENDIX A
BASIC-PLUS LANGUAGE SUMMARY

A.1 SUMMARY OF VARIABLE TYPES

Type	Variable Name	Examples	
Floating Point	Single letter optionally followed by a single digit	A I X3	
Integer	Any floating point variable name followed by a % character	B% D7%	
Character String	Any floating point variable name followed by a $ character	M$ R1$	

Floating Point Matrix	Any floating point variable name followed by one or two dimension elements in parentheses	S(4)
		N2(8)
			E(5,1)
			V8(3,3)

Integer Matrix	Any integer variable name followed by one or two dimension elements in parentheses	A%(2) E3%(4)	I%(3.5) R2%(2,1)
Character String Matrix	Any character string variable name followed by one or two dimension elements in parentheses	C$(1) A2$(8)	S$(8,5) V1$(4,2)

NOTE

When operating in EXTEND mode, the variable name may consist of a letter, followed by 0 to 29 additional characters, each a letter, digit or a dot (i.e., a period, or point). The rules for specifying integers, strings, and dimension elements remain the same for EXTEND mode.

Examples

PER.DIEM.FACTOR	(floating point variable)
Z%	(integer valid also in NO EXTEND)
BRANCH.CONTROL%	(integer variable)
HEADING.A31.FORM$	(string variable)
DECK.OF.CARDS$(3,12)	(string array)

A.2 SUMMARY OF OPERATORS

Type	Operator		Operates Upon
Arithmetic	–	Unary minus	Numeric variables and constants
	–	Exponentiation	
	* , /	Multiplication, division	
	+ , –	Addition, subtraction	
Relational	=	Equals	String or numeric variables and constants
	<	Less than	
	<=	Less than or equal to	
	>	Greater than	
	>=	Greater than or equal to	
	<>	Not equal to	
	==	Approximately equal to (numbers) Identically equal to (strings)	
Logical	NOT	Logical negation	Relational expressions composed of string or numeric elements, integer variables or integer valued expressions
	AND	Logical product	
	OR	Logical sum	
	XOR	Logical exclusive or	
	IMP	Logical implication	
	EQV	Logical equivalence	
String	+	Concatenation	String constants and variables
Matrix	+ , –	Addition and subtraction of matrices of equal dimensions, one operator per statement	Dimensioned variables. See Section 7.6.1 for further details.
	*	Multiplication of conformable matrices	
	*	Scalar multiplication of a matrix, see Section 7.6.1	

A.3 SUMMARY OF FUNCTIONS

Under the Function column, the functions is shown as:

Y=function

where the characters % and $ are appended to Y if the value returned is an integer or character string.

A floating value (X), where specified, can always be replaced by an integer value. An integer value (N%) can always be replaced by a floating value (an implied FIX is done) except in the CVT%$ and MAGTAPE functions (the symbol I% is used to indicate the necessity for an integer value).

Type	Function	Explanation
Mathematical	Y=ABS(X)	Returns the absolute value of X.
	Y=ATN(X)	Returns the arctangent (in radians) of X.
	Y=COS(X)	Returns the cosine of X where X is in radians.
	Y=EXP(X)	Returns the value of e^X, where e=2.71828...
	Y=FIX(X)	Returns the truncated value of X, SGN(X)*INT(ABS(X)).
	Y=INT(X)	Returns the greatest integer in X which is less than or equal to X.
	Y=LOG(X)	Returns the natural logarithm of X, log(e)X.
	Y=LOG10(X)	Returns the common logarithm of X, log(10)X.
	Y=PI	Returns the constant 3.14159...
	Y=RND	Returns a random number between 0 and 1.
	Y=RND(X)	Returns a random number between 0 and 1.
	Y=SGN(X)	Returns the sign function of X; + 1 if possible, 0 if zero, - 1 if negative.
	Y=SIN(X)	Returns the sine of X where X is in radians.
	Y=SQR(X)	Returns the square root of X.
	Y=TAN(X)	Returns the tangent of X where X is in radians.
Print	Y%=POS(X%)	Returns the current position of the print head for I/O channel X%, 0 is the user's Teletype.
	Y$=TAB(X%)	Moves print head to position X% in the current print record, or is disregarded if the current position is beyond X%. (The first position is counted as 0.)
String	Y%=ASCII(A$)	Returns the ASCII value of the first character in the string A$.
	Y$=CHR$(X%)	Returns a character string having the ASCII value of X. Only one character is generated.
	Y$=CVT%$(I%)	Maps integer into 2-character string, see Section 12.5.
	Y$=CVTF$(X)	Maps floating-point number into 4- or 8-character string, see Section 12.5.

Type	Function	Explanation
String cont'd.	Y%=CVT$%(A$)	Maps first two characters of string A$ into an integer, see Section 12.5.
	Y=CVT$F(A$)	Maps first four or eight characters of string A$ into a floating-point number. See Section 12.5.
	Y$=CVT$$(A$,I%)	Converts string A$ to string Y$ according to value of I%. See Section 12.5.
	Y$=RAD$(N%)	Converts an integer value to a 3-character string and is used to convert from Radix-50 format back to ASCII. See the RSTS/E Programming Manual.)
	Y%=SWAP%(N%)	Causes a byte swap operation on the two bytes in the integer variable N%.
	Y$=STRING$(N1%,N2%)	Creates string Y$ of length N1, composed of characters whose ASCII decimal value is N2. See Section 5.5.
	Y$=LEFT(A$,N%)	Returns a substring of the string A$ from the first character to the Nth character (the leftmost N characters).
	Y$=RIGHT(A$,N%)	Returns a substring of the string A$ from the Nth to the last character; the rightmost characters of the string starting with the Nth character.
	Y$=MID(A$,N1%,N2%)	Returns a substring of the string A$ starting with the N1 and being N2 characters long (the characters between and including the N1 to N1+N2-1 characters).
	Y%=LEN(A$)	Returns the number of characters in the string A$, including trailing blanks.
	Y%=INSTR(N1%,A$,B$)	Indicates a search for the substring B$ within the string A$ beginning at character position N1. Returns a value 0 if B$ is not in A$, and the character position of B$ if B$ is found to be in A$ (character position is measured from the start of the string).
	Y$=SPACE$(N%)	Indicates a string of N spaces, used to insert spaces within a character string.
	Y$=NUM$(N%)	Indicates a string of numeric characters representing the value of N as it would be output by a PRINT statement. For example: NUM$(1.0000) = (space)1(space) and NUM$(-1.0000) = -1(space).
	Y$=NUM1$(N)	Returns a string of characters representing the value of N. This is similar to the function NUM$, except that it does not return spaces or E-format results.

Type	Function	Explanation
String cont'd.	Y=VAL(A$)	Computes the numeric value of the string of numeric characters A$. If A$ contains any character not acceptable as numeric input with the INPUT statement, an error results. For example: VAL("15")=15
	X$=XLATE(A$,B$)	Translate A$ to the new string Y$ by means of the table string B$.
	Y$=SUM$(A$,B$)	Returns a numeric string equal to the arithmetic sum of numeric strings A$ and B$.
	Y$=DIFF$(A$,B$)	Returns a numeric string equal to the arithmetic difference A$-B$ of numeric strings A$ and B$.
	Y$=PROD$(A$,B$,P%)	Returns a numeric string equal to the product of numeric strings A$ and B$, rounded or truncated to P% places.
	Y$=QUO$(A$,B$,P%)	Returns a numeric string equal to the arithmetic quotient A$/B$ of numeric strings A$ and B$, rounded or truncated to P% places.
	Y$=PLACE$(A$,P%)	Returns a numeric string equal to the numeric string A, ronnded or truncated to P% places.
	T%=COMP%(A$,B$)	Returns a value reflecting the result of an arithmetic comparison between numeric strings A$ and B$; T% = -1 for A<B, 0 for A=B and 1 for A>B.
System	Y$=DATE$(0%)	Returns the current date in the following format: 02-Mar-71
	Y$=DATE$(N%)	Returns a character string corresponding to a calendar date as follows: N=(day of year)+[(number of years since 1970)*1000] DATE$(1) = "01-Jan-70" DATE$(125) = "05-May-70"
	Y$=TIME$(0%)	Returns the current time of day as a character string as follows: TIME$(0)="05:30 PM" or "17:30 "

