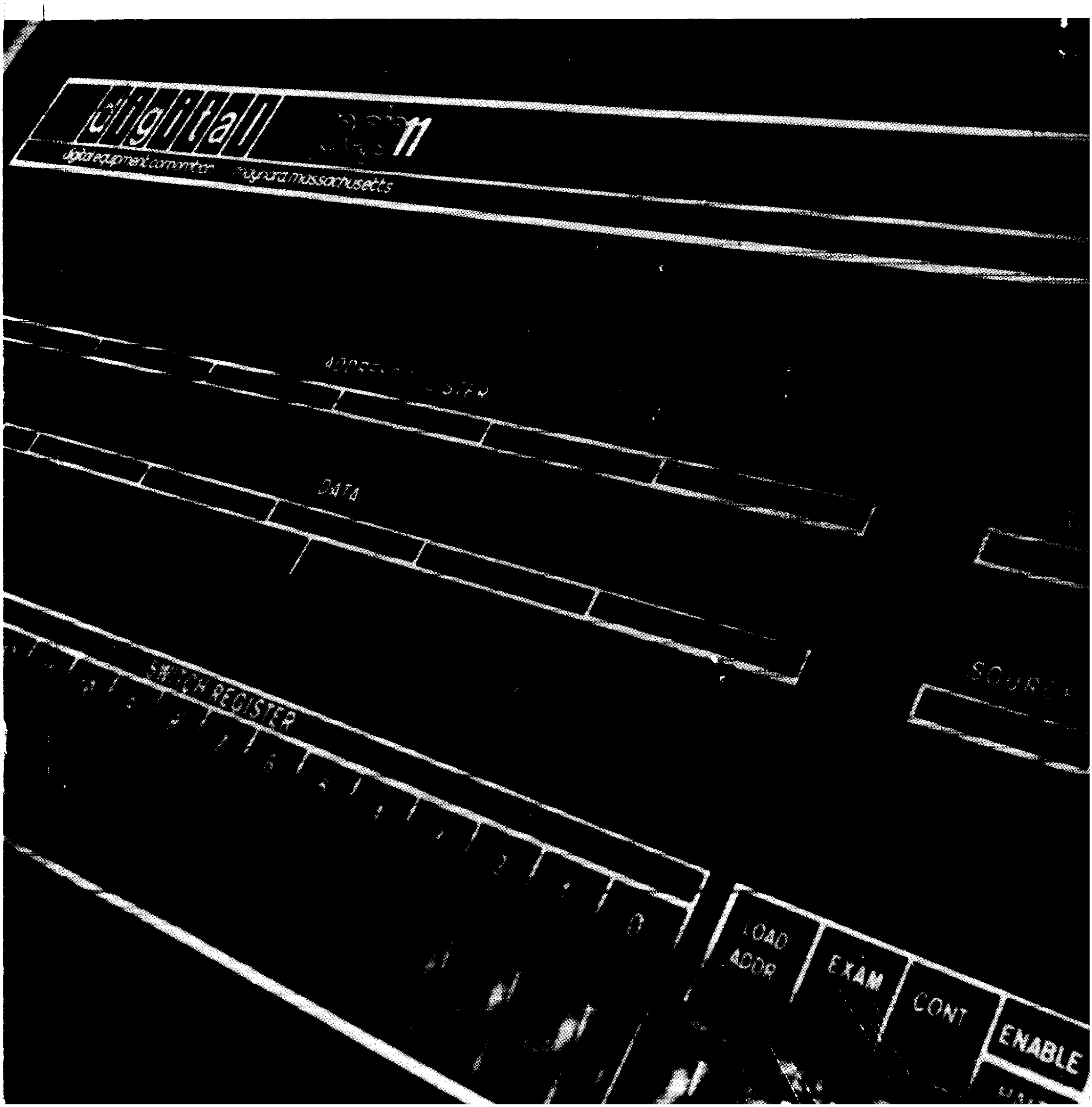
digital

pdp11

Resource Timesharing System
RSTS-11 BASIC-PLUS
user's guide



PL-11-71-Ø1-Ø1-A-D

PDP-11

RESOURCE TIME-SHARING SYSTEM (RSTS-11) USER'S GUIDE

BASIC-PLUS PROGRAMMING LANGUAGE

FOR ADDITIONAL COPIES, ORDER No. PL-11-71-Ø1-Ø1-A-D FROM DIGITAL EQUIPMENT CORPORATION, DIRECT MAIL, BUILDING 1-1, MAYNARD, MASS. 01754.

FIRST PRINTING, MAY 1971

YOUR ATTENTION IS INVITED TO THE LAST TWO PAGES OF THIS DOCUMENT. THE READER'S COMMENTS PAGE, WHEN COMPLETED AND RETURNED, IS BENEFICIAL TO BOTH YOU AND DEC. ALL COMMENTS RECEIVED ARE CONSIDERED WHEN DOCUMENTING SUBSEQUENT MANUALS. THE HOW TO OBTAIN SOFTWARE INFORMATION PAGE OFFERS YOU A MEANS OF KEEPING UP TO DATE WITH DEC'S SOFTWARE.

SUPPORTING AND REFERENCED DOCUMENTS:

RSTS-11 SYSTEM MANAGER'S GUIDE
(ORDER No. PL-11-71-Ø2-Ø1-A-D)

THESE AND OTHER DEC DOCUMENTS CAN BE ORDERED FROM DIGITAL EQUIPMENT CORPORATION, DIRECT MAIL, BLDG. 1-1, MAYNARD, MASSACHUSETTS 01754

PREFACE

This manual contains a comprehensive description of the PDP-11 Resource Time-Sharing System, RSTS-11. It is organized for the benefit of the beginning programmer, as it allows the reader to gradually acquire increased programming capabilities.

The BASIC-Plus language is an extension of BASIC[1] as originally developed at Dartmouth College. The experienced BASIC programmer may find the appendices sufficient for his use. However, BASIC-Plus offers many features not found in standard Dartmouth BASIC or any other version of BASIC.

BASIC-Plus incorporates the following special features:

1. Matrix Computations, a special set of 13 commands are available for performing matrix computations.

2. Alphanumeric String Capabilities, alphabetic and/or alphanumeric strings can be manipulated with the same ease as numeric data. Individual characters within these strings can be easily accessed by the user.

3. Program Control and Storage Facilities, facilities are included for storing both programs and data on any mass storage device (such as DECdisk or DECtape) and later retrieving them for use during program execution. Programs can be entered from the RSTS terminal paper tape reader as well as from the high-speed paper tape reader available on the computer. The manipulation of non-disk data files is a totally new concept in BASIC programming and will be greatly appreciated by the more experienced BASIC programmer. Lack of data storage facilities has always hampered BASIC from becoming as useful a language as, for example, FORTRAN. With this ability and the ease of learning the BASIC language, the new user has an extremely powerful tool at his command.

4. Program Editing Facilities, an existing program can be edited by adding or deleting lines, renaming the program, or resequencing the line numbers. The user can combine two programs into a single program, and request the listing of a program, either in whole or in part on his terminal or on a line printer.

5. Formatting of Output, controlled formatting of program output includes facilities for tabs, spaces, and the printing of column headings, as well as precise specification of the output line formatting.

[1] BASIC is a registered trademark of the Trustees of Dartmouth College.

6. Immediate Mode of Operation, commands typed by the user are immediately executed by BASIC-Plus instead of being stored for later execution.

7. Access to System Peripheral Equipment, the user program is able to perform input and output with various equipment, such as paper tape reader/punch, disk, DECtape, line printer, and card reader. Other peripherals will be available on the system in the future.

8. Documentation and Debugging Aids, the insertion of remarks and comments within a program is made somewhat more simple in this version of BASIC. Debugging of programs is aided by the printing of meaningful diagnostic messages which pinpoint syntactical and other errors detected during the program execution.

A minimum RSTS-11 system requires a PDP-11/20 computer, user terminals for program input and output, and a sufficient amount of program storage area (core memory, usually 20K) to accommodate several users simultaneously. Mass storage devices (DECtape and DECdisk) are included on the standard system, although additional mass storage can be connected at a later date. The system will support additional core, Extended Arithmetic Element, line printer, card reader, and high-speed reader/punch.

CONTENTS

CHAPTER 7 DATA STORAGE CAPABILITIES

CHAPTER 8 MATRIX MANIPULATION

CHAPTER 9 ADVANCED STATEMENT FEATURES

CHAPTER 10 BASIC-PLUS INPUT AND OUTPUT OPERATIONS

PART III

USING RSTS-11

CHAPTER 11 BASIC-PLUS SYSTEM COMMANDS

APPENDICES

PART I

RSTS-11 AND THE BASIC LANGUAGE

This first of three parts describes the RSTS-11 system, its hardware and user features, and the simplest level of the BASIC language. BASIC as described in this part is essentially Dartmouth BASIC as originally developed. Part II describes the extended capabilities of BASIC-Plus. As part of the introductory material, the reader will find references to some of the extended capabilities.

As a language, BASIC is easy to learn. BASIC-Plus allows for capabilities to keep BASIC a useful tool for the more sophisticated programmer. BASIC does not, however, penalize the beginning user. Almost any problem can be solved with the statements available in Part I. The statements and features in Part II allow the user to write more efficient code and better use his time and core space.

CHAPTER 1

AN INTRODUCTION TO RSTS-11

Although a computing system such as RSTS-11 is a complicated arrangement of interrelated computer programs, the RSTS-11 user need only be concerned with the writing and execution of programs in the BASIC-Plus language. This manual, therefore, assumes that the user is always (and only) communicating with BASIC.

1.1 INTRODUCTION TO PROGRAMMING

For the benefit of the new programmer approaching his first computing experience, there are four phases in programming a computer:

a. writing the computer program,
b. entering the program to the computing system,
c. testing and debugging the program, and
d. running the finished program.

BASIC-Plus is the language in which the user writes programs designed for the RSTS-11 system. Input of the completed program is generally performed from the terminal keyboard on a time-sharing system. A program can be input through various peripheral devices, such as the paper tape reader, magnetic tape (DECtape), or punched cards; however, the initial creation of a BASIC program is usually performed on-line to the computer with the terminal keyboard.

Ideally, a program runs correctly as written, but 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 first typed at the keyboard and appropriate error messages are printed. BASIC-Plus also evaluates the entire program for commonly made errors and generates messages which explain the mistakes to the user. Program errors are corrected on-line from the terminal keyboard.

The testing and debugging process is continued until the program appears to execute correctly. This is a good time to explain to the new user that a computer program only does what the programmer has written. The calculations performed by the computer are not necessarily those that will produce the correct results. In order to obtain correct results from a computer, the user must write a program which is not only free of detectable errors, but one which correctly analyzes his problem.

RSTS-11 provides keyboard commands which enable the user not only to create and execute his program but also 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.

1.2 THE BASIC-PLUS PROGRAMMING LANGUAGE

BASIC-Plus (which is hereafter referred to simply as BASIC in most cases) is one of the simplest of all programming languages because of its small number of powerful but self-explanatory statements and commands and its easy application in solving problems. Its wide use in scientific, business, and educational installations attests to its value and straightforward application.

BASIC is similar to many other programming languages in various respects (and is, consequently, very easy for the experienced programmer to learn), but is especially suited for time-sharing because of its conversational nature. A conversational language is one which allows the user to communicate with the language processor by typing on the terminal keyboard. BASIC responds by printing on the terminal printer, providing for an interactive man/machine relationship.

BASIC-Plus contains both elementary statements used to write simple programs and advanced programming techniques and statements to write complex and efficient programs. The key word here is not complex, but efficient. As the user progresses and gains programming experience, he will naturally find himself becoming more efficient and able to use the more sophisticated data manipulations. Almost any problem can be solved with the simple BASIC statements. Later in the user's programming experience, the advanced techniques can be added.

1.3 CONVENTIONS USED IN THIS MANUAL

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

a. Angle brackets indicate essential elements of the statement or command being described. For example:

line number LET <*variable*> = <*expression*>

b. Square brackets indicate a choice among two or more possibilities. For example:

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

c. Braces indicate optional matter or a choice among optional elements:

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

d. Items in lower case type (*formula*, *variable*, etc., above) are supplied by the user according to rules explained in the test. Items in capital letters (LET, IF, THEN, etc.) must appear exactly as shown because they form the BASIC language.

e. The term *line number* used in examples (as in (c) above) indicates that any line number is valid.

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:

a. BASIC (that is, the computer) prints on the teleprinter whereas the user types on the keyboard.

b. A statement is a line (or part of a line or multiple lines in some cases, see sections 2.3.1 and 2.3.2) within a user program containing a BASIC language instruction. Each line is preceded by a line number. A line is terminated by typing the RETURN key.

c. Commands cause BASIC to perform some operation or task immediately and are not preceded by a line number. Commands are always terminated with the RETURN key.

d. User programs consist of a series of statements written by a person using the system in the BASIC-Plus language.

e. The RSTS-11 terminal is in most cases an ASR-33 Teletype[1]. However, RSTS-11 can accommodate virtually any typewriter type device. The RSTS-11 user terminal is alternatively referred to as terminal, teleprinter, or keyboard, depending upon what part or whether the whole device is indicated.

[1] Teletype is a registered trademark of the Teletype Corporation.

1.4 ON-LINE WITH RSTS-11

In order to use the RSTS-11 system, the user should sit down at a terminal and turn the LINE-OFF-LOCAL knob to LINE. The user should then type

```
HELLO
```

followed by the RETURN key. The system responds by skipping a line at the terminal and printing a # character. The user should follow this with his project-programmer numbers (assigned by the system manager). The two numbers are separated by a comma and entered to the system with the RETURN key. The system then prints:

```
PASSWORD:
```

and waits for the user to type the password code assigned to him by the system manager. This code does not echo as it is typed by the user in order to maintain the security of controlled system usage. If the entry codes typed are acceptable to the system, the message:

```
WELCOME TO RSTS-11
NEW OR OLD--
```

is printed. In reply, the user can indicate the creation of a NEW program or recalling of an OLD one. Other system commands are explained in PART III.

The entire process of preparing to enter a new program onto the system might look as follows. Notice that 1,2 is the project-programmer number, the password is not echoed at the terminal, a NEW program is to be created, and its name is MATRIX. The system prints READY to indicate that it is able to accept BASIC input.

```
HELLO

#1,2
PASSWORD:


WELCOME TO RSTS-11
NEW OR OLD--NEW
NEW FILE NAME--MATRIX


READY
```

Once a program has been typed and minor typing errors corrected, the program can be made to execute by typing RUN or RUNNH. The RUNNH form omits printing the name of the program and the current date before the program output. The choice belongs to the user whether he wishes this data or not. By typing LIST or LISTNH (NH stands for "no heading"), the user can obtain a clean listing of the program. These two commands (RUN and LIST) are used frequently throughout this manual. See Figure 2-1 for an example.

1.5 SPECIAL TERMINAL KEYS

Throughout this manual, reference is made to typing various special keys on the RSTS-11 terminal. In many cases, these keys are not mentioned, but assumed. The user will quickly learn the use of the more important control keys on the terminal. As an introduction, the user is directed to consider the keys explained below. All special keys are described in Chapter 11.

The <u>RETURN key</u> causes two operations to be performed:

a. An automatic carriage return/line feed operation is executed. The printing head returns to the beginning of the line (carriage return) and the paper is advanced one line (line feed).

b. The data preceding the typing of the RETURN key is entered into the system for evaluation. All commands to BASIC and lines in a user program are terminated by typing the RETURN key.

The <u>RUBOUT key</u> is used to correct typing mistakes. Typing this key once causes the last character typed to be deleted from the terminal input buffer (remember that an entire line is entered at once when the RETURN key is typed). Pressing the RUBOUT key N times causes the last N characters typed to be deleted.

The ESCAPE key (ESC or ALT MODE on different terminals) performs the same function as the (b) description of the RETURN key. The ESCAPE key prints a $ character, terminates the current input line, and does not cause a carriage return/line feed operation.

The <u>CTRL key</u> (or control key) is used in combination with certain letter keys to cause BASIC to perform special operations. These combinations are performed by the user holding down the CTRL key while typing the desired letter key, then releasing both keys. CTRL/U and

CTRL/C are examples of these combinations and how they are shown in the test. Some of the CTRL/key combinations are introduced below for use when working through this manual. All usable combinations are described in Chapter 11.

a. CTRL/U is used to delete an entire line up to the last point at which the RETURN or ESCAPE key was last typed. BASIC responds with a carriage return/line feed so that the user can continue typing on a fresh line.

b. CTRL/C is used to interrupt the execution of a program and return to the interactive BASIC processor. When typed by the user, CTRL/C causes the system to echo ↑C and when BASIC is able to accept commands, the system prints READY. (READY is printed instantaneously after typing CTRL/C.)

The LINE FEED key serves as a "local" RETURN key, allowing a user to type a logical BASIC program line longer than the 72 characters which can be typed on one line on the teleprinter. Anything typed on the line subsequent to the typing of the LINE FEED key is treated as if it were part of the preceding line. ~~The statement to be broken into two or more lines can be of any length.~~ For example:

```
1Ø REM THIS IS A PROGRAM WHICH
       COMPUTES AND PRINTS THE STATISTICAL
       RESULTS OF A CENSUS SURVEY
```

The message within the REM statement is 84 characters long, but is spaced over three lines to make it more conspicuous. The 91 characters in the whole REM statement would not fit on a single teleprinter line. Notice that the LINE FEED key does not cause a character to be printed.

CHAPTER 2

FUNDAMENTALS OF PROGRAMMING IN BASIC-PLUS

2.1 EXAMPLE BASIC PROGRAM

The program in Figure 2.1 is an example of a user program written in the BASIC-Plus language. It illustrates the syntax and elements of the language as well as standard formatting of statements and the appearance of terminal output.

The user program (the lines numbered 10 through 200) may at this time mean little, although the remark in the first line (line 10) and the printed results (following the word RUN) clearly show that the program computes interest payments.

A user program is composed of lines of statements containing instructions to BASIC. Each line of the program begins with a line number that serves to identify that line as a statement and to indicate the order in which statements are to be evaluated for execution. Each statement starts with an English word specifying the type of operation to be performed.

2.2 LINE NUMBERS

Each line of a user program is preceded by a line number. Line numbers:

a. indicate the order in which statements are normally evaluated;

b. 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); and

c. enhance program debugging by permitting modification of any specified line without affecting any other portion of the program (see section 11.4).

Line numbers are in the range 1 to 32767. It is good programming practice to number lines in increments of 5 or 10 when first writing a program, to allow for insertion of forgotten or additional lines when debugging the program.

```
LISTNH
 1Ø REMARK - THIS PROGRAM COMPUTES INTEREST PAYMENTS
 2Ø INPUT "INTEREST IN PERCENT";J
 3Ø LET J=J/1ØØ
 4Ø INPUT "AMOUNT OF LOAN"; A
 5Ø INPUT "NUMBER OF YEARS"; N
 6Ø INPUT "NUMBER OF PAYMENTS PER YEAR"; M
 7Ø LET N=N*M: I=J/M: B=1+I
 8Ø LET R=A*I/(1-1/B↑N)
 9Ø PRINT
 1ØØ PRINT " AMOUNT PER PAYMENT ="; INT(R*1Ø↑2+.5)/1Ø↑2
 11Ø PRINT "TOTAL INTEREST      ="; INT((R*N-A)*1Ø↑2+.5)/1Ø↑2
 12Ø PRINT
 13Ø LET B=A
 14Ø PRINT "INTEREST   APP TO PRIN     BALANCE OF PRIN"
 15Ø LET L=B*I: P=R-L: B=B-P
 16Ø PRINT INT(L*10↑2+.5)/1Ø↑2, INT(P*1Ø↑2+.5)/1Ø↑2,
           INT(B*1Ø↑2+.5)/1Ø↑2
 17Ø IF B>=R GOTO 15Ø
 18Ø PRINT INT((B*I)*1Ø↑2+.5)/1Ø↑2, INT((R-B*I)*1Ø↑2+.5)/1Ø↑2
 19Ø PRINT "LAST PAYMENT ="; INT((B*I+B)*1Ø↑2+.5)/1Ø↑2
 2ØØ END

 READY

 RUNNH
 INTEREST IN PERCENT? 7.5
 AMOUNT OF LOAN? 25ØØ
 NUMBER OF YEARS? 2
 NUMBER OF PAYMENTS PER YEAR? 4

 AMOUNT PER PAYMENT =339.44
 TOTAL INTEREST     =215.51

 INTEREST   APP TO PRIN     BALANCE OF PRIN
 46.88         292.56         22Ø7.44
 41.39         298.Ø5         19Ø9.39
 35.8          3Ø3.64         16Ø5.75
 3Ø.11         3Ø9.33         1296.42
 24.31         315.13         981.29
 18.4          321.Ø4         66Ø.25
 12.38         327.Ø6         333.19
 6.25          333.19
 LAST PAYMENT =339.44

 READY
```

Figure 2-1

EXAMPLE BASIC PROGRAM

When the program is executed (with the use of the RUN command), BASIC evaluates the statements in the order of their line numbers, starting with the smallest line number and going to the largest (regardless of the order in which they were typed or entered).

2.3 STATEMENTS

Each line number is followed by an English word. The word identifies the type of statement and informs BASIC what to do or how to treat the data (if any) which follows the word.

2.3.1 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. Thus 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, while

```
20  LET X=1:  PRINT X,Y,Z:  IF X=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 the discussion of the individual statements.

2.3.2 A Single Statement on Multiple Lines

A single statement can be continued on the next line of the program. To indicate that a statement is to be continued, the line is terminated with the LINE FEED key instead of the RETURN key. The LINE FEED performs a carriage return/line feed operation on the terminal and the line to be continued does not contain a line number. For example:

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

where the first line was terminated with the LINE FEED key is equivalent to:

```
    1Ø LET W7=(W-X4*3)*(Z-A/(A-B)-17)
```

Note that the LINE FEED key does not cause a printed character to appear on the page.

The length of a multiple line statement is limited to 255 characters.

2.4 CHARACTER SET

User program statements are composed of individual characters. Allowable characters come from the following character set:

A through Z
Ø through 9

and the following special symbols and keys.

Key	Function
$	Used in specifying string variables (section 5.1).
%	Used in specifying integer variables (section 6.1).
' "	Used to delimit string constants, i.e., text strings (section 5.1).
!	Begins comment part of a line (section 3.1).
:	Separates multiple statements on one line (section 2.3.1).
#	Denotes a device or filename, or is used as an output format effector (Chapter 7 and section 9.9).
,	Output format effector and list terminator (section 3.11).
;	Output format effector~~x~~ (section 3.11).
LINE FEED	When used at the end of a line, indicates that the current statement is continued on the next line (section 2.3.2).
()	Used to group arguments in an arithmetic expression (section 2.5).
+ - = * / ↑	Arithmetic operators (section 2.5.3).

Spaces can be used freely throughout the program to make statements easier to read. For example:

```
10 LET B = D↑2+1
```

instead of:

```
10LETB=D↑2+1
```

Both of the above statements mean the same thing to BASIC and are stored exactly the same within the computer when the program is executed.

TABS, like spaces, are used to make a program easy to read. TAB is generated by SHIFT/I

2.5 EXPRESSIONS

An expression is a group of symbols which can be evaluated by BASIC. Expressions are composed of numbers, variables, functions, or a combination of the preceding, separated by arithmetic or relational operators. Expressions are created by the programmer and inserted into the standard BASIC statements in order to perform the various operations which comprise the user program.

The following are examples of expressions acceptable to BASIC-Plus.

```
4
A7*(B↑2+1)
X<Y
((A>B) OR (C=D)) AND A/B<>C/D
```

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 expressions.

2.5.1 Numbers

Numbers, called numeric constants because they retain a constant value throughout a program, can be positive or negative and can contain up to eight digits. Numeric constants are written using decimal notation, as follows:

```
2
-3.675
1234.5678
-1234567.8
-.00000078
```

The following are not acceptable numbers in BASIC:

$\frac{14}{3}$

$\sqrt{7}$

However, BASIC can find the decimal expansion of those two mathematical formulas as shown below:

$\frac{14}{3}$ is expressed as 14/3

$\sqrt{7}$ is expressed as SQR(7)

These formats are explained further in later sections.

The letter E allows further flexibility in number representation If numbers were limited to eight digits, a computer would not be able to solve many problems involving large numbers. Consequently, rather than saying that BASIC can only accept numbers with a maximum of eight digits, we say that BASIC has eight digits of precision. Larger numbers can be written using the letter E to indicate "times ten to the power," thus:

```
.ØØØ12345678     can be written in BASIC as 123.45678E-6
1234567890.      can be written in BASIC as 12345679E 2
-123456789ØØ.    can be written in BASIC as -1.2345679E 1Ø
```

This E format representation of numbers is very flexible in that the number .001 can be written as 1E-3, .01E-1, 100E-5, or any number of ways. If more than eight 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-Ø4; 1E Ø4.)

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.

Numbers are specified according to the following rules:

a. line numbers are unsigned decimal integers in the range 1 to 32767.

b. integers are signed decimal numbers in the range -32767 to +32767. (When using numbers on a computer, a comma is never used to separate the thousands place from the hundreds place or anywhere else within a number.)

c. floating point numbers have the absolute range 1E-9800≤n≤1E9800.

2.5.2 Variables

A variable is a data item whose value can be changed by the programmer. A numeric variable is denoted by a single letter or by a letter followed by a single digit. Thus BASIC interprets E8 as a variable, along with A, X, N5, LØ, and O1. (Subscripted , Integer, and character string variables are described in later sections.)

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 when the variable is incremented by a FOR statement. (These conditions are explained further in later sections.) All variables are set equal to zero (Ø) before program execution. Consequently it is only necessary to assign a value to a variable when an initial value other than zero is required.

2.5.3 Mathematical Operators

BASIC 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; they are as follows:

Symbol	Example	Meaning
+	A+B	Add B to A
-	A-B	Subtract B from A
*	A*B	Multiply A by B
/	A/B	Divide A by B
↑	A↑B	Calculate A to the B power, A^B

When more than one operation is to be performed in a single formula, as is most often the case, rules are observed as to the precedence of the above operators. The arithmetic operations are performed in the following sequence, with (a) having the highest precedence:

Unary plus and minus are also allowed.

a. Any formula inside 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.

b. In the absence of parentheses in a formula, BASIC performs operations as follows:

   1. Unary minus
   2. exponentiation
   3. multiplication and division
   4. addition and subtraction

c. In the absence of parentheses in a formula involving more than one operation on the same level in (b) 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 formula (or expression) A+B*C↑D is evaluated as follows:

first,	C is raised to the D power
second,	the result of the first operation is multiplied by B
third,	the result of the previous operation is added to A.

Parentheses are used to indicate any other order of evaluation. For example, if it is the product of B and C that is to be raised to the D power, the expression would look as follows:

A+(B*C)↑D

If it is desired to multiply the quantity A+B by C to the D power:

(A+B)*C↑D

The user is encouraged to use parentheses even where they are not strictly required in order to make the formulas easier for his own reading. Ambiguities exist only in the programmer's mind, the computer always performs the operations as explained above.

2.5.4 Relational Symbols

Relational symbols are used in IF-THEN statements (see section 3.5) where it is necessary to compare values. The relational symbols are as follows:

Mathematical Symbol	BASIC 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. (That is, when printed, the two quantities are equal in value. Within the computer floating point numbers can differ by a miniscule amount in the last decimal place but still be considered equal for all practical purposes.)

2.5.5 Logical Operators

Logical operators are used in IF-THEN statements (see section 3.5) where some condition is used to determine subsequent operations within the user program. 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. If both A and B have logical values, A AND B is true if and only if both values are true and false if either or both are false.
OR	A OR B	The logical sum of A and B. If A and B have logical values, then A OR B is true if and only if at least one is true, and false if both are false.
XOR	A XOR B	The logical exclusive OR of A and B. If both A and B have logical values, A XOR B is true if and only if the values differ, and false if both values are equal.
IMP	A IMP B	The logical implication of A and B. If A and B have logical values, then A IMP B is false if and only if A is false and B is true; otherwise the value is true.
EQV	A EQV B	A is logically equivalent to B. If both A and B have logical values, A EQV B is true if the two values ~~differ~~ and false if they ~~are the same~~. are the same di.ffer

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. In logical operations, the only possible values a term can have are true and false (T and F).

		B	
		T	F
A	T	T	F
	F	F	F

AND

		B	
		T	F
A	T	T	T
	F	T	F

OR

		B	
		T	F
A	T	F	T
	F	T	F

XOR

		B	
		T	F
A	T	T	F
	F	F	T

EQV

		B	
		T	F
A	T	T	T
	F	F	T

IMP

A	
T	F
F	T

NOT

CHAPTER 3

ELEMENTARY BASIC STATEMENTS

This chapter describes the simplest forms of the more elementary BASIC 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 (such as automatic matrix manipulation) explained in Part II.

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 solution may not be as efficient. BASIC is a language in which lack of experience does not penalize the beginning user. As long as the user understands the details of his problem he will be able to represent it in BASIC 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. Such data 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 are useful things to have present in the program for ready reference by anyone using that program.

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

a. the REMARK statement, and
b. use of the exclamation mark (!).

The REMARK statement must be preceded by a line number. The word REMARK can be abbreviated to REM for typing convenience, and the message itself can contain any printing character on the keyboard. BASIC 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 the program execution.) Typical REM statements are shown below:

```
1Ø REM- THIS PROGRAM COMPUTES THE
11 REM- ROOTS OF A QUADRATIC EQUATION
```

The exclamation mark is used to terminate the statement part of a line and begin the comment part of the line. For example:

```
125 LET A=2+4*SQR(C)       !A SET EQUAL TO INITIAL VALUE
13Ø PRINT A/2+1            !PRINT SECOND CALCULATED VALUE
```

BASIC ignores everything on the line after encountering the exclamation mark.

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.

The lines below indicate three ways of putting the same remark on two lines. Lines 1Ø 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.

```
1Ø REM THIS PROGRAM COMPUTES THE
11 REM ROOTS OF A QUADRATIC EQUATION
2Ø REM THIS PROGRAM COMPUTES THE
       ROOTS OF A QUADRATIC EQUATION
3Ø ! THIS PROGRAM COMPUTES THE
     ROOTS OF A QUADRATIC EQUATION
```

3.2 LET STATEMENT

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

line number 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:

```
1Ø LET X=X+1
2Ø LET W2=(A4-X↑3)*(Z-A/B))
```

In line 10 the old value of X is increased by one and becomes the new value of X. In the second example, 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 A5=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 also allows the user to completely omit the word LET from what would ordinarily be called the LET statement. The user may find it easier to type:

```
1Ø X=12*(S+7)
```

than

```
1Ø 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, such as the following:

```
                    Z=
1Ø X=44: Y=X↑2+Y1: B2+3.5*A
```

Expressions in a LET statement can include both integer and floating point variables (section 2.5.2 and 6.1). (Constants are always considered floating point values.). Calculations involving data of one type yield results of the same type. If, however, one operand is an integer variable and one is a floating point variable, the result is a floating point value. The value resulting from the evaluation of any expression is stored according to the data type to the left of the equal sign. For example:

```
1Ø LET A=6.2
2Ø LET B%=A
3Ø PRINT A,B%
```

when executed, would result in the following being printed:

```
       6.2      6
```

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

```
        1Ø 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 gives the beginning user a preview of the techniques used in performing BASIC program I/O (an abbreviation for the term input/output). The most elementary forms of the PRINT, INPUT, READ, and DATA statements are described here so that the user is able to conceptualize the creation of BASIC programs.

Using the LET statement, already described, and the following executable statements, the user can easily write a viable BASIC program of the simplest sort. If he should want to try his program, these simple I/O statements will provide a means of doing so and obtaining tangible output.

These statements are described in detail at the end of this chapter and additional, more advanced, I/O techniques are described in later chapters.

The PRINT statement is used to output program results. The PRINT statement has the basic form:

line number PRINT {*list*}

where the optional list can consist of messages to be printed or numeric values, or both. Without the list, the PRINT statement

```
        1Ø PRINT
```

causes a carriage return/line feed to be performed at the teleprinter. In order to print numeric values, the word PRINT is followed by the variable or expression whose numeric value is to be printed. The PRINT statement, like the LET statement, can perform numeric calculations. For example:

```
        1Ø LET A=2: LET B=4
        2Ø PRINT (A+B)*2
```

causes the number 12 to be printed when line 20 is executed.

A message can be easily output on the teleprinter by enclosing the text to be printed in quotation marks, as follows:

```
7Ø PRINT "STUDENT NUMBER = ";X
```

causes the following to be printed (where X=7744):

```
STUDENT NUMBER = 7744
```

The READ and DATA statements are used to input data to a program during execution. A DATA statement contains values which are assigned to the variables within a READ statement. When the execution of the program encounters a READ statement of the form:

line number READ <*list*>

the BASIC processor assigns to the first variable in the list the first available value encountered in the pool of DATA statements within the program. The second variable is assigned the second value in the DATA pool, and so on. Variable names are separated by commas.

A DATA statement looks as follows:

line number DATA <*list*>

DATA statements are usually grouped together toward the end of a program. All of the DATA statements in a given program are considered to be one data pool from which subsequent READ statements obtain values. (The values in the list are separated by commas.) The DATA statements are referenced in the order of their line numbers. For example:

```
1Ø READ A,B,C
2Ø READ D,E,F
3Ø READ A,B,C
4Ø DATA 1,2,3,4
5Ø DATA 5,6,7,8,9
```

results in the following assignments being made:

$$\left.\begin{array}{l}A=1\\B=2\\C=3\end{array}\right\}\quad \text{when line 1Ø is executed}$$