Type	Function	Explanation
System cont'd.	Y$=TIME$(N%)	Returns a string corresponding to the time at N minutes before midnight. For example: TIME$(1)="11:59 PM" or "23:59 " TIMES(1440)="12:00 AM" or "00:00 " TIME$(721)="11:59 AM" or "11:59 "
	Y=TIME(0%)	Returns the clock time in seconds since midnight, as a floating-point number.
	Y=TIME(1%)	Returns the central processor time used by the current job in tenths of seconds.
	Y=TIME(2%)	Returns the connect time (during which the user is logged into the system) for the current job in minutes.
	Y=TIME(3%)	Returns to Y the decimal number of kilo-core ticks (KCT's) used by this job. See Section 8.8.
	Y=TIME(4%)	Returns to Y the decimal number of minutes of device time used by this job. See Section 8.8.
	Y%=STATUS	Returns to Y% the status of the OPEN statement executed most recently. See Section 12.3.5.
	Y%=BUFSIZ(N)	Returns to Y% the buffer size of the device or file open on channel N. See Section 12.3.4.
	Y%=LINE	Returns to Y% the line number of the statement being executed at the time of an interrupt. See Section 4.5.
	Y%=ERR	Returns value associated with the last encountered error if an ON ERROR GOTO statement appears in the program. See Section 8.4.
	Y%=ERL	Returns the line number at which the last error occurred if an ON ERROR GOTO statement appears in the program. See Section 8.4.3.
Matrix	MAT Y=TRN(X)	Returns the transpose of the matrix X. See Section 7.6.2.
	MAT Y=INV(X)	Returns the inverse of the matrix X. See Section 7.6.2.

Type	Function	Explanation
Matrix cont'd.	Y=DET	Following an INV(X) function evaluation, the variable DET is equivalent to the determinant of X.
	Y%=NUM	Following input of a matrix, NUM contains the number of rows input, or in the case of a 1-dimensional matrix, the number of elements entered.
	Y%=NUM2	Following input of a matrix, NUM2 contains the number of elements entered in that row.
Input/Output	Y%=RECOUNT	Returns the number of characters read following every input operation. Used primarily with non-file-structured devices. See Section 12.3.1.

A.4 SUMMARY OF BASIC-PLUS STATEMENTS

The following summary of statements available in the BASIC-PLUS language defines the general format for the statement as a line in a BASIC program. If more detailed information is needed, the reader is referred to the section(s) in the manual dealing with that particular statement.

In these definitions, elements in angle brackets are necessary elements of the statement. Elements in square brackets are necessary elements of which the statement may contain one. Elements in braces are optional elements of the statement.

The various elements and their abbreviations are described below:

variable or var	Any legal BASIC variable as described in A.1 or Section 2.5.2.
line number	Any legal BASIC line number described in Section 2.2.
expression or expr	Any legal BASIC expression as described in Section 2.5.
message	Any combination of characters.
condition or cond	Any logical condition as described in Section 3.5.
constant	Any acceptable integer constant (need not contain a % character).
argument(s) or arg	Dummy variable names.
statement	Any legal BASIC-PLUS statement.
string	Any legal string constant or variable as described in Section 5.1.
protection	Any legal protection code, as described in the *RSTS/E System User's Guide*.
value(s)	Any floating point, integer, or character string constant.
list	The legal list for that particular statement.
dimension(s)	One or two dimensions of a matrix, the maximum dimension(s) for that particular statement.

Statement Formats and Examples **Manual Section**

REM

line number REM <message> 3.1
line number <statement> !<message>

```
100      REM      THIS IS A COMMENT
110      ! THIS IS ANOTHER FORM OF COMMENT
120      PRINT              ! PERFORM A CR/LF
```

LET

line number {LET}<var> ,<var>,<var>. . . = <exp> 3.2

```
110      LET A% = 40% \ B=22
120      C,F1,V(0) = 0               !MULTIPLE ASSIGNMENT
```

DIM

line number DIM <var(dimension(s))> 3.6.2 7.1

```
30       DIM A(20), B$(6,5), C%(99)
```

line number DIM #<constant>,<var(dimension(s))>=<constant> 11.1

```
70       DIM #4, A$(100) = 32, B(50,50)
```

RANDOMIZE

line number RANDOMIZE — 3.7.4

```
40       RANDOM
100      RANDOMIZE
```

IF-THEN, IF-GOTO

line number IF <cond> [THEN<statement> THEN<line number> GOTO<line number>] 3.5

```
55       IF A>B OR B>C THEN PRINT 'NO'
65       IF FNA(R) = B THEN 250
90       IF L < X^2% AND L<> 0. GOTO 345
```

IF-THEN-ELSE

line number IF <cond> [THEN<statement> THEN<line number> GOTO<line number>] {ELSE<statement> ELSE<line number>} 8.5

```
100      IF B = A THEN PRINT 'EQUAL' ELSE PRINT 'NOT EQUAL'
110      IF A == N THEN 200 ELSE PRINT A \ STOP
120      IF FNA(R) = B
                  THEN GO TO 260
                  ELSE LET B = B+FNA(R1)
                  \ GO TO 370
```

Statement Formats and Examples **Manual Section**

FOR
line number FOR <var> = <exp>TO<exp> {STEP<exp>} 3.6.1

```
200       FOR I=2 TO 40 STEP 2
320       FOR T%=0% TO T6% STEP I%
410       FOR N=A TO (C + S1)/A
```

NEXT
line number NEXT <var> 3.6.1

```
460       NEXT I
465       NEXT N%
```

FOR-WHILE, FOR-UNTIL 8.6
line number FOR <var> = <exp> STEP<exp> $\begin{bmatrix}\text{WHILE}\\\text{UNTIL}\end{bmatrix}$ <cond>

```
450       FOR I=1. STEP 3. WHILE I<X
470       FOR N% = 2% STEP 4% UNTIL N%>A% OR N%=B%
500       FOR B = 0. STEP B+1. UNTIL B>B1
```

EXTEND
line number EXTEND

```
    10        EXTEND  !PROGRAM IN EXTEND MODE
```
2.1

NO EXTEND
line number NO EXTEND

```
    110       NOEXTEND
    110       NO EXTEND
```
2.1

DEF, single line 3.7.3
line number DEF FN<var> (arg) = <exp(arg)> 5.5.1

```
120       DEF FNA(X,Y,Z) = SQR(X^2% + Y^2% + Z^2%)
```
6.4

DEF, multiple line
line number DEF FN<var>(arg) 3.7.5
<statements> 5.5.1
line number FN<var> = <exp> 6.4
line number FNEND 8.1

```
300       DEF FNF(M%)       !FACTORIAL FUNCTION
310       IF M%=0% OR M%=1%
                    THEN FNF=1%
                    ELSE FNF=M%*FNF(M%-1%)
320       FNEND
```

Statement Formats and Examples — **Manual Section**

GOTO

line number GOTO <line number> 3.4

```
100      GOTO 150
```

ON-GOTO

line number ON <exp> GOTO <list of line numbers> 8.2

```
150      ON X% GOTO 170,570,430,300
```

GOSUB

line number GOSUB <line number> 3.8.1

```
190      GOSUB 2000
```

ON-GOSUB

line number ON <exp> GOSUB <list of line numbers> 8.3

```
230      ON FNC(M) GOSUB 2000,2400,3000
```