```
D=4 \
E=5  }      when line 2Ø is executed
F=6 /
A=7 \
B=8  }      when line 3Ø is executed
C=9 /
```

The INPUT statement allows the user to enter data to the program from the terminal keyboard while the program is being executed. The data is typed by the user as BASIC asks for it. For example:

```
1Ø INPUT A,B,C
```

causes BASIC to pause during execution, print a question mark, and wait for the user to type three numerical values. The numbers must be separated by commas and terminated with the RETURN key. BASIC keeps printing question marks until it obtains the desired number of numeric inputs from the keyboard. For example, line 1Ø above would cause:

```
?
```

to be printed. The user could type:

```
?15,24
```

followed by the RETURN key. BASIC would reply:

```
?15,24
?
```

and wait for the user to enter a third value. Any values entered beyond the number required (three in the above case) would be ignored. INPUT statements are used only when small amounts of data are to be entered, or when data can only be supplied while the program is running.

3.4 UNCONDITIONAL BRANCH, GOTO STATEMENT

The GOTO statement is used when it is desired to unconditionally transfer to some line other than the next sequential line in the program. In other words, a GOTO statement causes an immediate jump to a specified line, out of the normal consecutive line number order of execution. The general format of the statement is as follows:

line number 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 possible to jump both forward and backward within a program.

Consider the following simple example:

```
1Ø LET A=2
2Ø GOTO 5Ø
3Ø LET A=SQR(A+14)
5Ø PRINT A,A*A
```

When executed, the above lines will cause the following to be printed:

```
2        4
```

When the program encounters line 20, control transfers to line 50; line 50 is executed, control then continues to the line following line 50. 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 must be the last statement on the line. For example:

```
11Ø LET A=ATN(B2):PRINT A:GOTO 5Ø
```

Any statement following the GOTO on the same line would never be executed.

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 the truth of some mathematical relation or relations. The basic format of the IF statement is as follows:

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

The specified condition is tested. If the relationship is found false, then control is transferred to the statement following the IF statement. If the condition is true, the statement following the THEN is is executed or control is transferred to the line number given after THEN or GOTO. (See also the IF-THEN-ELSE statement, section 9.5.)

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

A+2>B	is a relational expression
A>B AND B<=SQR(C)	is a logical expression

Either type of condition, when evaluated, is 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 2.5.5 and 2.5.6 and are presented in Tables 3-1 and 3-2 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 to the next line following line 75.

TABLE 3-1
RELATIONAL OPERATORS

Mathematical Symbol	BASIC Symbol	BASIC Example	Meaning
=	=	A = B	A equals B
<	<	A < B	A is less than B
$\leq$	<=	A<=B	A is less than or equal to B
>	>	A > B	A is greater than B
$\geq$	>=	A>=B	A is greater than or equal to B
$\neq$	<>	A<> B	A is not equal to B
$\approx$	==	A==B	A is approximately equal to B

TABLE 3-2
LOGICAL OPERATORS

Logical Operator	BASIC Example	Meaning
NOT	NOT A	the logical negation of A
AND	A AND B	the logical product of A and B
OR	A OR B	the logical sum of A and B
XOR	A XOR B	the logical exclusive OR of A and B
IMP	A IMP B	the logical implication of A and B
EQV	A EQV B	A is logically equivalent to B

Truth Tables
of Logical Operators

		B	
		T	F
A	T	T	F
	F	F	F

AND

		B	
		T	F
A	T	T	T
	F	T	F

OR

		B	
		T	F
A	T	T	T
	F	F	T

IMP

		B	
		T	F
A	T	T	F
	F	F	T

EQV

		B	
		T	F
A	T	F	T
	F	T	F

XOR

	B	
	T	F
A	F	T

NOT

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

```
25 IF A>B THEN IF B>C THEN PRINT "A>B>C"

25 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 line following line 25 is executed.

In the following example, the IF-GOTO statement in line 20 is used to limit the range of the variable A in line 10. Execution of the loop continues until the relationship A>4 is true, then immediately branches to line 55 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.)

```
1Ø LET A=A+1: X=A↑2
2Ø IF A>4 GOTO 55
25 PRINT X
3Ø PRINT "VALUE OF A IS" A
4Ø GOTO 1Ø
55 END
```

when the above loop is executed, the following is printed:

```
1
VALUE OF A IS 1
4
VALUE OF A IS 2
9
VALUE OF A IS 3
16
VALUE OF A IS 4
```

(The reader is advised to execute these short example programs by hand to follow the operation of the computer.)

In IF statements, the following priorities are associated with each operator, in order to provide unambiguous evaluation of the conditions specified:

a. expressions in parentheses are evaluated first

b. intrinsic mathematical functions

c. unary minus

d. exponentiation (↑)

~~c. unary minus (-), that is, a negative number~~ or
~~variable such as -3, -A, etc.~~

e. multiplication and division (* and /)

f. addition and subtraction (+ and -)

g. relational operators (=, <, <=, >, >=, ==, <>)

h. NOT

i. AND

j. OR and XOR

k. IMP

l. EQV

Within the operators indicated in any one group above, operations proceed from left to right.

Examples of IF-THEN statements follow:

```
10 IF A>B THEN 100        !SIMPLE COMPARISON
20 IF A=B OR B=C THEN 200
30 IF A>B THEN A=-B       !AN ASSIGNMENT BY A LET STATEMENT
40 IF X>Y IMP Y>Z THEN PRINT "QED"
```

~~The IF statement can be used anywhere in a multiple statement line except when a GOTO statement follows THEN. In this case, the IF statement must be the last statement on the line. (Remember, when using GOTO, subsequent statements on the same line can never be executed.)~~
If there are following statements on the line after an IF--THEN statement, the following statements are within the scope of the "THEN".

3.6 PROGRAM LOOPS

We mentioned loops in the section on the IF-THEN and IF-GOTO statement. Programs frequently involve performing certain operations a specific number of times. This is a task for which a computer is particularly well suited. 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 long before the human being attempting the same task even becomes bored. To write a loop, the programmer must ensure that the series of statements is repeated until a terminal condition is met.

Programs which use 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:

```
     1Ø PRINT 1, SQR(1)
     2Ø PRINT 2, SQR(2)
     3Ø PRINT 3, SQR(3)
           .
           .
           .
    99Ø PRINT 99, SQR(99)
   1ØØØ PRINT 1ØØ, SQR(1ØØ)
   1Ø1Ø END
```

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

```
     1Ø LET X=1
     2Ø PRINT X,SQR(X)
     3Ø LET X=X+1
     4Ø IF X<=1ØØ THEN 2Ø
     5Ø END
```

Statement 10 assigns a value of 1 to X, thus setting up the initial conditions of the loop. In line 20, both 1 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 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. BASIC now receives a negative answer to the question in line 40, does not return to line 20, but goes to line 50 which ends the program.

All program loops have four characteristic parts:

a. initialization, the conditions which must exist for the first execution of the loop are created (line 10 above)

b. the body of the loop in which the operation which is to be repeated is performed (line 20 above)

c. modification, which alters some value and makes each execution of the loop different from the one before and the one after (line 30 above)

d. terminal condition, once this exit test is satisfied the loop is considered completed and execution continues to the program statements following the loop (line 40 above)

3.6.1 FOR and NEXT Statements

The FOR statement is of the form:

line number FOR <*variable*>=<*expression*> TO <*expression*> STEP <*expression*>

For example:

```
10 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 goes to the line following the associated NEXT statement.

The variable in the FOR statement must be unsubscripted, although a common use of such loops is to deal with subscripted variables using the FOR 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 expression as already defined in section 2.5.

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

line number 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 value to the variable and checks to see if the variable is still less than the terminal value. When the variable exceeds the terminal value control falls through the loop to the following statements.

If the STEP value 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 variable can be modified within the loop. When control falls through the loop, the variable value 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.

```
1Ø FOR I = 1 TO 1Ø
2Ø PRINT I
3Ø NEXT I
4Ø PRINT I
```

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:

```
1Ø FOR I = 1Ø TO 1 STEP -1
```

the value printed by line 40 would be 1.

```
1Ø FOR I = 2 TO 44 STEP 2
2Ø LET I = 44
3Ø NEXT I
```

The above loop will only be 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 will not be executed at all. A statement of the format:

```
1Ø FOR I = 2Ø TO 2 STEP 2
```

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

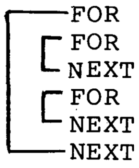
```
1Ø FOR I = 2Ø TO 2 STEP -2
```

For positive STEP values, the loop is executed until the control variable is greater than its final value. For negative STEP values, the loop continues until the control variable is less than its final value.

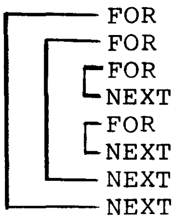
FOR loops can be nested but not overlapped. The depth of nesting depends upon the amount of user storage space available, in other words, upon the size of the user program and the amount of core each user has available. Nesting is a programming technique in which one or more loops are completely within 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. The following diagrams illustrate nesting procedures:

ACCEPTABLE NESTING TECHNIQUES

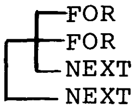
Two Level Nesting



Three Level Nesting



UNACCEPTABLE NESTING TECHNIQUES



An example of a nested FOR-NEXT loop is shown below:

```
 5 DIM X(5,1Ø)
1Ø FOR A=1 TO 5
2Ø FOR B=2 TO 1Ø STEP 2
3Ø LET X(A,B)= A+B
4Ø NEXT B
5Ø NEXT A
55 PRINT X
```

Upon execution of the above statements, BASIC will print 15 when line 55 is processed.

It is possible to exit from a FOR-NEXT loop without the counter 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 the termination and STEP values are assigned.

This implies that unfinished loops maintain parameters in memory, and that excessive premature exiting can exhaust memory.

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

```
          1Ø  FOR I=1 TO 1Ø STEP 5: NEXT I: PRINT "I=";I
```

causes:

```
          I=6
```

to be printed when executed.

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

```
          15  IF I<>J THEN NEXT I
          16  IF I=J THEN FOR I=1 TO J
```

However, FOR modifiers can appear in THEN or ELSE clauses of IF statements. (Modifiers are explained in section 9.8.) For example:

```
     75  IF I=1 THEN PRINT I; FOR I=1 TO 5 ELSE PRINT I; FOR I=1Ø TO 2Ø
```

which causes

```
     1 2 3 4 5
```

to be printed where I=1. The modifier clause applies only to the THEN or ELSE.

3.6.2 Subscripted Variables and the DIM Statement

In addition to the simple variables which were described in Chapter 2, BASIC 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, variables are allowed one or two subscripts.

The name of a subscripted variable is any acceptable BASIC 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 1 to 5 as shown below:

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

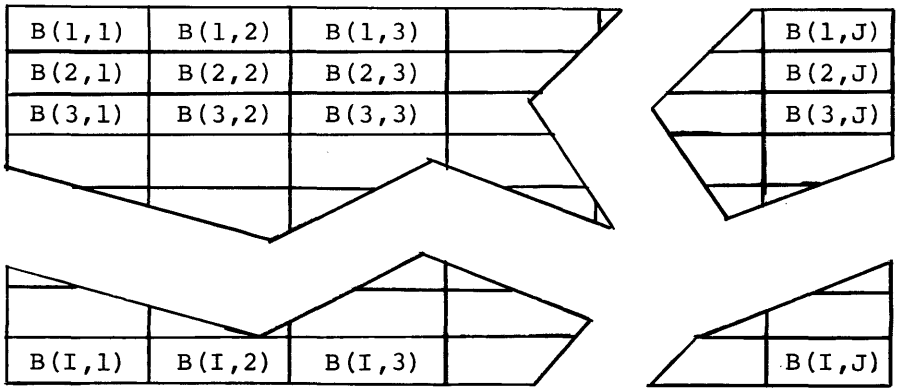
This allows the programmer to reference each of the five elements in the list, and can be considered a one dimensional algebraic matrix as follows:

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

A two dimensional matrix B(I,J) can be defined in a similar manner:

B(1,1), B(1,2), ..., B(1,J),..., B(I,J)

and graphically illustrated as follows:



B(1,1)	B(1,2)	B(1,3)		B(1,J)
B(2,1)	B(2,2)	B(2,3)		B(2,J)
B(3,1)	B(3,2)	B(3,3)		B(3,J)
B(I,1)	B(I,2)	B(I,3)		B(I,J)

Subscripts used with subscripted variables throughout a program can be explicitly stated or be any legal expression.

It is possible to use the same variable name as both a subscripted and as an unsubscripted variable. Both A and A(I) are valid variables and can be used in the same program. However, BASIC will not accept the same variable name as both a singly and a doubly subscripted variable name in the same program.

Use of subscripted variables requires a dimension (DIM) statement 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:

line number DIM *<variable (n)>,<variable(n,m)>,...*

For example:

```
    1Ø DIM X(5), Y(4,2), Z(1Ø, 1Ø)
    12 DIM I4(1ØØ)
```

Only 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 previously in a DIM statement, it is assumed to be of length 10 in each dimension. For this reason and for reasons of good programming practice, all matrices should be correctly dimensioned before their use 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 zero. Dimensioning A(6,10) sets up room for a matrix with 7 rows and 11 columns. This matrix can be thought of as existing in the following form:

$$\begin{array}{lcl}
A_{0,0} & A_{0,1} \cdots & A_{0,10} \\
A_{1,0} & A_{1,1} \cdots & A_{1,10} \\
A_{2,0} & A_{2,1} \cdots & A_{2,10} \\
\vdots & \vdots & \vdots \\
A_{6,0} & A_{6,1} & A_{6,10}
\end{array}$$

as shown in the following program:

```
LISTNH
1Ø REM - MATRIX CHECK PROGRAM
2Ø DIM A(6,1Ø)
3Ø FOR I=Ø TO 6
4Ø LET A(I,Ø) = I
5Ø FOR J=Ø TO 1Ø
6Ø LET A(Ø,J) = J
7Ø PRINT A(I,J);
8Ø NEXT J: PRINT: NEXT I
9Ø END

READY

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

READY
```

Notice that a variable has a value of zero until it is assigned a value.

If the user wishes to conserve core space and not make use of the extra variables set up within the matrix, he should, for example, say DIM A(5,9) which would result in a 6 x 10 matrix which would 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.

An example of subscripted variables follows. I/O is particularly easy when using subscripted variables.

```
LISTNH
5  REM PROGRAM DEMONSTRATING
       READING OF SUBSCRIPTED VARIABLES
1Ø DIM A(5), B(2,3)
15 PRINT "A(I) WHERE I=1 TO 5:"
2Ø FOR I=1 TO 5
25 READ A(I):PRINT A(I);
3Ø NEXT I
35 PRINT
4Ø PRINT
45 PRINT "B(I,J) WHERE I=1 TO 2"
5Ø PRINT "          AND J=1 TO 3:"
55 FOR I=1 TO 2
6Ø PRINT
65 FOR J=1 TO 3
7Ø READ B(I,J):PRINT B(I,J);
75 NEXT J:NEXT I
8Ø DATA 1,2,3,4,5,6,7,8
85 DATA 8,7,6,5,4,3,2,1
9Ø END

READY

RUNNH
A(I) WHERE I=1 TO 5:
1 2 3 4 5

B(I,J) WHERE I=1 TO 2
         AND J=1 TO 3:

6 7 8
8 7 6
READY
```

(The MAT READ statement described in Chapter 8 makes this process much easier.) A DIM statement can be placed anywhere in a multiple statement line. Integer and string variables can also be used in matrices as described in Sections 5.1 and 6.1.

3.7 MATHEMATICAL FUNCTIONS

Within the course of a user's programming experience, he encounters many cases where relatively common mathematical operations are to be 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. The user need never consult tables to obtain the value of the sine of 23° or the natural log of 144. When such values are to be used in an expression, the intrinsic functions, such as

```
SIN(23*PI/18Ø)
LOG(144)
```

are substituted.

The various mathematical functions available in BASIC-Plus are detailed in Table 3.3

Table 3.3

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(Ø)=Ø
INT(X)	returns the greatest integer in X which is less than or equal to X
FIX(X)	returns the truncated value of X, SGN(X)*INT(ABS(X))
COS(X)	returns the cosine of X in radians
SIN(X)	returns the sine of X in radians
TAN(X)	returns the tangent of X in radians
ATN ~~ATAN~~(X)	returns the arctangent of X in radians
SQR(X)	returns the square root of X
EXP(X)	returns the value of e↑X, where e=2.1418
LOG(X)	returns the natural logarithm of X, $\log_e X$
LOG1Ø(X)	returns the common logarithm of X, $\log_{10} X$
PI	has a constant value of 3.1415927
RND(X)	returns a random number between 0 and 1 the same sequence of random numbers is generated each time a program is run requiring the use of the random number generator

Most of these functions are self-explanatory. Those which are more complex are explained in the following section.

3.7.1 Examples of Particular Intrinsic Functions

Sign Function, SGN(X)

The sign function returns the value 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
1Ø REM- SGN FUNCTION EXAMPLE
2Ø READ A,B
25 PRINT "A="A,"B="B
3Ø PRINT "SGN(A)="SGN(A),"SGN(B)="SGN(B)
4Ø PRINT "SGN(INT(A))="SGN(INT(A))
5Ø DATA -7.32, .44
6Ø END

READY

RUNNH

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


READY
```

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*1Ø↑D+.5)/1Ø↑D
```

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

```
LISTNH
1Ø REM- INT FUNCTION EXAMPLE
2Ø PRINT "NUMBER TO BE ROUNDED";
3Ø INPUT A
4Ø PRINT "NO. OF DECIMAL PLACES";
5Ø INPUT D
6Ø LET B=INT(A*10↑D+.5)/1Ø↑D
7Ø PRINT "A ROUNDED ="B
8Ø GO TO 2Ø
9Ø END

READY

RUNNH

NUMBER TO BE ROUNDED?  55.65342
NO. OF DECIMAL PLACES? 2
A ROUNDED = 55.65
NUMBER TO BE ROUNDED? 78.375
NO. OF DECIMAL PLACES? -2
A ROUNDED = 1ØØ
NUMBER TO BE ROUNDED? 67.89
NO. OF DECIMAL PLACES? -1
A ROUNDED = 7Ø
NUMBER TO BE ROUNDED? ↑C


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.