RETURN

line number RETURN 3.8.2

```
370      RETURN
```

CHANGE 5.2

line number CHANGE $\begin{bmatrix}\text{<array name>}\\ \text{<string var>}\end{bmatrix}$ TO $\begin{bmatrix}\text{<string var>}\\ \text{<array name>}\end{bmatrix}$

```
300      CHANGE A$ TO X
450      CHANGE F1 TO F1$
```

OPEN

line number OPEN<string> FOR $\begin{Bmatrix}\text{INPUT}\\ \text{OUTPUT}\end{Bmatrix}$ AS FILE <exp> 9.2

{,RECORDSIZE<exp>} {,CLUSTERSIZE<exp>} {,MODE<exp>} 12.1

```
100      OPEN 'PP:' FOR OUTPUT AS FILE B1%
120      OPEN 'FOO' AS FILE 3
160      OPEN 'DT4:DATA.TR' FOR INPUT AS FILE 10
```

CLOSE 9.3

line number CLOSE <list of exp> 12.2

```
780      CLOSE 2%
920      CLOSE 11, 3, N1
```

Statement Formats and Examples **Manual Section**

READ 3.3.1
line number READ <list of variables> 5.3
6.3

```
110       READ A, B$, F1%, B(1%),R2
```
10.1

DATA 3.3.1
line number DATA <list of values> 5.3
6.3

```
1300      DATA 4.3, "STRING", 10,1000,1.45E9
```

RESTORE 3.3.1
line number RESTORE 10.2

```
130       RESTORE
```

PRINT 3.3.2
line number PRINT {#<exp>,}<list> 5.4
6.3

```
130       PRINT    !GENERATES CR/LF
170       PRINT 'BEGINNING OF OUTPUT:';I,A*I
240       PRINT #4, 'OUTPUT TO DEVICE';N%
260       PRINT 'TITLE: ';T$, 'REF. #';R$
```
10.3
10.3.1
10.3.2

PRINT USING
line number PRINT {#<exp>,}USING <string>, <list> 10.3.3

```
540       PRINT USING '##.##',AA
680       PRINT #7, USING B$,A,B,C
```

INPUT 3.3.3
line number INPUT {#<exp>,}<list> 5.3
6.3

```
140       INPUT 'TYPE YOUR NAME ',A$
180       INPUT #8, A,N, B$
```
10.4
10.4.1

INPUT LINE 5.3.1
line number INPUT LINE {#<exp>,}<string>

```
170       INPUT LINE R$
190       INPUT LINE #1,E$
```

NAME-AS
line number NAME <string> AS <string> 9.4

```
455       NAME 'NONAME' AS 'FILE1<48>'
990       NAME "DT4:MATRIX" AS 'MATA1<48>'
```

Statement Formats and Examples	Manual Section

KILL 9.5

line number KILL <string>

```
190     KILL 'NONAME'
```

ON ERROR GOTO 8.4

line number ON ERROR GOTO <line number>

```
100     ON ERROR GOTO 9000
110     ON ERROR GO TO               !DISABLES ERROR ROUTINE
111     ON ERROR GOTO 0              !DISABLES ERROR ROUTINE
```

RESUME 8.4.1

line number RESUME <line number>

```
1000    RESUME           !EQUIVALENT TO RESUME 0
650     RESUME 200       !SPECIFY RESUMPTION POINT
```

CHAIN

line number CHAIN <string> {<exp>} 9.6

```
440     CHAIN 'PROG2'
550     CHAIN 'PROG3' 75
670     CHAIN "PROG3" A
```

STOP

line number STOP 3.9

```
450     STOP
```

END

line number END 3.9

```
32767   END
```

Matrix Statements

MAT READ 7.2

line number MAT READ <list of matrices>

```
150     DIM A(20), B$(32), C%(15,10)
290     MAT READ A, B$(25), C%
```

MAT PRINT 7.3 10.3.4

line number MAT PRINT {#<exp>,} <matrix name>

```
150     DIM A(20), B%(15,30)
190     MAT PRINT A,     !PRINT 20 ELEMENTS FIVE TO A LINE
220     MAT PRINT B%(10,25);     !PRINT 10-BY-25 SUBSET
        OF B%, PACKED
270     MAT PRINT #2, A;         !PRINT ON OUTPUT CHANNEL 2
```

Statement Formats and Examples	Manual Section

MAT INPUT — 7.4, 10.4.1

line number MAT INPUT {#<exp>,} <list of matrices>

```
100     DIM B$(40), F1%(35)
110     OPEN 'DT3:FOO' FOR INPUT AS FILE 3%
120     MAT INPUT #3, B4, F1%
```

MAT Initialization — 7.5

line number MAT <matrix name> = [ZER / CON / IDN] {(dimension(s))}

```
100     DIM B(15,10), A(10), C%(5)
110     MAT C% = CON
120     MAT B = IDN(10,10)
130     MAT B = ZER(N,M)
```

Statement Modifiers (can be used in immediate mode)

IF — 8.7.1

<statement> IF <condition>

```
510     PRINT T% IF T%>T1%
```

UNLESS — 8.7.2

<statement> UNLESS <condition>

```
340     PRINT A$ UNLESS Y%<0%
```

FOR — 8.7.3

<statement> FOR <var> = <exp> TO <exp> STEP <exp>

```
175     LET B$(I%) = C$(I%) FOR I% = 1% TO J1%
190     READ A(I%) FOR I% = 0% TO 20% STEP I5%
```

WHILE — 8.7.4

<statement> WHILE <condition>

```
230     LET A(I%) = FNX(I%) WHILE A<45.5
```

UNTIL — 8.7.5

<statement> UNTIL <condition>

```
1060    IF B <> 0 THEN A(I%) = B UNTIL I% > K
```

System Statements

line number SLEEP <expression> — 8.8

```
260     SLEEP 20          !DISMISS JOB FOR 20 SECONDS
```

Statement Formats and Examples	Manual Section
line number WAIT <expression>	8.8
`520        WAIT A% + 5%`	
Record I/O Statements	
line number LSET <string var> ,<string var> =<string>	12.4.3
`900        LSET B$ = 'XYZ'`	
line number RSET <string var> ,<string var> =<string>	12.4.3
`250        RSET C$ = "67890"`	
line number FIELD# <expr>,<expr> AS <string var> ,<expr> AS <string var>	12.4.2
`710        FIELD #2%, 10% AS A$, 20% AS B$`	
line number GET# <expr> ,RECORD<expr>	12.3
`140        GET #1%, RECORD 99%`	
line number PUT# <expr> ,RECORD <expr> ,COUNT<expr>	12.3
`390        PUT #1%, COUNT 80%`	

APPENDIX B
BASIC-PLUS COMMAND SUMMARY

This appendix describes briefly those RSTS/E commands used most frequently by BASIC-PLUS users. It is intended as an introductory summary; its purpose is to help BASIC-PLUS learners begin programming at the terminal before having studied the *RSTS/E System User's Guide* in detail.