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, as that value is ignored and serves no function.

```
LISTNH
1Ø REM - RANDOM NUMBER EXAMPLE
25 PRINT "RANDOM NUMBERS"
3Ø FOR I=1 TO 3Ø
4Ø PRINT RND(Ø),
5Ø NEXT I
6Ø END

READY

RUNNH


RANDOM NUMBERS
 .2435Ø41     .2998482     .6Ø75527     .946681      .2121133
 .75255Ø9     .6Ø62854     .8647548     .7319596     .6Ø89648
 .Ø6615233    .916230Ø4    .9Ø2Ø116     .1659956     .8778697
 .7732576     .7387181     .47299Ø2     .1894784     .8799586
 .5744461     .527Ø493     .99228Ø8     .21Ø24Ø9     .33Ø9181
 .Ø9334Ø31    .5817788     .65Ø6Ø97     .6676494     .15Ø4Ø93

READY
```

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

```
4Ø PRINT INT(1Ø*RND(Ø)),
```

and tell BASIC to run the program again. This time the results will look as follows:

```
RANDOM NUMBERS
 2            2            6            9            2
 7            6            8            7            6
 Ø            9            9            1            8
 7            7            4            1            8
 5            5            9            2            3
 Ø            5            6            6            1

READY
```

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

```
(B-A)*RND(Ø)+A
```

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

3.7.2 RANDOMIZE Statement

If the random number generator is to calculate different random numbers every time the program is run, the RANDOMIZE statement is used. RANDOMIZE is normally placed at the beginning of a program which uses random numbers (the RND function). When executed, RANDOMIZE causes the RND function to choose a random starting value, so that the program run twice will give different results.

For example:

```
1Ø RANDOMIZE
2Ø PRINT RND(Ø)
3Ø END
```

will print a different number each time it is run. For this reason, it is a good practice to debug a program completely before inserting the RANDOMIZE statement.

The form of the statement is as follows:

line number RANDOMIZE

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

```
LISTNH
15 RANDOMIZE
2Ø FOR I=1 TO 5
25 PRINT "VALUE" I " IS" RND(Ø)
3Ø NEXT I
35 END

READY

RUNNH
```

```
VALUE 1 IS .7ØØ4438
VALUE 2 IS .67Ø6673
VALUE 3 IS .72ØØØ98
VALUE 4 IS .284Ø528
VALUE 5 IS .2242288

READY

RUNNH

VALUE 1 IS .59Ø55
VALUE 2 IS .34Ø9859
VALUE 3 IS .73Ø9656
VALUE 4 IS .31692Ø3
VALUE 5 IS .3228311

READY
```

Clearly, the output from each run is different.

3.7.3 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 allows the programmer to define his own functions and call these functions in the same way he would call the square root or trig functions.

These user defined functions consist of a function name, the first two letters of which are FN followed by any valid variable name. (The type of variable name used determines the type of the function.) For example:

FNA	floating point function
FNA1	floating point function
FNA%	integer function
FNA$	string function

We define the function once ~~at the beginning of the program before its first use.~~ The defining or DEF statement is formed as follows: ?

line number DEF FNα(*arguments*) = <*expression* (*arguments*)>

where α is any acceptable (one, two, or three character) variable name. The arguments must be the same on each side of the equal sign and may consist of one or more dummy variables. For example:

```
        1Ø DEF FNA(S) = S↑2
```

will cause a later statement:

```
        2Ø LET R = FNA(4)+1
```

to be evaluated as R=17.

The two following programs

Program #1:

```
        LISTNH
        1Ø DEF FNS(A) = A↑A
        2Ø FOR I=1 TO 5
        3Ø PRINT I, FNS(I)
        4Ø NEXT I
        5Ø END
```

Program #2:

```
        LISTNH
        1Ø DEF FNS(X) = X↑X
        2Ø FOR I=1 TO 5
        3Ø PRINT I, FNS(I)
        4Ø NEXT I
        5Ø END
```

cause the same output:

```
        RUNNH

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

The arguments in the DEF statement can be seen to have no significance; they are strictly dummy variables. The function itself can be defined in the DEF statement in terms of numbers, variables, other functions, or mathematical expressions. For example:

```
        1Ø  DEF FNA(X) = X↑2+3*X+4
        15  DEF FNB(X) = FNA(X)/2 + FNA(X)
        2Ø  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:

```
4Ø LET R = FNA(X+Y+Z)
```

The user defined function can be a function of from zero to five ~~eight~~ 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 was defined.

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

```
1Ø DEF FNZ(X) = X↑2
2Ø LET A = 2
3Ø PRINT FNZ(2+A)
 .
 .
 .
```

line 3Ø will cause 16 to be printed, when executed.

The function can be called recursively, meaning that an argument of the function can contain the function itself, as follows:

```
1Ø DEF FNZ(X) = X↑2
2Ø LET A= 2
3Ø PRINT FNZ(2+A*FNZ(2))
```

line 3Ø will cause 1ØØ to be printed when executed. FNZ(2)=4, which is then multiplied by A to give 8, plus 2 equals 10 as the argument. FNZ(10) = 100.

```
LISTNH
1  ! MODULUS ARITHMETIC PROGRAM
5  ! FIND X MOD M
1Ø DEF FNM(X,M) = X-M*INT(X/M)
15  !
2Ø ! FIND A+B MOD M
25 DEF FNA(A,B,M) = FNM(A+B,M)
3Ø !
35 ! FIND A*B MOD M
40 DEF FNB(A,B,M) = FNM(A*B,M)
41 !
45 PRINT
5Ø PRINT "ADDITION AND MULTIPLICATION TABLES, MOD M"
55 INPUT "GIVE ME AN M";M
6Ø PRINT: PRINT "ADDITION TABLES MOD "M
65 GOSUB 8ØØ
7Ø FOR I=Ø TO M-1
75 PRINT I;"  ";
8Ø FOR J=Ø TO M-1
85 PRINT FNA(I,J,M);
9Ø NEXT J: PRINT: NEXT I
1ØØ PRINT: PRINT
11Ø PRINT "MULTIPLICATION TABLES MOD " M
12Ø GOSUB 8ØØ
13Ø FOR I=Ø TO M-1
14Ø PRINT I;"  ";
15Ø FOR J=Ø TO M-1
16Ø PRINT FNB(I,J,M);
17Ø NEXT J: PRINT: NEXT I
18Ø STOP
8ØØ !SUBROUTINE FOLLOWS:
81Ø PRINT: PRINT TAB(4);Ø;
82Ø FOR I=1 TO M-1
83Ø PRINT I;: NEXT I: PRINT
84Ø FOR I=1 TO 2*M+3
85Ø PRINT "-";: NEXT I: PRINT
86Ø RETURN
87Ø END

READY
```

```
RUNNH

ADDITION AND MULTIPLICATION TABLES, MOD M
GIVE ME AN M? 7

ADDITION TABLES MOD 7

    Ø 1 2 3 4 5 6
-----------------
Ø   Ø 1 2 3 4 5 6
1   1 2 3 4 5 6 Ø
2   2 3 4 5 6 Ø 1
3   3 4 5 6 Ø 1 2
4   4 5 6 Ø 1 2 3
5   5 6 Ø 1 2 3 4
6   6 Ø 1 2 3 4 5


MULTIPLICATION TABLES MOD 7

    Ø 1 2 3 4 5 6
-----------------
Ø   Ø Ø Ø Ø Ø Ø Ø
1   Ø 1 2 3 4 5 6
2   Ø 2 4 5 1 3 5
3   Ø 3 6 2 5 1 4
4   Ø 4 1 5 2 6 3
5   Ø 5 3 1 6 4 2
6   Ø 6 5 4 3 2 1
STOP AT LINE 18Ø

READY
```

Figure 3-1   Modulus Arithmetic

If the same function name is defined more than once, the first definition is used and subsequent definitions are treated as errors and ignored.

```
1Ø DEF FNX(X) = X↑2
2Ø DEF FNX(X) = X+X
ILLEGAL FN REDEFINITION
3Ø PRINT FNX(6)
4Ø END
RUNNH
36
```

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

```
1Ø DEF FNA(X) = 4 + 2
2Ø LET R = FNA(1Ø)+1
3Ø PRINT R
4Ø END
RUNNH
7
```

The program in Figure 3-1 contains examples of a multi-variable DEF statement in lines 10, 25, and 40.

Integer and character string functions can also be created (in which case the variable name has two or three characters: FNA$(X),FNB1%(N)). See sections 5.5 and 6.5 for details.

3.8 SUBROUTINES

When particular mathematical expressions are 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, too, is possible. A subroutine is a section of code performing some unusually lengthy 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.

3.8.1 GOSUB Statement

Subroutines are usually 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:

line number GOSUB <*line number*>

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

```
        5Ø GOSUB 2ØØ
```

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:

line number RETURN

which causes control to return to the line following the GOSUB statement.

Before transferring to the subroutine, BASIC internally records the next line number to be processed after the GOSUB statement; the RETURN statement is a signal to transfer control to this line. In this way, no matter how many subroutines or how many times they are called, BASIC always knows where to go next. Figure 3-1 and the following program demonstrate simple subroutines:

```
LISTNH
1 REM - THIS PROGRAM ILLUSTRATES GOSUB AND RETURN
1Ø DEF FNA(X)= ABS(INT(X))
2Ø INPUT A,B,C
3Ø GOSUB 1ØØ
4Ø LET A=FNA(A)
5Ø LET B=FNA(B)
6Ø LET C=FNA(B)
7Ø PRINT
8Ø GOSUB 1ØØ
9Ø STOP
1ØØ REM - THIS SUBROUTINE PRINTS OUT THE SOLUTIONS
11Ø REM - OF THE EQUATION:  AX↑2 + BX + C = Ø
12Ø PRINT "THE EQUATION IS    " A "*X↑2  + " B "*X  + " C
13Ø LET D= B*B - 4*A*C
14Ø IF D<>Ø THEN 17Ø
150 PRINT "ONLY ONE SOLUTION... X =" -B/(2*A)
16Ø RETURN
17Ø IF D<Ø THEN 2ØØ
18Ø PRINT "TWO SOLUTIONS...X =";
185 PRINT (-B+SQR(D))/(2*A) "AND X =" (-B-SQR(D))/(2*A)
19Ø RETURN
2ØØ PRINT "IMAGINARY SOLUTIONS... X = (";
2Ø5 PRINT -B/(2*A) "," SQR(-D)/(2*A) ")  AND  (";
2Ø7 PRINT -B/(2*A) "," -SQR(-D)/(2*A) ")"
21Ø RETURN
9ØØ END

READY

RUNNH

? 1,.5,-.5
THE EQUATION IS    1 *X↑2  +  .5 *X  + -.5
TWO SOLUTIONS... X = .5 AND X = -1

THE EQUATION IS    1 *X↑2  +  Ø *X  +  1
IMAGINARY SOLUTIONS...X = ( Ø , 1 )  AND  ( Ø ,-1 )

READY
```

Lines 100 through 210 constitute the subroutine. The subroutine is executed from line 30 and again from line 80. When control returns to line 90 the program encounters the STOP statement and terminates execution. Note that even though the program logically ends with a STOP, the END statement is still present.

3.8.3 Nesting Subroutines

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.

Subroutines can also be nested, in terms of one subroutine calling another subroutine. If the execution of a subroutine encounters a RETURN statement, it will return control to the line following the GOSUB which called that subroutine; therefore, a subroutine can call another subroutine, even itself. Subroutines can be entered at any point and have more than one RETURN statement where certain conditions will cause control to reach only one RETURN statement. It is possible to transfer to the beginning or 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 core storage available at the particular installation. Exceeding this limit results in the message:

```
MAXIMUM CORE SIZE EXCEEDED AT LINE XXX
```

where XXX is the line number of the line containing the error.

3.9 READ, DATA, AND RESTORE STATEMENTS

READ and DATA statements are used to enter information into the user program during execution. A READ statement is used to assign to the listed variables those values which are obtained from a DATA statement. Neither statement is used without the other.

A READ statement is of the form:

line number READ <*list*>

A DATA statement is of the form:

line number DATA <*list*>

A READ statement causes the variables listed in it to be assigned, in order, the next available numbers in the collection of DATA statements. Before the program is run, BASIC takes all DATA statements in the order they appear and creates a data block out of the numbers. Each time a READ statement is encountered in the program, the data block supplies the next available number or numbers. 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.

READ and DATA statements appear as follows:

```
150 READ X,Y,Z,X1,Y2,Q9
330 DATA 4,2,1.7
340 DATA 6.734E-3, -174.321, 3.1415927
```

Note that only numbers are used in this particular DATA statement (string data is treated in section 5.3).

Since data must be read before it can be used in a program, READ statements normally occur near the beginning of a program. The location of DATA statements is arbitrary, as long as they occur in the correct order. A good practice is to collect all DATA statements near the end of the program. A DATA statement must be the only statement on a line, while a READ statement can be placed anywhere in a multiple statement line.

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

line number RESTORE

For example:

```
85 RESTORE
```

will cause the next READ statement following line 85 to begin reading data from the first DATA statement in the program, regardless of where the last data value was found.

You may use the same variable names the second time through the data or not, as you choose, 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 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 200.

```
LISTNH
1Ø REM PROGRAM TO ILLUSTRATE USE OF RESTORE
15 READ N: PRINT "VALUES OF X ARE:"
2Ø FOR I=1 TO N: READ X: PRINT X,
25 NEXT I
3Ø RESTORE
35 PRINT: PRINT "SECOND LIST OF X VALUES"
4Ø PRINT "FOLLOWING RESTORE STATEMENT:"
45 FOR I=1 TO N: READ X: PRINT X,
5Ø NEXT I
6Ø DATA 4,1,2
7Ø DATA 3,4
8Ø END

READY

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

When reading a BASIC program from the terminal tape reader, often the last line read is the READY printed by BASIC when the program was listed (and punched). BASIC interprets this as a READ Y command and, if there are no DATA statements in the program, gives an "OUT OF DATA AT LINE Ø" message.

3.10 INPUT STATEMENT

The second way to input 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. (Non-terminal INPUT is described in Chapter 7.) Depending upon how many values are to be accepted by the INPUT command, the programmer may wish to write himself a note reminding himself what data is to be typed at what time (this can be done with the PRINT or INPUT statement). In the example program following, the questions are asked at execution time: INTEREST IN PERCENT?, AMOUNT OF LOAN?, and NUMBER OF YEARS? The programmer knows which value is requested and proceeds to type and enter the appropriate value.

```
LISTNH
1Ø REM PROGRAM TO COMPUTE INTEREST PAYMENTS
15 INPUT "INTEREST IN PERCENT"; J
2Ø LET J=J/1ØØ
25 INPUT "AMOUNT OF LOAN"; A
3Ø INPUT "NUMBER OF YEARS"; N
35 INPUT "NO. OF PAYMENTS PER YEAR"; M
4Ø N=N*M: I=J/M: B=1+I: R=A*I/(1-1/B↑N)
45 PRINT: PRINT "AMOUNT PER PAYMENT =";R
5Ø PRINT "TOTAL INTEREST      =";R*N-A
55 PRINT: B=A
6Ø PRINT "INTEREST    APP TO PRIN    BALANCE OF PRIN"
65 L=B*I: P=R-L: B=B-P
67 PRINT L,P,B
7Ø IF B>=R GOTO 65
75 PRINT B*I,R-B*I
8Ø PRINT "LAST PAYMENT WAS "B*I+B
85 END

READY

RUNNH
INTEREST IN PERCENT? 9
AMOUNT OF LOAN? 25ØØ
NUMBER OF YEARS? 2
NO. OF PAYMENTS PER YEAR? 4

AMOUNT PER PAYMENT =344.96154
TOTAL INTEREST     =259.69234

INTEREST    APP TO PRIN    BALANCE OF PRIN
56.25         288.71154     2211.2885
49.75399      295.2Ø755     1916.Ø8Ø9
43.11182      3Ø1.84972     1614.2312
36.32Ø2Ø2     3Ø8.64134     13Ø5.5898
29.375772     315.58577     990.ØØ4Ø7
22.275Ø92     322.68645     667.31763
15.Ø14647     329.9469      337.37Ø73
7.59Ø8414     337.37Ø7
LAST PAYMENT WAS 344.96157

READY
```

The INPUT statement is of the form:

line number INPUT <*list*>

For example:

```
1Ø INPUT A,B,C
```

causes the computer to pause during execution, print a question mark, and wait for the user to type three numerical values separated by commas. The values typed are entered to the computer by typing the RETURN key.

As in the program above, the question mark is grammatically useful if a printed question is to prompt the typing of the input values.

The output for the program begins after the word RUN 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 have been typed in answer to the printed ? character, type the RETURN key to enter the values to the computer. If too few values are listed, 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 double quotes from the other arguments of the INPUT statement. For example:

```
10 INPUT "YOUR AGE IS"; A
```

is equivalent to

```
10 PRINT "YOUR AGE IS";
20 INPUT A
```

(For a description of the PRINT statement and the use of the semicolon and comma characters, see section 3.11.)

3.11 PRINT STATEMENT

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

line number PRINT $\{$*list*$\}$

where the list can contain expressions, text strings, or both. As the ~~square brackets~~ braces indicate, the list is optional. Used alone, the

PRINT statement:

```
        25 PRINT
```

causes a blank line to be printed on the teleprinter (a carriage return/line feed operation is performed).

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

```
        1Ø LET A=1: LET B=2: LET C=3+A
        2Ø PRINT
        3Ø PRINT A+B+C
```

When this program is executed, a carriage return/line feed is performed at the terminal (line 20) and the number 7 is printed.

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

```
        1Ø A=1: PRINT A: A=A+5: PRINT: PRINT A
```

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

```
        1

        6
```

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.

3.11.1 Formatting Printout into Columns

BASIC considers the terminal printer to be divided into five zones of 14 spaces each. When an item in a PRINT statement is followed by a comma, the next value to be printed will appear in the next available print zone. For example:

```
        1Ø LET A=3: LET B=2
        2Ø PRINT A,B,A+B,A*B,A-B,B-A
```

When the preceding lines are executed, the following is printed:

```
3              2              5              6              1
-1
```

Notice that the sixth element in the PRINT list is printed as the first entry on a new line.

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

```
         1Ø LET A=1: LET B=2
         2Ø PRINT A,B,,A+B
```

will cause the following to be printed:

```
1              2                             3
```

If the last item in a PRINT statement is followed by a comma, the next value to be printed by a later PRINT statement will appear in the next available print zone. For example:

```
         1Ø A=1:B=2:C=3
         2Ø PRINT A,:PRINT B: PRINT C
         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 the next value to be printed ~~one~~ Two spaces to the right of the preceding printout. A semicolon following the last item in the list causes the next printed value to appear ~~one~~ two spaces to the right of the preceding value on the same line (provided there is room left on the line for printing). The following example shows the various uses of the semicolon and comma.

```
LISTNH
1Ø LET A=1: B=2: C=3
2Ø PRINT A;B;C;
3Ø PRINT A+1;B+1;C+1
4Ø PRINT A,B,C

READY

RUNNH
1 2 3 2 3 4
1              2              3

READY
```

3.11.2 Character Strings in PRINT Statements

The PRINT statement can be used to print a message, either alone or together with the evaluation and printing of numeric values. Characters are indicated for printing by enclosing them in single or double quotation marks (therefore each type of quotation mark can only be printed if surrounded by the other type of quotation mark). For example:

```
LISTNH
1Ø PRINT "THIS IS A TEXT STRING"
2Ø PRINT '"NEVERMORE"'

READY

RUNNH
THIS IS A TEXT STRING
"NEVERMORE"

READY
```

As another example, consider the following line:

```
4Ø PRINT "AVERAGE GRADE IS ";X
```

which prints the following (where X is equal to 83.4):

```
AVERAGE GRADE IS  83.4
```

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; the spaces will appear in the printout exactly as they are 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 ignored. Thus, the previous example could be expressed as:

```
         4Ø PRINT "AVERAGE GRADE IS " X
```

and the same printout would result. A colon or semicolon appearing as the last item of a PRINT list will always suppress the carriage return/line feed operation.

The following example demonstrates the use of the formatting characters (, and ;) with text strings.

```
         12Ø PRINT "STUDENT NUMBER "X,"GRADE = "G;"AVE. = "A;
         13Ø PRINT "NO. IN CLASS = "N
```

could cause the following to be printed (assuming calculations were done prior to line 130):

```
STUDENT NUMBER 11556    GRADE = 76    AVE. = 85.4    NO. IN CLASS = 27
```

The above line is exactly 72 characters long (the length of the teleprinter line). The user is advised to limit the length of his lines for output to 72 characters to avoid their being partially printed on the next line.

If a PRINT statement contains a message to be printed beginning with a quotation mark and not limited by a closing quote, the entire statement is printed as though the quote were the last character. For example:

```
         1Ø PRINT "A =;X
```

causes

```
         A =;X
```

to be printed, while (if X=3):

```
         1Ø PRINT "A =";X
```

causes

```
        A = 3
```

to be printed.

3.12 STOP AND END STATEMENTS

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

line number END

The line number of the END statement should be the largest line number in the program. Any lines having line numbers greater than that of the END statement are not executed.

NOTE

> A program will execute without an END statement; however, an error message is printed if a program is recalled having been saved without an END statement.

The STOP statement is of the form:

line number STOP

and is the equivalent to a GOTO n, where n is the line number of the END statement.

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 completed.

CHAPTER 4

IMMEDIATE MODE OPERATIONS

4.1 USE OF IMMEDIATE MODE FOR STATEMENT EXECUTION

It is not always necessary to write a BASIC program to use the RSTS-11 system. Most of the statements discussed in this manual can either be included in a program for later execution or be given as commands, which are immediately executed by BASIC. This latter facility permits the RSTS-11 user to have an extremely powerful desk calculator available whenever he is on-line.

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 executed immediately upon being entered to the system. Thus the line:

```
10 PRINT "THIS IS A PDP-11"
```

will produce no action at the console upon entry, while the statement:

```
PRINT "THIS IS A PDP-11"
THIS IS A PDP-11

READY
```

when entered causes the immediate output shown above. The READY message is then printed to indicate the system readiness for another direct command.

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 a complete program being written.

In order to facilitate debugging a user program, the user can place STOP statements liberally throughout the program. Each STOP statement will cause the program to halt, at which time the user can examine various data values, perhaps change them in Immediate Mode, and then give the

```
CONT
```

command to continue his program execution.

4.3 ONE STATEMENT PER LINE

When using Immediate Mode, nearly all the standard verbs can be used to generate or print results. In Immediate Mode, however, there can only be one BASIC statement per line. If a ":" is included in an Immediate Mode statement, the instructions to its right are ignored. For example:

```
A=1: PRINT A
```

will cause the variable A to be assigned the value 1, but nothing is printed at the console. This operation could be successfully performed as follows:

```
A=1

READY
PRINT A
1

READY
```

Since multiple statement lines are ignored in Immediate Mode, explicit FOR-NEXT loops are not possible. The use of the FOR modifier (and all other modifiers) is allowed. Thus a table of square roots can be produced as follows:

```
PRINT I, SQR(I) FOR I=1 TO 1Ø
1             1
2             1.4142136
3             1.7320508
4             2
5             2.236068
6             2.4494897
7             2.6457513
8             2.86284271
9             3
10            3.1622777

READY
```

4.4 RESTRICTIONS ON IMMEDIATE MODE

Some statements, particularly those that would cause execution of lines within a user's stored program, are not allowed in Immediate Mode. These statements include:

```
GOTO
GOSUB
References to user defined functions
```

Thus the following dialog might result if the user defined a function in his program and tried to reference it in Immediate Mode.

```
1Ø DEF FNA(X) = X↑2 + 2*X   !SAVED STATEMENT
PRINT FNA(1)                !IMMEDIATE MODE
PLEASE USE THE RUN COMMAND

READY
```

Certain commands make no logical sense when used in Immediate Mode. When these are given, the message ILLEGAL IN IMMEDIATE MODE is given:

```
DEF FNA(X) = 1
ILLEGAL IN IMMEDIATE MODE

READY
```

Commands in this category include:

```
DEF
DIM
DATA
FOR
RETURN
NEXT
FNEND
```

PART II

BASIC-PLUS ADVANCED FEATURES

This part of the manual describes the special features of BASIC-Plus which make the language a superior tool for all manner of data manipulation. Additional capabilities of the statements previously described are included, along with new statements, character string manipulating facilities, integer mode variables and arithmetic, intrinsic matrix functions, disk and DECtape storage of data files, and a complete description of all BASIC Input/Output facilities.

In general, the new techniques presented here allow the user to write programs which conserve core space and enhance execution time. With the ability to manipulate character strings, the user can write sophisticated programs requiring the ability to handle a wide range of data. Formatting of output is also described to enable the user to obtain maximum effective output.

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

CHAPTER 5

CHARACTER STRINGS

5.1 CHARACTER STRINGS

The previous chapters describe the manipulation of numerical information only; however, BASIC 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 alphabetic, numeric, or alphanumeric characters. (An alphanumeric string is one which contains letters, numbers, spaces, or any combination of characters.)

Without realizing it, the reader has already encountered character strings. Consider the following program which prints the name of a month, given its number (the ON GOTO statement is described in section 9.2):

```
LISTNH
1Ø INPUT "NUMBER OF MONTH IS";N
15 IF N<1 OR N>12 THEN STOP
2Ø ON N GOTO 3Ø,35,4Ø,45,5Ø,55,6Ø,65,7Ø,75,8Ø,85
3Ø PRINT "JANUARY": GOTO 9Ø
35 PRINT "FEBRUARY": GOTO 9Ø
4Ø PRINT "MARCH": GOTO 9Ø
45 PRINT "APRIL": GOTO 9Ø
5Ø PRINT "MAY": GOTO 9Ø
55 PRINT "JUNE": GOTO 9Ø
6Ø PRINT "JULY": GOTO 9Ø
65 PRINT "AUGUST": GOTO 9Ø
7Ø PRINT "SEPTEMBER": GOTO 9Ø
75 PRINT "OCTOBER": GOTO 9Ø
8Ø PRINT "NOVEMBER": GOTO 9Ø
85 PRINT "DECEMBER": GOTO 9Ø
9Ø END

READY

RUNNH
NUMBER OF MONTH IS? 7
JULY

READY
```

In Chapter 3 we saw how the INPUT and PRINT statements could be used to print a verbal message along with the input and output of numeric values. These messages are actually character string constants (much like the number 4 is a numeric constant). In a similar way, there are character string variables and functions.

5.1.1 Character String Names

Variable names can be introduced for simple strings and for both lists and matrices composed of strings (which is to say one and two dimensional string matrices). Any variable name followed by a dollar sign ($) character indicates a string variable. For example:

```
A$
C7$
```

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)
C$(m,n)            G1$(m,n)
```

where m and n indicate the position of that element of the matrix within the whole.

The same name can be used as a numeric variable and as a string variable in the same program with the restriction that a one and a two dimensional matrix cannot have the same name in the same program. For example:

```
A                  A(n)
A$                 A$(m,n)
```

can all be used in the same program, but

```
    A(n) and A(m,n)
```

or

```
    A$(n) and A$(m,n)
```

cannot.

5.1.2 Characteristics of Strings

A character string can contain any number of characters (from Ø to n). However, the LINE FEED key cannot be used to type a string on two or more terminal lines. Since core storage is limited, strings can also be saved in files on the system disk (see section 7.1).

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

```
10 DIM S1$(5)
```

indicates that there are six strings, S1$(Ø) through S1$(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). 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:

```
1Ø FOR I = 1 TO 7
2Ø LET B$(I) = "PDP-11"
3Ø NEXT I
```

they would cause a list B$(n) to be created having 11 accessible elements, B$(Ø) through B$(1Ø). The elements B$(1) through B$(7) are set equal to "PDP-11" and the others would be null strings (have no characters). As a general rule, all lists and matrices should be dimensioned to the maximum size being referenced in the program.

Just as numbers can be used as constants as well as being referenced by variable names, BASIC-Plus allows for character string constants. In line 20 above, "PDP-11" is a character string constant. Character string constants are delimited by either single or double quotes. For example:

```
1Ø5 LET Y$ = "FILE4"
33  B1$ = 'CAN'
8Ø  IF A$ = "YES" GOTO 25Ø
```

Character string constants can be used in any statement where their use has a logical meaning.

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

```
55 IF A$(I)<A$(I+1) GOTO 1ØØ
```

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, trailing blanks are ignored. That is to say "YES" is equivalent to "YES ".

TABLE 5-1

Relational Operators Used With String Variables

Operator	Example	Meaning
=	A$ = B$	The strings A$ and B$ are equivalent.
<	A$ < B$	The string A$ occurs before B$ in alphabetical sequence.
<=	A$ <= B$	The string A$ is equivalent to or occurs before B$ in alphabetical sequence.
>	A$ > B$	The string A$ occurs after B$ in alphabetical sequence.
>=	A$ >= B$	The string A$ is equivalent to or occurs after B$ in alphabetical sequence.
<>	A$ <> B$	The strings A$ and B$ are not equivalent.

5.2 INDIVIDUAL CHARACTERS WITHIN STRINGS, 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 (the entirety of) 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. Table 5-2 describes the relationship between the ASCII characters and their numerical values.

As an illustration, consider the following:

```
LISTNH
1Ø DIM X(3)
15 LET A$ = "CAT"
2Ø CHANGE A$ TO X
25 PRINT X(Ø);X(1);X(2);X(3)
3Ø END

READY

RUNNH
3 67 65 84

READY
```

TABLE 5-2

ASCII CHARACTER CODES

Character	ASCII Code No. (Decimal)	Character	ASCII Code No. (Decimal)
Space	32	@	64
!	33	A	65
"	34	B	66
#	35	C	67
$	36	D	68
%	37	E	69
&	38	F	70
'	39	G	71
(	40	H	72
)	41	I	73
*	42	J	74
+	43	K	75
,	44	L	76
-	45	M	77
.	46	N	78
/	47	O	79
0	48	P	80
1	49	Q	81
2	50	R	82
3	51	S	83
4	52	T	84
5	53	U	85
6	54	V	86
7	55	W	87
8	56	X	88
9	57	Y	89
:	58	Z	90
;	59	[	91
<	60	\	92
=	61	]	93
>	62	↑	94
?	63	←	95

Additional symbols useful on output are as follows:

LF (line feed)	10
CR (carriage return)	13

The above list is not complete; there are 128 characters numbered 0 through 127.

X(1) through X(3) take on the ASCII values of the characters in the string variable A$. The first element of X, X(Ø) becomes the number of characters present in A$. 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 string which have been successfully transformed into numeric values, and will be less than or equal to the dimension of the list.

Another program performing this operation is shown below:

```
LISTNH
1Ø DIM A(65)
15 READ A$
2Ø CHANGE A$ TO A
25 FOR I=Ø TO A(Ø)
3Ø PRINT A(I);:NEXT I
35 DATA ABCDEFGHIJKLMNOPQRSTUVWXYZ
4Ø END

READY

RUNNH
26 65 66 67 68 69 7Ø 71 72 73 74 75 76 77 78 79 8Ø 81 82 83 84 85 86 87
88 89 9Ø
READY
```

Notice that A(Ø) = 26.

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

```
LISTNH
1Ø FOR I=Ø TO 5
15 READ A(I)
2Ø NEXT I
25 DATA 5,65,66,67,68,69
3Ø CHANGE A TO A$
35 PRINT A$
4Ø 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. In line 15 of the previous program, A(Ø) is read as 5. The following is another example:

```
LISTNH
1Ø DIM V(128)
15 INPUT "HOW MANY CHARACTERS";V(Ø)
2Ø FOR I=1 TO V(Ø)
25 INPUT V(I)
3Ø NEXT I
35 CHANGE V TO A$
4Ø PRINT A$: END
5Ø END
READY

RUNNH
HOW MANY CHARACTERS? 3
? 67
? 64
? 87
C@W

READY
```

Numbers which have no character equivalent in Table 5-2 do not cause a character to be printed.

5.3 STRING INPUT

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

```
1Ø READ A$, B, C, D
2Ø 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 an error message to be printed
```

Quotation marks are necessary around string items in DATA statements only if the string contains a comma or if embedded blanks within the string are significant. Quotes (single or double) are always acceptable around string items, even though not always necessary. For example, the items in line 40 in the following program are all acceptable character strings and would be read as printed.

```
LISTNH
1Ø READ A$,B$,C$,D$,E$
2Ø PRINT A$;B$;C$;D$;E$
3Ø PRINT A$,B$,C$,D$,E$
4Ø DATA "MR. JONES",MISS SMITH, "MRS. BROWN", "MISS", '"MR"'

READY

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

READY
```

A READ statement can appear anywhere in a multiple statement line, but a DATA statement must be the only statement on a line. See also the section on MAT READ which reads matrices (either numeric or string) section 8.1.