Command	**Explanation**	**Section in *RSTS/E System User's Guide***
APPEND	Used to include contents of a previously saved source program in current program.	9.1.4
ASSIGN	Used to reserve an I/O device for the use of the individual issuing the command. The specified device can then be given commands only from the job which issued the ASSIGN. Also establishes a logical name for a device, establishes an account for the @ character, and establishes a default protection code.	5.1 5.4 6.1
ATTACH	Attaches a detached job to the current terminal.	14.1
BYE	Indicates to RSTS/E that a user wishes to leave the terminal. Closes and saves any files remaining open for that user. After the user types BYE, the system responds: Confirm: At this point the user has five options: ? Requests information on valid responses to the "Confirm" prompt. Y Requests normal logout. N Requests no logout; effectively negates the BYE command. I Requests an opportunity to delete files individually prior to logout. F Fast logout. If BYE is followed immediately with one of the valid responses, the "Confirm" prompt is not printed.	2.3
CAT CATALOG	Returns the user's file directory. Unless another device is specified following the term CAT or CATALOG, the disk is the assumed device.	8.9
CCONT	For privileged users. Same as CONT command, but detaches job from terminal.	10.2.3
COMPILE	Allows the user to store a compiled version of his BASIC program. The file is stored on disk with the current name and the extension .BAC. Or, a new file name can be indicated and the extension .BAC will still be appended.	8.4.3

Command	Explanation	Section in *RSTS/E System User's Guide*
CONT	Allows the user to continue execution of the program currently in memory following the execution of a STOP statement.	10.2.2
DEASSIGN	Used to release the specified device for use by others. If no particular device is specified, all devices assigned to that terminal are released. An automatic DEASSIGN is performed when the BYE command is given. Also releases any logical name for a device.	5.2 5.4.3 5.4.4 5.7
DELETE	Allows the user to remove one or more lines from the program currently in memory. Following the word DELETE the user types the line number of the single line to be deleted or two line numbers separated by a dash (–) indicating the first and last line of the section of code to be removed. Several single lines or line sections can be indicated by separating the line numbers, or line number pairs, with a comma.	9.1.2
EXTEND	Allows the user to include Extend Mode features in programs and to execute programs that include Extend Mode features. The system default is No Extend Mode following log-in, and remains so until an EXTEND command changes mode. Either mode condition can be overridden at the program level.	9.2.1
HELLO	Indicates to RSTS/E that a user wishes to log onto the system. Allows the user to input project-programmer number and password. Also attaches a detached job to the current terminal or changes accounts without having to log off the system	2.2 14.1.1
KEY	Used to re-enable the echo feature on the user terminal following the issue of a TAPE command. Enter with LINE FEED or ESCAPE key.	5.8.2
LENGTH	Returns the length of the user's current program in core, in 1K increments, along with the maximum size. If the current program is between 6K and 7K, for instance, and the maximum size is 16K, the following message appears: 7(16)K of memory used	8.8
LIST	Allows the user to obtain a printed listing at the user terminal of the program currently in memory, or one or more lines of that program. The word LIST by itself will cause the listing of the entire user program. LIST followed by one line number will list that line; and LIST followed by two line numbers separated by a dash (–) will list the lines between and including the lines indicated. Several single lines or line sections can be indicated by separating the line numbers, or line number pairs, with a comma.	9.1.1
LISTNH	Same as LIST, but does not print header containing the program name and current data.	9.1.1
LOGIN	Same as HELLO.	14.1.1

Command	Explanation	Section in *RSTS/E System User's Guide*
MOUNT	Allows user to logically mount a disk pack during timesharing. This command specifies the physical device name and pack identification label.	5.6.11
NEW	Clears the user's area in memory and allows the user to input a new program from the terminal. A program name can be indicated following word NEW or when the system requests it.	8.1.1
NO EXTEND	Negates any previous EXTEND command, placing the system default in No Extend Mode. Extend Mode features are no longer available unless the program includes an EXTEND statement to override system default.	9.2.1
OLD	Clears the user's area in memory and allows the user to recall a saved program from a storage device. The user can indicate a program name following the word OLD or when the system requests it. If no device name is given, the file is assumed to be on the public structure. A device specification without a filename will cause a program to be read from an input-only device (such as high-speed reader, card reader). The default filename extension is .BAS.	8.3
REASSIGN	Transfers control of a device to another job.	5.3
RENAME	Causes the name of the program currently in memory to be changed to the name specified after the word RENAME.	8.5
REPLACE	Same as SAVE, but allows the user to substitute a new program for an old program with the same name, erasing the old program.	8.6
RUN	Allows the user to begin execution of the program currently in memory. The word RUN can be followed by a filename in which case the file is loaded from the public structure, compiled (if necessary), and run; alternatively, the device and filename can be indicated if the file is not on the public structure. A device specification without a filename will cause a program to be read from an input only device (such as high-speed reader, card reader). The default filename extension is .BAS.	8.4
RUNNH	Causes execution of the program currently in memory but header information containing in program name and current data is not printed. If a filename is used, the command is executed as if no filename were given. The default filename extension is .BAS.	8.4

Command	Explanation	Section in *RSTS/E System User's Guide*
SAVE	Causes the program currently in memory to be saved on the public structure under its current filename with the extension .BAS. Where the word SAVE is followed by a filename or a device and a filename, the program in memory is saved under the name given and on the device specified. A device specification without a filename will cause the program to be output to any output only device (line printer, high-speed punch). The default filename extension is .BAS.	8.2
SCALE	Sets the scale factor to a designated value or prints the value(s) currently in effect if no value is designated.	8.10
TAPE	Used to disable the echo feature on the user terminal while reading paper tape via the low-speed reader.	5.8.1
UNSAVE	The word UNSAVE is followed by the filename, and optionally, the extension of the file to be removed. The UNSAVE command cannot remove files without an extension. If no extension is specified, the source (.BAS) file is deleted. If no device is specified, the public structure is assumed.	

Special Control Character Summary

CTRL/C	Causes the system to return to BASIC command mode to allow for issuing of further commands or editing. Echoes on terminal as ^C.	5.9.1 10.3
CTRL/O	Used as a switch to suppress/enable output of a program on the user terminal. Echoes as ^O.	5.9.2 10.3
CTRL/Q	When generated by a device on which a CTRL/S has interrupted output, causes computer to resume output at the next character.	5.9.3
CTRL/S	When generated by a device for which STALL characteristics are set, interrupts computer output on the device until either CTRL/Q or another character is generated.	5.9.3
CTRL/U	Deletes the current typed line, echoes as ^U and performs a carriage return/line feed.	9.1.3.2
CTRL/Z	Used as an end-of-file character.	5.9.4
ESCape or ALT MODE Key	Enters a typed line to the system, echoes on the user terminal as a $ character and does not cause a carriage return/line feed.	5.9.6
LINE FEED Key	Used to continue the current logical line on an additional physical line. Performs a line feed/carriage return operation.	9.2.2.2
RETURN Key	Enters a typed line to the system, results in a carriage return/line feed operation at the user terminal.	5.9.5
RUBOUT Key	Deletes the last character typed on that physical line. Erased characters are shown on the teleprinter between backslashes.	9.1.3.1
TAB or CTRL/I	Performs a tabulation to the next tab stop (eight spaces apart) of the terminal printing line.	9.2.2.3
FORM FEED or CTRL/L	Enters a typed line to the system, results in a form feed operation at the terminal.	

APPENDIX C
ERROR MESSAGES

Messages in RSTS/E are generated for BASIC-PLUS errors[1] and RSTS/E errors. To avoid confusion, both types of messages are called RSTS/E error messages and are described as one set. The BASIC-PLUS errors cover compiler and run time conditions such as a violation of the syntax rules (SYNTAX ERROR) and referencing an element of an array beyond the defined limits (SUBSCRIPT OUT OF RANGE). The RSTS/E errors involve operating system conditions such as failing to locate the file or account specified (CAN'T FIND FILE OR ACCOUNT) and requesting the hardware to perform a function for which it is not ready (DEVICE HUNG OR WRITE LOCKED).

In most cases, if no error trapping is being done (that is, an ON ERROR GOTO statement is not in effect), BASIC-PLUS stops running the program. It prints the error message and the line number of the BASIC-PLUS statement that was being executed when the error occurred. The following sample printout shows the procedure.

```
10      OPEN 'Z' FOR INPUT AS FILE 1%
RUNNH
?CAN'T FIND FILE OR ACCOUNT AT LINE 10

READY
```

As the READY message indicates, control returns to the system.