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

```
1Ø INPUT "YOUR NAME";N$,"AGE";A
```

is equivalent to

```
1Ø PRINT "YOUR NAME";
15 INPUT N$
2Ø PRINT "YOUR AGE";
25 INPUT A
```

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

line number INPUT LINE <*string variable*>

For example:

```
1Ø INPUT LINE A$
```

which causes the program to accept a line of input from the terminal up to 72 characters long (the length of the terminal line) containing 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.

When inputting or reading strings, leading blanks are ignored unless enclosed in single or double quotes.

5.4 STRING OUTPUT

When character string constants are included in PRINT statements, only those characters within quotes are printed. No leading or trailing spaces are added. For example:

```
LISTNH
1Ø X=1.Ø:Y=2.Ø1
2Ø PRINT "A = ";X " B = "Y
3Ø PRINT "DONE"
4Ø END

READY

RUNNH
A = 1  B = 2.Ø1
DONE

READY
```

Semicolons separating character string constants from other list items are optional. In line 10 above the variable Y is not separated from the character string " B = ".

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

5.5 STRING FUNCTIONS

Like the intrinsic mathematical functions found in BASIC, BASIC-Plus contains various functions for use with character strings. These functions allow the program to access part of a string, determine the length attribute of a string, generate a numeric string or value given a numeric value or string, search for a substring within a larger string, and other useful operations. (These functions are particularly useful when dealing with whole lines of alphanumeric information input with an INPUT LINE statement.) The various functions are summarized in Table 5.3.

5.5.1 User Defined String Functions

Functions can be written (see section 3.7.3 and 9.1) containing string variables in the same way as they are created for numeric variables. The name of the function is indicated as being a string function by the $ character after the function name.

TABLE 5.3

String Functions

Function Code	Meaning
LEFT(A$,N)	Indicates a substring of the string A$ from the first character to the Nth character (the leftmost N characters of the string A$).
RIGHT(A$,N) : 2 PARAM. "SUBSTR ()"	Indicates a substring of the string A$ from the Nth character to the last character in A$ (the rightmost characters of the string A$ starting with the Nth character).
MID(A$,N1,N2) = 3 PARAM. "SUBSTR()"	Indicates a substring of the string A$ starting with character N1 and N2 characters long (the characters between and including the N1 to N1+N2-1 characters of the string A$).
LEN(A$)	Indicates the number of characters in the string A$ (including trailing blanks).
+	Indicates a concatenation operation. For example: "ABC" + "DEF" is equivalent to "ABCDEF" "12" + "34" + "56" is equivalent to "123456".
CHR$(N)	Generates a character string having the ASCII value of N (see Table 5.2). For example: CHR$(40) is equivalent to "(". Only one character can be generated.
ASCII(A$)	Generates the ASCII value of the first character in A$. For example, ASCII("X") is equivalent to 88, the ASCII value of X.
DATE$(Ø)	Indicates the current date in the form: 12-JUN-71 This quantity can be printed on output by simple reference to the function.
INSTR(N1,A$,B$)	Indicates a search for the substring B$ within the string A$ beginning at character position N1. Returns a value of Ø 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).
SPACE$(N)	Indicates a string of N spaces.
NUM$(N) POS. # GETS A LEADING BLANK IN THE SIGN POS.	Indicates a string of numeric characters representing the value of N as it would be output by a PRINT statement. For example: NUM$(1.ØØØØ) = 1. PUTS A BLANK ON TRAILING END OF STRING.
VAL(A$)	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
TIME$(Ø)	Generates the time of day as a string of the form: 12:59 AM

For example, the following multiple line function (see section 9.1) returns the string which comes first in alphabetical order:

```
10 DEF FNF$(A$,B$)
20 FNF$=A$
30 IF A$>B$ THEN FNF$=B$
40 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. The following line is unacceptable.

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

The following code is a string function which returns the leftmost 5 digits from the sum of 3 arguments:

```
75 DEF FNA$(X,Y,Z) = LEFT(NUM$(X+Y+Z),5)
80 PRINT FNA$(1E6,1E5,1E4)
```

The output of line 80 looks as follows:

```
11100
```

CHAPTER 6

INTEGER VARIABLES AND INTEGER ARITHMETIC

6.1 INTEGER VARIABLES

Normally, all numeric values (variables and constants) specified in a BASIC program are stored internally as floating point numbers (three words per value). If operations to be performed deal with integer numbers, significant economies in storage space can be achieved by use of the integer data type (which uses only one word per variable). Integer variables (and constants) can assume values in the range -32,768 to +32,767.

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

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

6.2 INTEGER ARITHMETIC

Arithmetic performed with integer type variables is performed modulo 2↑15. 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:

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

The result of the expression on the right is truncated to provide an integer value for X%.

As another example, the statement below:

```
1Ø PRINT 2/4, 2%/4, 2/4%, 2%/4%
```

when executed causes the following to be printed:

```
        .5            .5            .5            Ø
```

6.3 INTEGER I/O

Input and output of integer type 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 integral value it is automatically printed as an integer but is still stored internally as a floating point number and hence takes more storage space.) For example:

```
         1Ø READ A, B%, C, D%, E
         2Ø PRINT A, B%, C, D%, E
         3Ø DATA 2,3,4.Ø,5.7,6.8
```

when executed prints:

```
     2             3             4             5             6.8
```

6.4 USER DEFINED INTEGER FUNCTIONS

Functions can be written to handle integer variables as well as floating point variables (see sections 3.7.3 and 9.1). The 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:

```
         1Ø DEF FNR%(I%,J%) = I%-J% * (I%/J%)
```

and could be called later in a program as follows:

```
         1ØØ PRINT FNR%(A%,11%)
```

Integer arguments can be used where floating point arguments are expected and vice versa. However, strings cannot be used where numbers are required (or vice versa).

```
         75 DEF FNA$(X%) = X%-1
         8Ø 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 can be used within IF statements in any place that a relational expression can appear. An integer value of Ø% corresponds to the logical value FALSE, and non-zero integers are defined to be TRUE. Any logical operators specified (AND, OR, NOT, XOR, IMP, EQV) operate on logical data in a bitwise manner; thus the integer -1% (which is represented internally as sixteen binary ones) is normally used when a TRUE value is required.

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

The following Immediate Mode sequence illustrates the use of integers in logical applications:

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

READY
IF -1% AND Ø% THEN PRINT "TRUE" ELSE PRINT "FALSE"
FALSE

READY
IF 4% AND 2% THEN PRINT "TRUE" ELSE PRINT "FALSE"
FALSE

READY
IF -1% IMP -1% THEN PRINT "TRUE" ELSE PRINT "FALSE"
TRUE

READY
IF 1<Ø XOR -1% THEN PRINT "TRUE" ELSE PRINT "FALSE"
~~FALSE~~ TRUE

READY
```

CHAPTER 7

DATA STORAGE CAPABILITIES

7.1 FILE STORAGE

Thus far, techniques have been presented for entering data into a program as it is written (via READ and DATA statements) or when it is executed (via the INPUT statement). Both of these techniques pose operational problems when the amount of data a program reads or writes is increased beyond a few items. In order to alleviate these problems, BASIC-Plus provides the user with a facility to define Input/Output files.

A BASIC-Plus file consists of a sequence of data which is transmitted to (or from) a BASIC program from (or to) an external Input/Output device connected to RSTS-11. The external device can be the user's terminal, some other terminal, the RSTS system disk, a line printer, magnetic tape, or high-speed paper tape equipment. Each file has both an external name by which it is known within the system and an internal file designator (a number used to refer to the file within the program). A special statement (OPEN, see section 7.4) is used to associate an external name with an internal designator.

An external file name is completely specified with the following information:

device:filename.extension [*proj,prog*]

where the <u>device</u> can be one of the following:

DF:	~~DSK:~~	system disk
DTn:	~~DTAØ to DTA7:~~	DECtape units 0 to 7
PR:	~~PTR:~~	high-speed paper tape reader
PP:	~~PTP~~:	high-speed paper tape punch
LP:	~~LPT:~~	line printer
CR:	~~CDR:~~	card reader
MR:	~~MSR:~~	mark-sense card reader
KB:	~~TTY:~~	user's terminal
KBn:	~~TTYn~~:	terminal n in the system
	$	system library
MTn.		magtape unit n

The <u>filename</u> is a six character (maximum) alphanumeric name. The <u>extension</u> is a three character (maximum) alphanumeric file name extension usually specifying the type of file. The extensions used by the system are as follows (the user can create his own extensions):

.BAS	BASIC source program, ASCII format
.BAC	Compiled BASIC program, binary format
.TMP	System scratch file for temporary use

A user can have up to 12 files open (with internal designators 1 through 12) for access at any given time. Each open file consumes a buffer within core storage. The buffer sizes for various devices are indicated in Table 7-1. If a buffer cannot be created for a file, due to a lack of storage space in core, then the file cannot be opened. (The process of opening a file is described in section 7.2)

Table 7-1

DEVICE BUFFER SIZES

Device	Buffer Size
DF: ~~DSK:~~	256 words
DTn: ~~DTAn:~~	256 words
PR: ~~PTR:~~	64 words
PP: ~~PTP:~~	64 words
LP: ~~LPT:~~	64 words
KB: ~~TTY:~~	64 words

7.2 OPEN STATEMENT

The OPEN statement is used to associate a file on a bulk storage device or an I/O device with an internal file designator. This allows the file to be readily referenced in INPUT, PRINT, and (in some cases) DIM statements. The format of the OPEN statement is as follows:

$$\textit{line number}\ \ \text{OPEN}\ <\textit{string}> \left\{\text{FOR} \begin{bmatrix}\text{INPUT}\\ \text{OUTPUT}\end{bmatrix}\right\}\ \text{AS FILE}<\textit{expression}>$$

For example:

```
        1Ø OPEN "P~~T~~R:" FOR INPUT AS FILE 1
        2Ø INPUT #1, A$
```

Line number 10 causes the paper tape reader to be opened as an input source with the internal file designation 1. Line number 20 causes input to be accepted from file #1, and the input is associated with the variable A$.

The formats for the INPUT and PRINT statements to be used with the OPEN statement are as follows:

line number INPUT #<*expression*>,<*list*>
line number PRINT #<*expression*>,<*list*>

where the expression is the same expression present in the OPEN statement (the internal file designator) and the list is a list of variable names (or constants as explained in the sections on the PRINT and INPUT statements). (The virtual matrix dimension statements reference OPEN statements without the FOR INPUT or FOR OUTPUT phrase, as explained later.)

The string portion of the OPEN statement can be specified as a string constant (in quotes) or as a variable (with its value deferred until program execution). The string portion indicates the device on which a file is to be opened.

RSTS-11 distinguishes between two kinds of devices: those that are directory-structured (disk and DECtape) and those which are not (all others). When indicating a file for either input or output on a directory structured device, it is necessary to specify both a device name and a file name (and, optionally, an extension). On non-directory structured devices, the device name alone identifies the file. Thus:

DF: ~~DSK:~~	is insufficient information to specify a file
DF: ~~DSK:~~FRED	is sufficient to specify the file FRED on the disk
PP: ~~PTP:~~	uniquely specifies the high-speed punch
PP ~~PTP:~~FOO	specifies the high-speed punch, and FOO is ignored

File name syntax is such that the device name DSK: is not required if a file name is specified. Thus:

```
DSK:FOO
```

is equivalent to:

```
FOO
```

When a device is not specified, a file name alone always indicates the disk as a default storage device. However, to store a file on DECtape, the device must be specifically indicated:

```
DTA4:FOO
```

The following sequence is useful and allows for easy change in the device to be used before program execution begins:

```
1Ø LET I$ = "PTR:"
2Ø OPEN I$ FOR INPUT AS FILE 1
3Ø INPUT #1, A$
```

If a file being opened for input does not exist, an error message is returned. If a file being opened for output does not exist, it is created. If a file for output already exists and is not write-protected, it is deleted and recreated.

If an assignable device is referenced in any OPEN statement and that device is unavailable for assignment, an error message is printed.

File names used in an OPEN statement are composed of up to six alphanumeric characters with an extension of up to three alphanumerics. Thus, an output file could be created as follows:

```
1Ø OPEN "DSK:SCRTCH.TMP" FOR OUTPUT AS FILE N1
```

Thereafter, reference can be made to file SCRTCH.TMP on device DSK: as follows (notice that the internal file designator is represented as a variable, although its value must still be between 1 and 12):

```
1ØØ PRINT #N1, A$, B$
```

The internal file designator (following the # character in the

INPUT or PRINT statements) is always in the range 1 to 12. File designator #Ø is, by definition, always open as the user's terminal. File #Ø cannot be closed or opened as any other device or file. Use of file #Ø is indicated below (no OPEN #Ø is necessary):

```
1Ø INPUT #Ø, A$
```

is equivalent to:

```
1Ø 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 ("TTY:") on some internal file designator other than Ø, since the "?" is only generated for input requests on file #Ø, as shown in the following example:

```
LISTNH
1Ø OPEN "TTY:" AS FILE 1
15 PRINT "WITH USE OF INTERNAL FILE DESIGNATOR:"
2Ø PRINT "TYPE YOUR NAME ";
3Ø INPUT #1, A$
4Ø PRINT "FOR COMPARISON, WITHOUT FILE DESIGNATOR:"
45 INPUT "TYPE YOUR NAME ";A$

READY

RUNNH
WITH USE OF INTERNAL FILE DESIGNATOR:
TYPE YOUR NAME CHARLIE BROWN
FOR COMPARISON, WITHOUT FILE DESIGNATOR:
TYPE YOUR NAME ? CHARLIE BROWN

READY
```

If a file is being opened for both input and output or to be referenced as virtual disk matrices (see section 7.5), the form:

line number OPEN<*string*>AS FILE<*expression*>

is used. If the file indicated by the name "string" is found, it will be used and, if it is not found, it will be created.

When a program uses a statement such as:

```
5Ø OPEN "FOO" AS FILE 4
```

it can perform input and output to that file. However, such a file (FOO on the system disk) can only be referenced in a sequential fashion. If data is already in the file, it can be read via INPUT statements similar to the manner in which a READ statement pulls data from the DATA statement pool. Any attempt to use a PRINT statement with the file FOO will work only if there is nothing already in that file. If data already exists in the file FOO, a PRINT statement will begin to write over any data beyond the point where the INPUT stopped. This is not a recommended technique since the entire file will be garbled and useless.