An exception to this procedure occurs when an INPUT statement is being executed at the job's console terminal and error trapping is not in effect. The system generates the error message and executes the statement again as shown in the sample printout below.

```
10      ON ERROR GOTO 0 \ INPUT 'INTEGER VALUE';A%
RUNNH
INTEGER VALUE? C
%DATA FORMAT ERROR AT LINE 10
INTEGER VALUE?
```

With error trapping disabled at line 10, an invalid response to the INPUT statement causes the system to print the error message, clear the error condition, and execute the statement again.

Associated with each message is an error variable called ERR. Whenever an error occurs with trapping in effect, the system checks the error variable which is a decimal number in the range 0 to 127. An error with a number between 1 and 70 causes the system to transfer control to the line number indicated in the ON ERROR GOTO statement. The system does not print the error message. The user program is able to check the ERR variable and perform a recovery procedure. If the error number is between 71 and 127, the system does not transfer control to the recovery routine but prints the message and returns control to the system. (Error number 0 is reserved to identify the system installation name.)

Because a BASIC-PLUS program can recover from certain errors, this appendix lists errors in two categories – recoverable and non-recoverable. The recoverable error messages are listed in ascending order of their related error numbers. A program can use these error numbers to differentiate errors. Non-recoverable errors are in alphabetical order without error numbers because a program can not use these numbers in an error handling routine.

[1]Different messages are generated while a job is operating under run-time systems other than BASIC-PLUS. Such run-time systems are those for COBOL and FORTRAN-IV. For these error messages, consult the appropriate User's Guides.

The first character position of each.message indicates the severity of the error. Table C-1 describes this standard.

Table C-1 Severity Standard in Error Messages

Character	Severity	Meaning
%	Warning	Execution of the program can continue but may not generate the expected results.
?	Fatal	Execution cannot continue unless the user removes the cause of the error.
	Information	A message beginning with neither a question mark nor a percent is for information only.

The severity indication is useful for utility programs such as BATCH which examines system output.

In the descriptions of error messages, certain abbreviations, as shown in Table C-2, denote special characteristics of the error.

Table C-2 Special Abbreviations for Error Descriptions

Abbreviation	Meaning
(C)	Continue. If an ON ERROR GOTO statement is not in effect, execution continues but with the conditions described.
(SPR)	Software Performance Report. This error should occur only under the conditions described. If it occurs under any other conditions, the user should file an SPR with DIGITAL and document the conditions under which the error occurred.

An error whose description is accompanied by the abbreviation (C) indicates an exception to the error trapping procedure. If such an error occurs in a program with no error trapping in effect, BASIC-PLUS prints the error message and line number but continues running the program. The following sample printout shows the procedure.

```
100     ON ERROR GOTO 0 \ A% = 32768.
200     PRINT A%
RUNNH
%INTEGER ERROR AT LINE 100
 0

READY
```

The INTEGER ERROR is generated at line 100 by the attempt to compute a value outside the range for integers. After the error message is printed, processing continues but with the conditions described in the error meaning. 0 is substituted for the erroneously computed value.

The number of RSTS/E error messages is restricted to 127. Because of this restriction, certain error messages have multiple meanings. The specific meaning of an error message depends on the operation being performed when the error condition occurs. For example, if the system attempts a file access and the designated file can not be located, RSTS/E generates the CAN'T FIND FILE OR ACCOUNT error (ERR=5). That same error condition, however, applies to other, generically similar access operations. Thus, if a program attempts to send a message to another program and the proper entry is not found in the system table of eligible receivers, RSTS/E returns error number 5. Though the second failure does not involve a file access error, it too is classified as an access failure.

Certain RSTS/E errors, although classified as user recoverable, are not capable of being trapped by a program. Table C-3 lists such errors.

Table C-3 Non-Trappable Errors in Recoverable Class

ERR	Message Printed
34	RESERVED INSTRUCTION TRAP
36	SP (R6) STACK OVERFLOW
37	DISK ERROR DURING SWAP
38	MEMORY PARITY FAILURE

These errors involve special conditions which a user program cannot control and which ought not to occur on a normal system. For example, the DISK ERROR DURING SWAP error indicates a hardware problem. The system does not return control to the program. The error condition itself, however, can be either transient or recurring. Such errors should be brought to the attention of the system manager for further investigation. These errors are recoverable in the strict sense that the monitor can take corrective action but the BASIC-PLUS run-time system does not return control to the user program.

C.1 USER RECOVERABLE

ERR	Message Printed	Meaning
0	(system installation name)	The error code 0 is associated with the system installation name and is used by system programs to print identification lines.
1	?Bad directory for device	The directory of the device referenced is in an unreadable format. The magtape label format on tape differs from the system-wide default format, the current job default format, or the format specified in the OPEN statement. Use the ASSIGN command to set the correct format default or change the format specification in the MODE option of the OPEN statement.
2	?Illegal file name	The filename specified is not acceptable. It contains unacceptable characters or the filename specification format has been violated. The CCL command to be added begins with a number or contains a character other than A through Z, 0 through 9 and commercial at (@).
3	?Account or device in use	Reassigning or dismounting of the device cannot be done because the device is open or has one or more open files. The account to be deleted has one or more files and must be zeroed before being deleted. The run time system to be deleted is currently loaded in memory and in use. Output to a pseudo keyboard cannot be done unless the device is in KB wait state. An echo control field cannot be declared while another field is currently active. The CCL command to be added already exists.

ERR	Message Printed	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 or account number specified was not found on the device specified. The CCL command to be deleted does not exist.
6	?Not a valid device	The device specification supplied is not valid for one of the following reasons. The unit number or its type is not configured on the system. The specification is logical and untranslatable because a physical device is not associated with it.
7	?I/O channel already open	An attempt was made to open one of the twelve I/O channels which had already been opened by the program. (SPR)
8	?Device not available	The specified device exists on the system but a user's attempt to ASSIGN or OPEN it is prohibited for one of the following reasons. The device is currently reserved by another job. The device requires privileges for ownership and the user does not have privilege. The device or its controller has been disabled by the system manager. The device is a keyboard line for pseudo keyboard use only.
9	?I/O channel not open	Attempt to perform I/O on one of the twelve channels which has not been previously opened in the program.
10	?Protection violation	The user was prohibited from performing the requested operation because the kind of operation was illegal (such as input from a line printer) or because the user did not have the privileges necessary (such as deleting a protected file).
11	?End of file on device	Attempt to perform input beyond the end of a data file; or a BASIC source file is called into memory and is found to contain no END statement.
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. This error is caused by hardware condition. Report such occurrences to the system manager. (See the discussion at beginning of appendix.)
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	User should 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.
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 number which is already within the system.

ERR	Message Printed	Meaning
17	?Too many open files on unit	Only one open DECtape output file is permitted per DECtape drive. Only one open file per magtape drive is permitted.
18	?Illegal SYS( ) usage	Illegal use of the SYS system function.
19	?Disk block is interlocked	The requested disk block segment is already in use (locked) by some other user.
20	?Pack IDs don't match	The identification code for the specified disk pack does not match the identification code already on the pack.
21	?Disk pack is not mounted	No disk pack is mounted on the specified disk drive.
22	?Disk pack is locked out	The disk pack specified is mounted but temporarily disabled.
23	?Illegal cluster size	The specified cluster size is unacceptable. The cluster size must be a power of 2. For a file cluster, the size must be equal to or greater than the pack cluster size and must not be greater than 256. For a pack cluster, the size must be equal to or greater than the device cluster size and must not be greater than 16. The device cluster size is fixed by type.
24	?Disk pack is private	The current user does not have access to the specified private disk pack.
25	?Disk pack needs 'CLEANing'	Non-fatal disk mounting error; use the CLEAN operation in UTILTY.
26	?Fatal disk pack mount error	Fatal disk mounting error. Disk cannot be successfully mounted.
27	?I/O to detached keyboard	I/O was attempted to a hung up dataset or to the previous, but now detached, console keyboard for the job.
28	?Programmable ↑C trap	A CTRL/C combination was typed while an ON ERROR GOTO statement was in effect and programmable CTRL/C trapping was enabled.
29	?Corrupted file structure	Fatal error in CLEAN operation.
30	?Device not file structured	An attempt is made to access a device, other than a disk, DECtape, or magtape device, 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 or in the COUNT option of the PUT statement is not a multiple of the block size of the device being used for I/O, or is illegal for the device. An attempt is made to run a compiled file which has improper size due to incorrect transfer procedure.
32	?No buffer space available	The user accesses a file and the monitor requires one small buffer to complete the request but one is not currently available. If the program is sending messages, two conditions are possible. The first occurs when a program sends a message and the receiving program has exceeded the pending message limit. The second occurs when a sending program attempts to send a message and a small buffer is not available for the operation.

ERR	Message Printed	Meaning
33	?UNIBUS timeout fatal trap	This hardware error occurs when an attempt is made to address non-existent memory or an odd address using the PEEK function. An occurrence of this error message in any other case is cause for an SPR.
34	?Reserved instruction trap	An attempt is made to execute an illegal or reserved instruction or an FPP instruction when floating point hardware is not available. (See discussion at beginning of appendix.)
35	?Memory management violation	This hardware error occurs when an illegal Monitor address is specified using the PEEK function. Generation of the error message in situations other than using PEEK is cause for an SPR.
36	?SP (R6) stack overflow	An attempt to extend the hardware stack beyond its legal size is encountered. (See discussion at beginning of appendix.) (SPR)
37	?Disk error during swap	A hardware error occurs when a user's job is swapped into or out of memory. The contents of the user's job area are lost but the job remains logged into the system and is reinitialized to run the NONAME program. Report such occurrences to the system manager. (See discussion at beginning of appendix.)
38	?Memory parity failure	A parity error was detected in the memory occupied by this job. (See discussion at beginning of appendix.)
39	?Magtape select error	When access to a magtape drive was attempted, the selected unit was found to be off line.
40	?Magtape record length error	When performing input from magtape, the record on magtape was found to be longer than the buffer designated to handle the record.
41	?Non-res run-time system	The run time system referenced has not been loaded into memory and is therefore non-resident.
42	?Virtual buffer too large	Virtual core buffers must be 512 bytes long.
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 the integer numbers 1 to 12.
47	?Line too long	Attempt to input a line longer than 255 characters (which includes any line terminator). Buffer overflows.
48	%Floating point error	Attempt to use a computed floating point number outside the range $1E-38 < n < 1E38$ excluding zero. If no transfer to an error handling routine is made, zero is returned as the floating point value. (C)

ERR	Message Printed	Meaning
49	%Argument too large in EXP	Acceptable arguments are within the approximate range $-89<arg<+88$. The value returned is zero. (C)
50	%Data format error	A READ or INPUT statement detected data in an illegal format. For example, 1..2 is an improperly formed number, and 1.3 is an improperly formed integer, and X" is an illegal string. (C)
51	%Integer error	Attempt to use a computed integer outside the range $-32768<n<32767$. For example, an attempt is made to assign to an integer variable a floating point number outside the integer range. If no transfer to an error handling routine is made, zero is returned as the integer value. (C)
52	?Illegal number	Integer overflow or underflow or floating point overflow. The range for integers is -32768 to +32767; for floating point numbers, the upper limit is 1E38. (For floating point underflow, the FLOATING POINT ERROR (ERR=48) is generated.)
53	%Illegal argument in LOG	Negative or zero argument to LOG function. Value returned is the argument as passed to the function. (C)
54	%Imaginary square roots	Attempt to take square root of a number less than zero, The value returned is the square root of the absolute value of the argument. (C)
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 or nearly singular matrix.
57	?Out of data	The DATA list was exhausted and a READ requested additional data.
58	?ON statement out of range	The index value in an ON-GOTO or ON-GOSUB statement is less than one 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 32766 or below -32767.
61	%Division by 0	Attempt by the user program to divide some quantity by zero. If no transfer is made to an error handler routine, a 0 is returned as the result. (C)
62	?No run-time system	The run-time system referenced has not been added to the system list of run time systems.
63	?FIELD overflows buffer	Attempt to use FIELD to allocate more space than exists in the specified buffer.
64	?Not a random access device	Attempt to perform random access I/O to a non-random access device.

ERR	Message Printed	Meaning
65	?Illegal MAGTAPE ( ) usage	Improper use of the MAGTAPE function.
66	?Missing special feature	User program employs a BASIC-PLUS feature not present on the given installation.
67	?Illegal switch usage	A CCL command contains an error in an otherwise valid CCL switch. (For example, the /SI:n switch was used without a value for n or a colon; or more than one of the same type of CCL switch was specified.) A file specification switch is not the last element in a file specification or is missing a colon or an argument.

C.2 NON-RECOVERABLE

Message Printed	Meaning
?Arguments don't match	Arguments in a function call do not match, in number or in type, the arguments defined for the function.
?Bad line number pair	Line numbers specified in a LIST or DELETE command were formatted incorrectly.
?Bad number in PRINT-USING	Format specified in the PRINT-USING string cannot be used to print one or more values.
?Can't compile statement	
?Can't CONTinue	Program was stopped or ended at a spot from which execution cannot be resumed.
?Data type error	Incorrect usage of floating-point, integer, or character string format variable or constant where some other data type was necessary.
?DEF without FNEND	A second DEF statement was encountered in the processing of a user function without an FNEND statement terminating the first user function definition.
?End of statement not seen	Statement contains too many elements to be processed correctly.
?Execute only file	Attempt was made to add, delete or list a statement in a compiled (.BAC) format file.
?Expression too complicated	This error usually occurs when parentheses have been nested too deeply. The depth allowable is dependent on the individual expression.
?File exists-RENAME/REPLACE	A file of the name specified in a SAVE command already exists. In order to save the current program under the name specified, use REPLACE, or use RENAME followed by SAVE.
?FNEND without DEF	An FNEND statement was encountered in the user program without a previous function call having been executed.

Message Printed	Meaning
?FNEND without function call	A FNEND statement was encountered in the user program without a previous DEF statement being seen.
?FOR without NEXT	A FOR statement was encountered in the user program without a corresponding NEXT statement to terminate the loop.
?Illegal conditional clause	Incorrectly formatted conditional expression.
?Illegal DEF nesting	The range of one function definition crosses the range of another function definition.
?Illegal dummy variable	One of the variables in the dummy variable list of user-defined function is not a legal variable name.
?Illegal expression	Double operators, missing operators, mismatched parentheses, or some similar error has been found in an expression.
?Illegal FIELD variable	The FIELD variable specified is unacceptable.
?Illegal FN redefinition	Attempt was made to redefine a user function.
?Illegal function name	Attempt was made to define a function with a function name not subscribing to the established format.
?Illegal IF statement	Incorrectly formatted IF statement.
?Illegal in immediate mode	User issued a statement for execution in immediate mode which can only be performed as part of a program.
?Illegal line number(s)	Line number reference outside the range 1<n<32767.
?Illegal mode mixing	String and numeric operations cannot be mixed.
?Illegal statement	Attempt was made to execute a statement that did not compile without errors.
?Illegal symbol	An unrecognizable character was encountered. For example, a line consisting of a # character.
?Illegal verb	The BASIC verb portion of the statement cannot be recognized.
%Inconsistent function usage	A function is defined with a certain number of arguments but is elsewhere referenced with a different number of arguments. Fix the reference to match the definition and reload the program to reset the function definition.
%Inconsistent subscript use	A subscripted variable is being used with a different number of dimensions from the number with which it was originally defined.
x(y)K of memory used	Message printed by the LENGTH command. The value for x is the current size, to the nearest 1K-word increment, of the program in memory. The value for y is the size to which the program can expand, given the run time system being used and the job's private memory size maximum set by the system manager.