7.3 OUTPUT TO VARIOUS DEVICES

In order to direct output to a device other than the user terminal, the PRINT command is formatted as follows:

line number PRINT #<*expression*>,<*list*>

where the expression is the internal file designator of a previously opened output file (see section 7.2). The list of information to be output can include any of the output information described as applicable to the PRINT statement. For example:

```
1Ø OPEN "DATA1" FOR OUTPUT AS FILE 1
2Ø PRINT #1, "START OF DATA FILE"
```

The above lines open a file called DATA1 on the disk with internal file designator #1 (of 12 possible open files available in the system). 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
1Ø LET I$ = "LPT:"
2Ø OPEN I$ FOR OUTPUT AS FILE 1
3Ø PRINT #1, I,SQR(I) FOR I= 1 TO 5
4Ø END

READY

RUNNH
```

The results would appear on the line printer as follows:

```
        1                1
        2                1.4142136
        3                1.732Ø5Ø8
        4                2
        5                2.236Ø68
```

7.4 INPUT FROM VARIOUS DEVICES

Like the PRINT statement, the INPUT statement can operate upon devices other than the user terminal. The form:

line number INPUT #<*expression*>,<*list*>

causes input to be accepted from the previously opened file or device indicated in the expression (see section 7.1). As long as the value of the expression is non-zero, the specified file is read through one of the 12 available user I/O buffers (internal file designators). If the expression is zero, or missing completely, input is from the user terminal. No ? character is printed on the terminal paper when input is requested from a device other than the terminal, opened on file #Ø. For example:

```
1Ø OPEN "PTR:" FOR INPUT AS FILE 3
2Ø INPUT #3, A$, B$
```

causes the strings A$ and B$ to be read from the high-speed paper tape reader.

Note that spaces are ignored in input data. If the user wants to read numeric data from a file previously created (on disk or DECtape, for example) he should insert commas in the data when he prints the data in the file. For example:

```
1ØØ OPEN "DTA4:LEN" AS FILE 1
11Ø PRINT #1, A "," B "," C
2ØØ OPEN "DTA4:LEN" AS FILE 1
21Ø INPUT #1, A,B,C
22Ø PRINT A,B,C
```

is an acceptable sequence to print three values onto a DECtape file, read them from that DECtape file, and print the three values on the user terminal. As in the example above, once a file is opened it can be closed and reopened through the use of a second OPEN statement. Reopening the file moves the positioned pointer within the file back to the beginning of the file, so that the entire file becomes available again for sequential referencing. This serves much the same function as a RESTORE statement would to the pool of DATA statements.

7.5 VIRTUAL DATA 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 core memory for data storage than is economically feasible. BASIC-Plus fills both these requirements with its easy-to-use random access file system, called virtual core.

Essentially the BASIC-Plus virtual core system provides a mechanism for the programmer to specify that a particular data matrix is not to be stored in the computer's core memory, but within the RSTS-11 file system, instead. Data stored in files external to the user program will survive, even after the user leaves his terminal, and can be retrieved by name at a later RSTS-11 session. Items within the file are individually addressable, as are items within core matrices. In fact, it is the similar way in which data are treated in both core and random-access files which leads to the name virtual core.

The matrix format is used to store data because in a normal data file, described earlier, the PRINT and INPUT statements deal only with the next sequential data element. A normal data file, then, is much more limited in its applications and depends upon a strictly sequential treatment of I/O. With virtual data storage, the user can reference any element of the file, no matter where in the file it resides. This random access of data allows the program to perform non-sequential referencing of the data for use in any BASIC statement (which is to say that the virtual core matrices need not be read into core to be available to the program for use).

In order for a matrix of data to exist in virtual core, it must be declared in a special form of the DIM statement (placed in program sequence somewhere after the corresponding OPEN statement). This special statement is as follows:

line number DIM #<~~expression~~ integer const.>,<*list*>

where the ~~expression~~ <integer const.> is an integer constant between 1 and 12 and corresponds to the internal file designator on which the program has opened an internal file (see below and also section 7.2). The variable list appears as it would for a normal core resident matrix DIM statement. Thus, a 100 by 100 matrix could be defined as:

```
10 DIM #12, A(100 ,100 )
```

Floating point numbers, integers, and strings can all reside in virtual core matrices. More than one matrix can be specified in one virtual core file. For example:

```
25 DIM #1, A(1000), B%(2000), C$(2500)
```

which allocates space for 1000 floating point numbers, 2000 integer numbers, and 2500 character strings (16 characters long each).

One of the few differences in data handling between core and disk matrices occurs in the storage of strings within string matrices in virtual core. Strings in the computer core memory are of variable length from 0 to any arbitrary length. Strings in virtual core are of fixed length (all elements having a particular name are of the same length). This length can be defined by the programmer and varies from 1 character to 512 characters. The system forces lengths to be a power of 2:

1, 2, 4, 8, 16, 32, 64, 128, 256, 512

If the user indicates other than one of these values, he will receive the next higher size. Thus:

```
10 DIM #1, X$(10)= 65
```

is the same as:

```
10 DIM #1, X$(10)= 128
```

If no length is specified, a default length of 16 characters is assumed. The length attribute of virtual core strings is specified in the DIM statement, using the notation:

```
15 DIM #1, A$(100) = 32, B$(100)=4, C$(100)
```

where A$ consists of 101 strings of 32 characters each;
B$ consists of 101 strings of 4 characters each;
C$ consists of 101 strings of 16 characters each.

If a length attribute is given in a DIM statement for an in-core string matrix, it is ignored, since core storage can be allocated dynamically to hold a string of any length.

In order for the user to reference his virtual core file, he must first associate one of his files (known by name) with an internal file designator from 1 to 12 (which is then used in the virtual DIM declaration). This is normally done with the following OPEN statement:

line number OPEN<*string*>AS FILE<*expression*>

where the string is the name of a disk file and the expression specifies an internal file designator; thus:

```
35 OPEN "FOO" AS FILE 1
```

associates the file named "FOO" with internal file 1. If "FOO" already exists, then the existing file is used; if there is no file named "FOO" one would be created. If the user wishes to destroy any old "FOO" file, he can write:

```
35 OPEN "FOO" FOR OUTPUT AS FILE 1
```

which causes the file to be deleted if it already exists and recreated. If the user wanted to be alerted that the file "FOO" is not present, he could write:

```
35 OPEN "FOO" FOR INPUT AS FILE 1
```

which would cause an error message to be printed if "FOO" is not found.

Recoverable errors will occur when using virtual core if the user program does any of the following:

1. Reference a virtual matrix without first opening the file.
2. Reference a non-disk file (for example, DECtape or the line printer) as a virtual matrix.

Virtual array files must be closed before Chaining, if the file is to be read correctly when re-opened.

3. Exceed virtual core, that is, define a matrix which is bigger than the amount of available disk storage on the system or the amount which the user is allowed to own.

Sophisticated users are urged to read Appendix E which describes the system implementation of the virtual core processor. A mastering of this information will produce programs which utilize the system resources in a highly efficient manner.

As an example of virtual core usage, consider the problems of implementing an information retrieval system for a small organization. There might be 1000 employees, each needing a 256-character record containing the name, home address, phone, work station, and phone extension of the employee. Rather than order the records in the file, it is decided to maintain a separate index file containing only badge numbers. The order of employee records in the master file is the same as the badge number sequence in the index file. Thus, to extract information on an employee with badge n, we find his badge number in the index file and use the index found to retrieve his data from the master file. Since the number of employees is small, integer data can be used in the badge file; only alphanumeric data is stored in the master file.

A program to print an employee's name, given his badge number, might appear as follows:

```
 1Ø !PROGRAM TO LOOK UP NAMES IN MASTER FILE
 2Ø  OPEN "BADGE" AS FILE 1     !BADGE FILE
 3Ø  OPEN "MASTER" AS FILE 2    !MASTER FILE
 4Ø  DIM #1, B%(1ØØØ)           !1ØØØ BADGE NUMBERS
 5Ø  DIM #2, A$(1ØØØ)=256       !1ØØØ RECORDS, EACH 256
                                !CHARACTERS LONG
 6Ø  INPUT "BADGE NUMBER";E%    !GET EMPLOYEE NUMBER FROM TTY
 7Ø  GOTO 1ØØ IF B%(I)=E% FOR I=1 TO 1ØØØ     !IS BADGE # IN FILE?
 8Ø  PRINT "NO SUCH EMPLOYEE": GOTO 6Ø        !NO
100  !WE NOW HAVE INDEX INTO FILE,I           !YES
11Ø  R$ = A$(I)                 !BRING RECORD INTO CORE
12Ø  PRINT "NAME IS"; MID(R$,1Ø,15)   !NAME IS FROM COLUMN 1Ø TO 25
13Ø  GOTO 6Ø                    !NEXT...
```

7.6 CLOSE STATEMENT

The CLOSE statement is used to terminate I/O to or from a device. Once a file has been closed, it can be reopened for reading or writing on any internal file designator. All files are automatically closed at the end of program execution. The format of the CLOSE statement is as follows:

line number CLOSE *<expression>*

CLOSE does not write an end-of-file marker. To mark a disc file so that end-of-file error can operate, write "Z^C" (CHR$(26Ø)) after last record.

Any number of files can be closed with a single CLOSE statement; if more than one, they are separated by commas. The expression indicated is the same expression used in the OPEN statement and indicates the internal file designator. By closing a file with the CLOSE statement, the user frees more core storage space to open other files (a maximum of 12 depending upon the space available). For example:

```
255 CLOSE  2, 4
345 CLOSE  1Ø
```

Line 255 above closes the files opened on internal device designators 2 and 4. Line 345 closes the file open on internal device designator 1Ø.

7.7 NAME-AS STATEMENT, FILE PROTECTION AND RENAMING

The NAME-AS statement is used to assign protection codes to a file (and to rename an existing file). The format of the command is as follows:

line number NAME<*string*>AS<*string*>{<*protection*>}

The specified file (the first string indicated) is renamed (as the second string indicated). A file protection code can be specified within typed angle brackets, although it is not required. If a new file protection is specified, it is reflected in the protection assigned to the renamed file.

Each user in the system is given a unique project-programmer number which identifies him to the system. This is a two-part number of the form [1Ø,2Ø] where "10" is the user's project or group number and "20" is his individual programmer number. Files are protected from various user classes determined by the project-programmer number. Specifically, files can be protected against:

the owner (creator)
other programmers in the owner's group
all others

The protection is part of the second string and is itself a string.

The following table contains the codes used to determine file protection.

Code	Protection Against	Meaning
1	owner	read protect
2	owner	write protect
4	group	read protect
8	group	write protect
16	others	read protect
32	others	write protect

The file protection can be specified as the sum of any combination of these values; for example:

```
1ØØ NAME "FOO" AS "FOO<48>"
```

would deny access to the file FOO to anyone not in the owner's group (32 plus 16).

```
275  NAME "DTA2: MATRIX" AS "MAT2"
```

renames the file MATRIX on DECtape 2 as MAT2 (in this case on DECtape 2; the device specification need not be repeated).

7.8 KILL STATEMENT

The KILL statement is of the form:

line number KILL<*string*>

and causes the file named string to be deleted from the user's file area. For example, when the user has completed all work with the file XYZ.TMP on the system disk, he could remove the file from storage by executing the following statement:

```
455 KILL "XYZ.TMP"
```

A user is not allowed to KILL a file that is write-protected against him. (He must use the NAME-AS statement to change its protection first.)

CHAPTER 8

MATRIX MANIPULATION

The following chapter deals with intrinsic functions to handle matrices. 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. All MAT operations ignore the zero element of each matrix (A(Ø), B(Ø,Ø)).

8.1 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:

line number 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:

```
1Ø  DIM A(2Ø,2Ø)
2Ø  MAT READ A
```

The above lines read a twenty by twenty matrix of floating point data. Data is read in row by row; that is, the second subscript varies most rapidly. If line 30 had read:

```
2Ø  MAT READ A(5,15)
```

a 5 by 15 matrix would be read (the remaining elements of A are zero).

8.2 MAT PRINT STATEMENT

The MAT PRINT statement prints each element of a one or two dimensioned matrix. The statement is of the following form:

line number MAT PRINT #<*expression*>, <*matrix name*>

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.

Variables may be used as subscripts in the < matrix name list >.

Where the expression is not included, output is to the user terminal.

If the matrix name is followed by a semicolon (;), the values are printed in a packed fashion; otherwise, each element is printed in its own zone.

```
 1Ø  DIM A(1Ø,1Ø), B(2Ø,2Ø)
12Ø  MAT PRINT A;              !PRINT 1Ø*1Ø MATRIX, PACKED FORMAT
13Ø  MAT PRINT B(1Ø,5)         !PRINT 1Ø*5 MATRIX, 5 ELEMENTS PER
                               !LINE
```

In order to direct the output of the MAT PRINT statement to a device other than the user terminal, an internal file designator is used following the opening of such a file. For example:

```
 1Ø  DIM A(2Ø,5),B(1Ø,2Ø)
11Ø  OPEN "LPT:" AS FILE 2
12Ø  MAT PRINT #2, A
13Ø  MAT PRINT #2, B;
```

The above lines cause the matrices A and B to be output to the line printer. Remember only one matrix can be output by one MAT PRINT statement. In order to perform a jump to the top of the next line printer page, send a CTRL/L as part of a PRINTER statement to the line printer, or type CHR$(12) at the end of a PRINT statement.

It is also possible to obtain printing of both row and column matrices. For example:

```
    1Ø DIM A(7), X(5)
    2Ø MAT READ A,X
    3Ø MAT PRINT A;:PRINT: MAT PRINT X
    4Ø DATA  21,22,23,34,35,36,37,51,52,53,54,55
    5Ø END

    RUNNH

21  22  23  34  35  36  37

51
52
53
54
55
```

The format:

```
MAT PRINT V
```

prints the matrix V as a column matrix.

```
MAT PRINT V,
```

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

```
MAT PRINT V;
```

prints the matrix V as a row matrix, closely packed.

8.3 MAT INPUT STATEMENT

The MAT INPUT statement is used to input the value of each element of a matrix from a specified input device. Where no particular device is specified, the input is accepted from the user terminal. For example:

```
2ØØ MAT INPUT A(2Ø)
```

will cause BASIC to accept 20 floating point values as elements of the matrix A from the user terminal.

A statement of the format:

line number MAT INPUT {#<*expression*>,} <*variable list*>

causes the input to be read from a file or device, indicated by the expression, which has been previously opened. (For details, see the sections on the OPEN and INPUT statements, sections 7.1 and 7.4)

If the input is to be from the user terminal, a ? character is printed to indicate that the program is ready to accept input. If input is from another specified device or file, no ? character is printed.

Depending upon the variable names, the MAT INPUT statement can allow the input of integer, floating point, or character string values. For example:

```
        2ØØ OPEN"DTA1:MATN" FOR INPUT AS FILE 4
        2Ø5 DIM #4, N$(3Ø)
        21Ø MAT INPUT #4, N$
```

reads 30 elements of the file MATN on DECtape unit 1 and equates them with the elements of the character string matrix N.

8.4 MATRIX INITIALIZATION STATEMENT

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

line number MAT<*name*>=<*value*>{(DIM1,DIM2)}

The name element is the name of a particular matrix, and the optional DIM1 and DIM2 specifications indicate the maximum size of the matrix elements which are to be initialized to the value specified. The value can be one of the following:

Value	Meaning
ZER	Sets all elements of the matrix to Ø (this is true of all matrices when they are first created)
CON	Sets all elements of the matrix to 1
IDN	Sets up an identity matrix

All matrices used in a matrix initialization statement must be previously declared in a DIM statement. If no dimensions are indicated (DIM1 and DIM2 are not specified), then the maximum dimensions of the matrix are assumed. For example:

```
         1Ø DIM A(1Ø,1Ø), B(15), C(2Ø,2Ø)
        1ØØ MAT A = ZER                 !SETS ALL ELEMENTS OF A=Ø
        1Ø1 MAT B = CON (1Ø)            !SETS FIRST 1Ø ELEMENTS OF B=1
        1Ø2 MAT C = IDN(1Ø,1Ø)          !C MUST BE SQUARE
```

It should be noted, however, that these instructions have no effect on row and column zero. Thus, the following instructions:

```
 1Ø DIM M(2Ø,7)
 2Ø MAT READ M(7,3)

1ØØ MAT M=CON
11Ø MAT M=ZER(15,7)

2ØØ MAT M=ZER(16,1Ø)
```

first read in a 7 by 3 matrix for M. Then they set up a 7 by 3 matrix of all 1's for M (the actual dimension having been set up as 7 by 3 in line 20). Next they set up M as a 15 by 7 all-zero matrix. (Note that although this is larger than the previous M, it is within the limits set in 10.) An error message results because of line 200. The limit set in line 10 is (20+1) x (7+1) = 168 components, and in line 200 the program calls for (16+1) x (10+1) = 187 components. Thus, although the zero rows and columns are ignored in MAT instructions, they play a role in determining dimension limits. For example:

```
 2ØØ MAT M=ZER(25,5)
```

would not yield an error message.

Perhaps it should be noted that an instruction such as MAT READ M(2,2) which sets up a matrix and which, as previously mentioned, ignores the zero row and column does, however, affect the zero row and column. The redimensioning which may be implicit in an instruction causes the relocation of some numbers; therefore, they may not appear subsequently in the same place. Thus, even if we have first LET M(1,0) , M(2,0) = 1, and then MAT READ M(2,2), the values of M(1,0) and M(2,0) now are 0. Thus, when using MAT instructions, it is best not to use row and column zero.

8.5 MATRIX CALCULATIONS

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

8.5.1 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 statement is begun with the word MAT and 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 in the statement, although if two matrices are not conformable to an operation, a subset of one matrix can be indicated as part of the operation.

The following operation is acceptable:

```
1Ø4 DIM A(5Ø), B(25), C(25)
1Ø5 MAT C = A + B
```

Multiplication of conformable matrices is indicated as follows:

```
11Ø MAT C = A*B
```

By conformable matrices is meant that the number of columns in matrix A is equal to the number of rows in matrix B. The operation A =A*B is illegal.

Scalar multiplication of a matrix is performed as follows:

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

Each element of matrix A is multiplied by the scalar value (constant, variable, or formula) K, indicated in parentheses. If K is not specified, matrix A is copied into matrix C (providing sufficient space is available for matrix C) as shown below:

```
12Ø MAT C = A
```

The form A=(K)*A is legal.

8.5.2 Matrix Functions

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

```
15Ø MAT C = TRN(A)
```

causes matrix C to be set equal to the transpose of matrix A. C(I,J) = A(J,I) for all I,J.