Message Printed	Meaning
?Literal string needed	A variable name was used where a numeric or character string was necessary.
?Matrix dimension error	Attempt was made to dimension a matrix to more than two dimensions, or an error was made in the syntax of a DIM statement.
?Matrix or array without DIM	A matrix or array element was referenced beyond the range of an implicitly dimensioned matrix.
?Maximum memory exceeded	During an OLD operation, the job's private memory size maximum was reached. While running a program, the system required more memory for string or I/O buffer space and the job's private memory size maximum or the system maximum (16K words for BASIC-PLUS) was reached.
?Modifier error	Attempt to use one of the statement modifiers (FOR, WHILE, UNTIL, IF, or UNLESS) incorrectly. An OPEN statement modifier, such as a RECORDSIZE, CLUSTERSIZE, FILESIZE, or MODE option, is not in the correct order.
?NEXT without FOR	A NEXT statement was encountered in the user program without a previous FOR statement having been seen.
?No logins	Message printed if the system is full and cannot accept additional users or if further logins are disabled by the system manager.
?Not enough available memory	An attempt is made to load a non-privileged compiled program which is too large to run, given the job's private memory size maximum. The program must be made privileged to allow it to expand above a private memory size maximum; or the system manager must increase the job's private memory size maximum to accommodate the program.
?Number is needed	A character string or variable name was used where a number was necessary.
?1 or 2 dimensions only	Attempt was made to dimension a matrix to more than two dimensions.
?ON statement needs GOTO	A statement beginning with ON does not contain a GOTO or GOSUB clause.
Please say HELLO	Message printed by the LOGIN system program. User not logged into the system has typed something other than a legal, logged-out command to the system.
?Please use the RUN command	A transfer of control (as in a GOTO, GOSUB or IF-GOTO statement) cannot be performed from immediate mode.
?PRINT-USING buffer overflow	Format specified contains a field too large to be manipulated by the PRINT-USING statement.
?PRINT-USING format error	An error was made in the construction of the string used to supply the output format in a PRINT-USING statement.
?Program lost-Sorry	A fatal system error has occurred which caused the user program to be lost. This error can indicate hardware problems or use of an improperly compiled program. Consult the system manager or the discussion of such errors in the *RSTS/E System Manager's Guide*.

Message Printed	Meaning
?Redimensioned array	Usage of an array or matrix within the user program has caused BASIC-PLUS to redimension the array implicitly.
?RESUME and no error	A RESUME statement was encountered where no error had occurred to cause a transfer into an error handling routine via the ON ERROR GOTO statement.
?RETURN without GOSUB	RETURN statement encountered in user program without a previous GOSUB statement having been executed.
%SCALE factor interlock	An attempt was made to execute a program or source statement with the current scale factor. The program runs but the system uses the scale factor of the program in memory rather than the current scale factor. Use REPLACE and OLD or recompile the program to run with the current scale factor. (C)
?Statement not found	Reference is made within the program to a line number which is not within the program.
Stop	STOP statement was executed. The user can usually continue program execution by typing CONT and the RETURN key.
?String is needed	A number or variable name was used where a character string was necessary.
?Syntax error	BASIC-PLUS statement was incorrectly formatted.
?Too few arguments	The function has been called with a number of arguments not equal to the number defined for the function.
?Too many arguments	A user-defined function may have up to five arguments.
?Undefined function called	BASIC-PLUS interpreted some statement component as a function call for which there is no defined function (system or user).
?What?	Command or immediate mode statement entered to BASIC-PLUS could not be processed. Illegal verb or improper format error most likely.
?Wrong math package	Program was compiled on a system with either the 2-word or 4-word math package and an attempt is made to run the program on a system with the opposite math package. Recompile the program using the math package of the system on which it will be run.

APPENDIX D
BASIC-PLUS CHARACTER SET

User program statements are composed of individual characters. Allowable characters come from the following character set:

1. A through Z
2. 0 through 9
3. Space
4. Tab

and the following special symbols and keys:

Key	Use and Section in *BASIC-PLUS Language Manual*
$	Used in specifying string values (Chapter 5), or as the System Library file designator *(RSTS/E System User's Guide).*
%	Used in specifying integer values (Chapter 6). Also denotes account [1,4].
' "	Used to delimit string constants, i.e., text strings (Chapter 5).
!	Begins comment part of a line (Section 3.1). Also denotes account [1,3].
\ :	Separates multiple statements on one line (Section 2.4.1). The colon is accepted for compatibility with previous versions.
#	Denotes a device or file channel number, or is used as an output format effector (Chapter 10). Also denotes account number using current project number with a programmer number of 0.
,	Output format effector and list terminator (Section 3.3.2).
;	Output format effector (Section 3.3.2).
&	If EXTEND Mode is in effect, can be used at the end of a line to indicate that the current statement is continued on the next line. Denotes account [1,5].
@	Denotes the assignable account.
LINE FEED	When used at the end of a line, indicates that the current statement is continued on the next line (Section 2.4.2).
.	Valid variable-name character in Extend Mode. Also used in graphic representations of floating point numbers and strings to denote a decimal point. Delimits filename extensions.
( )	Used to group arguments in an arithmetic expression (Section 2.6.3), or to delimit project-programmer number.
[ ]	Used to group project-programmer number. Equivalent to ( ).
< >	Used to delimit file protection codes.
+ - */^	Arithmetic operators (Section 2.6.3).

Key	Use and Section in ***BASIC-PLUS Language Manual***
=	Replacement operator (Section 3.2). Relational equivalence operator (Section 2.6.4 and 5.1.5).
<	Logical "less than" operator (Sections 2.6.4 and 5.1.5).
>	Logical "greater than" operator (Sections 2.6.4 and 5.1.5).
==	Numeric "approximately equal to" operator (Section 2.6.4).
	Relational "exactly equal to" string operator (Section 5.1.5).

The decimal values 128 through 255 can appear in character strings. For most practical purposes, the characters represented by N ($0 \geqslant N \geqslant 127$) and N + 128 are the same. However, the characters returned from the function CHR$(N) and CHR$(N + 128) do not test as equal when compared. These values may also influence device-dependent operations. (See the *RSTS/E Programming Manual*.)

Decimal Value	ASCII Character	RSTS Usage
0	NUL	FILL character
1	SOH	
2	STX	
3	ETX	CTRL/C
4	EOT	CTRL/D
5	ENQ	
6	ACK	
7	BEL	BELL (CTRL/G)
8	BS	BACKSPACE
9	HT	HORIZONTAL TAB
10	LF	LINE FEED
11	VT	VERTICAL TAB
12	FF	FORM FEED (CTRL/L)
13	CR	CARRIAGE RETURN
14	SO	
15	SI	CTRL/O
16	DLE	
17	DC1	XON (CTRL/Q)
18	DC2	
19	DC3	X OFF (CTRL/S)
20	DC4	
21	NAK	CTRL/U
22	SYN	
23	ETB	
24	CAN	
25	EM	
26	SUB	CTRL/Z
27	ESC	ESCAPE [1]
28	FS	
29	GS	
30	RS	
31	US	
32	SP	SPACE
33	!	
34	"	
35	#	
36	$	
37	%	
38	&	
39	“	
40	(	
41	)	
42	*	
43	+	
44	'	
45	-	
56	.	
47	/	
48	0	
49	1	
50	2	
51	3	
52	4	
53	5	
54	6	
55	7	
56	8	
57	9	
58	:	
59	;	
60	<	
61	=	
62	>	
63	?	
64	@	
65	A	
66	B	
67	C	
68	D	
69	E	
70	F	
71	G	
72	H	
73	I	
74	J	
75	K	
76	L	
77	M	
78	N	
79	O	
80	P	
81	Q	
82	R	
83	S	
84	T	
85	U	
86	V	
87	W	
88	X	
89	Y	
90	Z	
91	[	
92	\	
93	]	
94	^ OR ↑	
95	— OR ←	
96	` Grave accent	
97	a	
98	b	
99	c	
100	d	
101	e	
102	f	
103	g	
104	h	
105	i	
106	j	
107	k	
108	l	
109	m	
110	n	
111	o	
112	p	
113	q	
114	r	
115	s	
116	t	
117	u	
118	v	
119	w	
120	x	
121	y	
122	z	
123	{	
124	| Vertical Line	
125	}	
126	~ Tilde	
127	DEL RUBOUT	

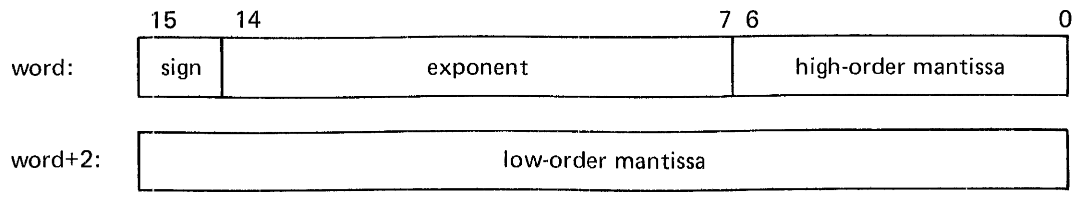
[1]ALTMODE (ASCII 125) or PREFIX (ASCII 126) keys which appear on some terminals are translated internally into ESCAPE.

APPENDIX E
RSTS/E FLOATING-POINT AND INTEGER FORMATS

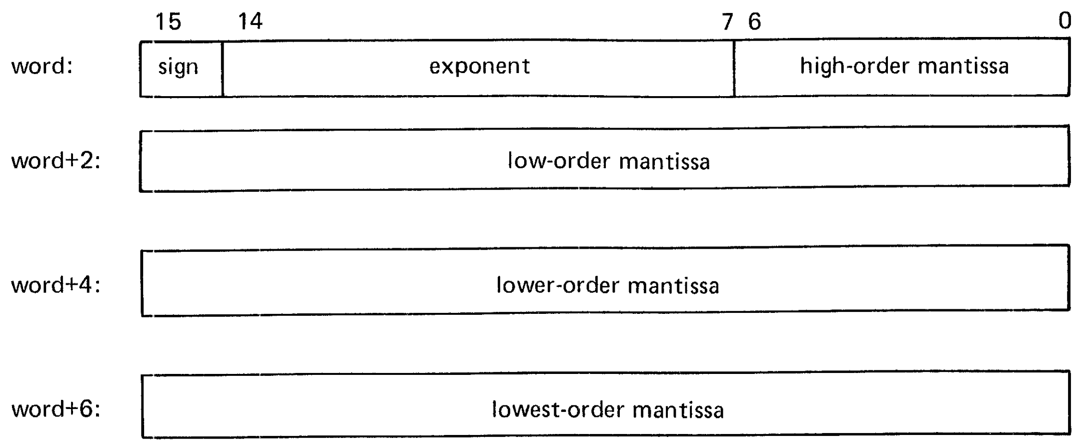
E.1 FLOATING-POINT FORMATS

RSTS/E systems use two standard floating-point packages: the single precision, 2-word package or the double precision, 4-word package. The determination of which package will be used is made by the system manager at the time the BASIC-PLUS runtime system is generated.

The single precision format provides economical storage, while the double precision format is used for high accuracy. The single precision format provides up to 24 bits of approximately seven decimal digits of accuracy. The magnitude range lies between .29 * 10 ^(-38) and 1.7 * 10 ^(38). Double precision calculations have a precision of 56 bits of approximately 16 decimal digits, with magnitudes in the same range as for single precision format.



SINGLE PRECISION FORMAT (2 WORD)



DOUBLE PRECISION FORMAT (4 WORD)

The exponent is stored in excess 128 (200_8) notation. Exponents from –127 to +127 are represented by the binary equivalent of 1 through 255 (1 through 377_8). Fractions are represented in sign magnitude notation with the binary radix point to the left. Numbers are assumed to be normalized and, therefore, the most significant bit is not stored because it is redundant (this is called "hidden bit normalization"); it is always a 1 unless the exponent is 0 in which case it is assumed to be 0. The value 0 is therefore represented by two or four words of 0's. For example: +1 would be represented by:

word:	040200
word+2:	000000

in the 2-word format, or:

word:	040200
word+2:	000000
word+4:	000000
word+6:	000000

in the 4-word format. -5 would be:

word:	140640
word+2:	000000

in the 2-word format, or:

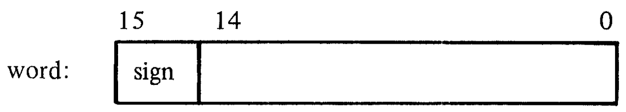
word:	140640
word+2:	000000
word+4:	000000
word+6:	000000

in the 4-word format.

While it is generally possible to run programs written on one RSTS system on another RSTS system, certain restrictions apply if the math packages are not the same. These are:

1. Programs depending on 4-word accuracy cannot be run with the 2-word package.
2. .BAC compiled programs cannot be interchanged. The program source file must be recompiled.
3. Floating-point virtual core array file formats are not compatible between math packages.
4. Programs using the Record I/O functions CVT$F and CVTF$ are not compatible between math packages.

E.2 INTEGER FORMAT



Integers are stored in a 2's complement representation. Integer values must be in the range -32768 to +32767. For example:

$+22 = 000026_8$
$-7 = 177771_8$

As a rule, an integer value is assumed by RSTS only where a constant or variable name is followed by a % character. Otherwise, constants and variables are assumed to be floating-point values.

BIBLIOGRAPHY

1. *101 BASIC Computer Games*
   Digital Equipment Corporation. 1975. Maynard, Mass.

2. *Teach Yourself BASIC,* Volume 1 and Volume 2 (self teaching wordbook)
   Robert L. Albrecht
   Technical Education Corporation. 1970

3. *Programming Time-Shared Computers in BASIC Language*
   Eugene H. Barnett
   Wiley-Interscience Books. 1972

4. *An Introduction to Computer Programming BASIC Language*
   James S. Coan
   Hayden Book Company, Inc. 1970

5. *Introduction to Programming: A BASIC Approach*
   Van C. Hare, Jr.
   Harcourt Brace Jovanovich, Inc. 1970

6. *BASIC Programming,* Second Edition
   John G. Kemeny and Thomas E. Kurtz
   John Wiley and Sons, Inc. 1971

7. *Introduction to Computing Through the BASIC Language*
   R.L. Nolan
   Holt, Rinehart Winston, Inc. 1974

8. *Computer Programming in BASIC*
   Joseph P. Pavlovich and Thomas E. Tahan
   Holden-Day Co. 1971

9. *Simplified BASIC Programming*
   Gerald A. Silver
   McGraw-Hill Co. 1974

10. *Programming in BASIC, with Applications*
    Bernard M. Singer
    McGraw-Hill Co. 1973

11. *Fundamentals of Digital Computers* (elementary and historical)
    Donald D. Spencer
    Howard W. Sams and Co., Inc. 1969

INDEX

INDEX (Cont.)

INDEX (Cont.)

INDEX (Cont.)

INDEX (Cont.)

BASIC-PLUS
Language Manual
DEC-11-ORBPB-B-D

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 cut along this line.

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 ____________

Fold Here

Do Not Tear - Fold Here and Staple

FIRST CLASS
PERMIT NO. 33
MAYNARD, MASS.

BUSINESS REPLY MAIL
NO POSTAGE STAMP NECESSARY IF MAILED IN THE UNITED STATES

Postage will be paid by:

digital

Software Documentation
146 Main Street ML 5-5/E39
Maynard, Massachusetts 01754