```
151 MAT C = INV(A)
```

causes C to be computed as the inverse of matrix A (A must be a square matrix). After the inversion is complete, the function DET is set to the determinant of matrix A and can be tested to decide whether a singularity was found. The value of DET, then, can be used as a variable in any formula. For example:

```
2ØØ MAT A = INV(X):D1=DET
21Ø MAT B = INV(Y):D2=DET
22Ø IF D1<>(-D2) GOTO 34Ø ELSE PRINT "RELATIONSHIP TRUE"
```

```
151 MAT C = INV(A)
```

causes C to be computed as the inverse of matrix A (A must be a square matrix). After the inversion is complete, the function DET is set to the determinant of matrix A and can be tested to decide whether a singularity was found. The value of DET, then, can be used as a variable in any formula. For example:

```
2ØØ MAT A = INV(X):D1=DET
21Ø MAT B = INV(Y):D2=DET
22Ø IF D1<>(-D2) GOTO 34Ø ELSE PRINT "RELATIONSHIP TRUE"
```

Matrix inversion, like the other BASIC Plus matrix operations, does not operate on the elements of the row Ø and column Ø of the matrix; unlike the other operations, inversion destroys the previous content of these elements.

CHAPTER 9

ADVANCED STATEMENT FEATURES

9.1 DEF STATEMENT, MULTIPLE LINE FUNCTION DEFINITIONS

In Chapter 3 the DEF statement is described as having the ability to create a one-line function which the user can call as an element in a BASIC statement. The user has, by now, probably felt the need for a user-defined function which can extend onto more than one line; such a facility is available. The format for a multiple line definition is as follows:

```
line number   DEF  FN<identifier><(dummy arguments)>
              <body of definition>
line number   FN<identifier>= <expression>
line number   FNEND
```

The multiple line DEF function is distinguished from the one line use functions by the absence of an equal sign following the function name on the first line. (From zero to ~~eight~~ 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 must be a statement of the form:

```
line number {LET} FN<identifier> = <expression>
```

It is the value of this expression which is returned as the value of the function.

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

```
1Ø DEF FNM(X,Y)
2Ø LET FNM = X
3Ø IF Y<=X THEN 5Ø
4Ø LET FNM = Y
5Ø FNEND
```

As another example, the following function computes N-factorial:

```
LISTNH
1Ø DEF FNF(M)
2Ø IF M=1 THEN FNF=1 ELSE FNF=M*FNF(M-1)
3Ø FNEND
35 INPUT "VALUE FOR FACTORIAL";M
4Ø PRINT M" FACTORIAL EQUALS "FNF(M)

READY

RUNNH
VALUE FOR FACTORIAL? 4
4  FACTORIAL EQUALS 24

READY
```

Any variable which is not an argument in a 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 or itself). There must not be a transfer from within the definition to outside its boundaries or from outside the definition into it. The line numbers used by the definition must not be referenced elsewhere in the program.

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

```
1Ø DEF FNB(X)
2Ø X=Ø :FNB=1Ø
3Ø FNEND
4Ø A=1
5Ø B=FNA(A)
6Ø PRINT A, B
RUNNH
 1    1Ø

READY
```

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

```
LISTNH
1Ø DEF FNA$(A,B,C)
2Ø IF A>B GOTO 4Ø
3Ø FNA$=CHR$(A+1):GOTO 5Ø
4Ø FNA$=CHR$(A+C)
5Ø FNEND
6Ø INPUT "VALUES FOR A,B,C";A,B,C
7Ø PRINT "FNA$(A,B,C) = "FNA$(A,B,C)

READY
```

(Cont.d next page)

Note: The function in a statement will be evaluated before the address of the left-side variable is computed. Be careful with array subscript warning!

```
RUNNH
VALUES FOR A,B,C? 36,2,31
FNA$(A,B,C) = C

READY
```

9.2 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:

line number 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:

```
5Ø ON X GOTO 1ØØ, 2ØØ, 3ØØ
```

transfers control to line number 1ØØ if the value of X is 1, to line number 2ØØ if X is 2, and to 3ØØ if X is 3. Any other values of X (other than 1, 2, or 3 in this example) cause a program error which will terminate execution or cause the error subroutine specified via the ON ERROR GOTO statement to be entered (see section 9.4 for details).

9.3 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 in the same manner as the ON-GOTO statement described in the previous section. The statement is of the form:

line number ON *<expression>* GOSUB *<list of line numbers>*

Depending on the integer value 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.

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.

An example of the statement follows:

```
80 ON X GOSUB 900,933,1014
```

When line 80 is executed, the value of X being either 1, 2, or 3 will cause control to transfer to line 900, 933, or 1014 respectively. If X is not equal to 1, 2, or 3, the error message.

```
ON STATEMENT OUT OF RANGE AT LINE 80
```

is printed.

9.4 ON ERROR GOTO STATEMENT

Certain errors can be detected by BASIC while executing a user program. 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 simple mathematical errors are corrected (or compensated for) automatically by BASIC. Some applications, however, may require the continued execution of a user program after an Input/Output 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:

line number ON ERROR GOTO <*line number*>

If an error does occur, the user program execution is interrupted and the error subroutine is started at the line number indicated. The variable ERR in the program assumes one of the values listed in Table 9.1.

Table 9.1

USER RECOVERABLE ERRORS

(F) = Execution terminated if no ON ERROR GOTO statement is present in the program.

ERR	Message Printed	Meaning
1	BAD DIRECTORY FOR DEVICE	File lookup is impossible. (F)
2	ILLEGAL FILE NAME	Name contains embedded blanks or non-acceptable characters. (F)
3	FILE IS CURRENTLY OPEN	User cannot write in a file while other users are reading it. (F)
4	NO ROOM ON DEVICE	No storage space to store more data. (F)
5	CAN'T FIND FILE	The file to be opened for input is not present on the specified device. (F)
6	NOT A VALID DEVICE	The specified device is not present in the system. (F)
7	I/O CHANNEL ALREADY OPEN	The channel must be closed before another OPEN can occur on that channel. (F)
8	DEVICE NOT AVAILABLE	Another job is using the specified device. (F)
9	I/O CHANNEL NOT OPEN	The user tried to read or write on an unopened channel. (F)
10	PROTECTION VIOLATION	The user tried to read or write a file protected against him or open another user's file that was both read and write protected against him. (F)
11	END OF FILE ON DEVICE	User has tried to read beyond the end of his data. (F)
12	OPERATION ABORTED	Serious I/O failure. (F)
13	DATA ERROR ON DEVICE	Parity error detected on operation. (F)
14	DEVICE OK?	Device seems to be off-line or in need of service. (F)
15	TELETYPE WAIT EXHAUSTED	User did not respond on Teletype within allotted time. (F)
16	FILE OF SAME NAME EXISTS	User is renaming a file with a name already in use. (F)
43 ~~17~~	VIRTUAL CORE NOT ON DISK	User has tried to use a non disk device for a virtual core matrix. (F)
44 ~~18~~	~~VIRTUAL CORE EXCEEDED~~ ARRAY TOO BIG	There is not enough disk available to store an entire virtual core matrix. (F)
45 ~~19~~	VIRTUAL ARRAY NOT OPENED	Referenced virtual core matrix without opening file first. (F)

17 TOO MANY DT OUTPUT USERS

18 ILLEGAL UUO FOR USER

Table 9.1 (Cont.)

ERR	Message Printed	Meaning
46 ~~20~~	ILLEGAL I/O CHANNEL	User has tried to open a channel greater than 12. (F)
47 ~~21~~	LINE TOO LONG	A record longer than 256 bytes was read from an I/O device. (F)
48 ~~22~~	FLOATING POINT ERROR	Underflow or overflow has occurred.
49 ~~23~~	ARGUMENT TOO LARGE IN EXP	Inaccurate results will follow.
50 ~~24~~	ARGUMENT TOO LARGE IN SIN	Inaccurate results will follow.
51 ~~25~~	INTEGER ERROR	Overflow has occurred.
52 ~~26~~	ILLEGAL NUMBER	A non-numeric character was encountered in a number.
53 ~~27~~	TRANSCENDENTAL ERROR	Attempt to take log of Ø or negative number.
54 ~~28~~	IMAGINARY SQUARE ROOT	User requested square root of negative number. SQR(ABS(X)) returned.
55 ~~29~~	SUBSCRIPT OUT OF RANGE	Undefined matrix element referenced. (F)
56	CAN'T INVERT MATRIX	
57 ~~30~~	OUT OF DATA	User has exhausted DATA list. (F)
58 ~~31~~	ON-STATEMENT OUT OF RANGE	The expression in an ON-statement was less than 1 or greater than the number of line numbers specified.
59 ~~32~~	NOT ENOUGH DATA IN RECORD	User tried to input more data then existed in the next logical record of file.
18 ~~33~~	ILLEGAL UUO FOR USER	Invalid use of system function. (F)
60	INTEGER OVERFLOW, FOR LOOP	
61	DIVISION BY Ø	

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 where the program can interrogate the variable ERR to discover precisely what problem occurred and decide what action is to be taken.

After the problem is corrected (if this is both possible and desired by the program), execution of the user program can be resumed through use of the RESUME statement. The RESUME statement causes the program statement 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:

```
1ØØØ RESUME
1ØØ1 RESUME 1ØØ
```

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

```
1ØØØ RESUME Ø
```

Line 1001 above will restart the user program at line 100 (which can be used to print some terminal message for that particular operation.

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:

line number   ON ERROR GOTO Ø

which allows the system to handle errors or, equivalently:

line number   ON ERROR GOTO

in which case line Ø is assumed. Executing this statement causes the system to treat errors as it would if no ON ERROR GOTO had ever been executed. (Anything other than a line number following the GOTO is equivalent to ON ERROR GOTO Ø.)

It is sometimes inconvenient to be continually turning the error subroutine facility on and off. For this reason, BASIC-Plus allows the statement

```
ON ERROR GOTO Ø
```

to be executed within the error subroutine itself. Special treatment is accorded this case, in that the disabling occurs retroactively; the error which caused entry to the error subroutine is then reported through the normal system error reporting facilities.

As an example of the usage of this feature, consider an application in which inexperienced students 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 9.10) and then to alert it that the user has not replied. The program then prints additional information for the user.

```
  1Ø  ON ERROR GOTO 1ØØØ    !SET UP ERROR ROUTINE
  2Ø  WAIT (6Ø)             !6Ø SECONDS TO RESPOND
  3Ø  INPUT "YOUR NAME";N$  !GET STUDENT NAME

1ØØØ  !THIS IS THE ERROR HANDLING ROUTINE
1ØØ1  IF ERR <> 15 THEN ON ERROR GOTO Ø   !TIME ERROR HANDLER ONLY
1Ø1Ø  PRINT                 !SKIP TO NEW LINE
1Ø2Ø  PRINT "PLEASE TYPE YOUR NAME"
1Ø3Ø  PRINT "AND THEN HIT THE 'RETURN' KEY"
1Ø4Ø  RESUME                !TRY AGAIN
```

9.5 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 specified condition.

The IF-THEN-ELSE statement is exactly the same as the IF-THEN statement, except that rather than falling through to execute 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:

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

where the condition is defined as one of the following:

<relational expression> <logical operator> <relational expression>

and a relational expression is defined as:

<expression> <relational operator> <expression>

as described in section 3.5. The specified condition is tested and if it is true the THEN/GOTO part of the statement is executed. If the condition is false, then the ELSE part of the statement is executed. Following the word ELSE can be either a statement to be executed or a line number to which control is transferred.

The IF-THEN-ELSE statement can appear anywhere in a multiple statement line. As an example of an IF-THEN-ELSE statement:

```
75 IF X>Y THEN PRINT "GREATER" ELSE PRINT "NOT GREATER"
```

Since any statement can follow either the THEN or ELSE in the above statement, it is possible to nest IF statements to any desired level. For example:

```
1ØØ  IF A>B THEN IF B>C  THEN PRINT "A>B>C"
```

The message A>B>C is printed only if the two conditions ~~epe~~specified are both true. An equivalent statement could be expressed as follows:

```
1ØØ  IF A>B  AND B>C THEN PRINT "A>B>C"
```

To further clarify the relationship between the quantities A, B, and C, the following statement could be used:

```
1ØØ  IF A>B THEN IF B>C THEN PRINT "A>B>C"
     ELSE IF A>C THEN PRINT "A>C>B"
```

(Notice the use of the LINE FEED character to continue the statement on a second line.) There are two ways in which the above statement could be interpreted. One in which the ELSE clause is attached to the test A>B and one in which it is attached to the test B>C. In order to resolve such situations, BASIC-Plus uses the following rule:

> An ELSE clause is always associated with the nearest unmatched IF-THEN to its LEFT.

Thus in the above statement the ELSE clause is executed only if the B>C test is false. If the A>B test had been false, execution would have immediately proceeded to the next statement and nothing else in statement 100 would be executed or tested.

If any clause within a complex IF statement is satisfied, then execution proceeds to the ~~statement~~ next line immediately following the IF statement. ~~This is either the next higher numbered statement or the~~ Any statement to the right of a colon (:) on the same physical line as the IF statement is within the scope of the "THEN" or the "ELSE". It should be noted that several physical lines can be included in one logical line by use of the line feed character.

The reader is invited to consider the following program:

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

READY

RUNNH

? 3,6,1
B>A>C

READY

RUNNH
? 2,9,21
C>B>A

READY
```

The use of the line feed and tab characters greatly improves the legibility of complex program statements such as line 20 above.

9.6 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:

line number 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:

line number   FOR<*variable*>=<*expression*>{STEP<*expression*>}WHILE<*condition*>

and

line number   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 first the two loops below:

```
1Ø FOR I = 1 TO 1Ø
11 PRINT I;
12 NEXT I
```

and

```
2Ø FOR I = 1 UNTIL I>1Ø
21 PRINT I;
22 NEXT I
```

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 20, 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
1Ø X=1Ø
2Ø FOR I=1 TO X
3Ø X=X/2
4Ø PRINT I,X
5Ø NEXT I
1ØØ X=1Ø
11Ø FOR I=1 UNTIL I>X
12Ø X=X/2
13Ø PRINT I,X
14Ø NEXT I
```

(Cont. next page)

```
RUNNH
1                   5
2                   2.5
3                   1.25
4                   .625
5                   .3125
6                   .15625
7                   .Ø78125
8                   .Ø390625
9                   .Ø1953125
1Ø                  .9765625E-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.

```
1Ø FOR I = 1 UNTIL I=N OR X(I)=Ø AND X(I+1)=Ø
2Ø NEXT I
```

9.7 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.

9.7.1 The IF Statement Modifier

The form:

<*statement*>IF<*condition*>

is analogous to the form:

IF<*condition*>THEN<*statement*>

For example:

```
1Ø PRINT X IF X<>Ø
```

is the same as:

```
1Ø IF X<>Ø 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:

```
1ØØ IF 1=1 THEN PRINT "HELLO" ELSE PRINT "BYE" IF 1=Ø
```

will print

```
HELLO
```

since the test 1=1 is true. The modifier IF 1=Ø is false, 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:

```
1ØØ PRINT X(I,J) IF I=J IF X(I,J)<>Ø
```

which will print the value of X(I,J) only if the value of (I,J) is non-zero and if I equals J. Whenever 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 one is executed last. This situation is described by the term "nested modifiers".

An additional operational advantage of this interpretation of IF modifiers is illustrated in the discussion of FOR modifiers below.

9.7.2 The UNLESS Statement Modifier

The form:

<*statement*> UNLESS <*condition*>

causes the statement to be executed only if the condition is <u>false</u>. For example, the following statements are all equivalent:

```
1Ø PRINT A UNLESS A=Ø
1Ø PRINT A IF NOT A=Ø
1Ø IF NOT A=Ø THEN PRINT A
1Ø IF A<>Ø THEN PRINT A
```

This particular form simplifies the negation of a logical condition.

9.7.3 The FOR Statement Modifier

The form:

<*statement*> FOR <*variable*>=<*formula*>TO <*formula*> STEP <*formula*>

can be used to imply a FOR loop on a single line. For example:

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

This statement is equivalent to the following FOR-NEXT loop:

```
1Ø FOR I = 1 TO 1Ø
11 PRINT I, SQR(I)
12 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 is the case with all modifiers, a FOR modifier in an IF statement will operate 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, dimensioned to be 100 elements long, the following program:

```
1Ø DIM X(1ØØ)
15 READ X(I) FOR I=1 TO 1ØØ
2Ø IF X(I)<>Ø THEN PRINT I,X(I) FOR I=1 TO 1ØØ
```

will *not* operate properly, since the implied FOR loop at line 20 applies *only* to the THEN PRINT... part of the statement, and *not* to the IF... part. Thus, 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 2Ø, not as an IF statement, but rather as a PRINT statement with nested modifiers; for example:

```
2Ø PRINT I,X(I) IF X(I)<>Ø FOR I=1 TO 1ØØ
```

when expressed in the latter form, the nested modifier rule takes effect, and all values of X(I) are tested and printed as appropriate.

9.7.4 The WHILE Statement Modifier

The form:

<statement> WHILE *<condition>*

is used to repeatedly execute the statement *while* the specified condition is *true*. For example:

```
1Ø LET X=X↑2 WHILE X↑2<1E6
```

is equivalent to:

```
1Ø LET X=X↑2
11 IF X<1E6 THEN 1Ø
```

The WHILE modifier (and the UNTIL modifier below) will operate 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 statements will never terminate properly; such programs are sometimes called "infinite loops":

```
1Ø X=X+I WHILE I<1ØØØ
2Ø PRINT I, A(I) WHILE A(I)<>Ø
```

In both cases, the program fails to alter the values which are used to determine when the loop is done.

Successful applications of the WHILE modifier include the following:

```
5  !TEST OF SQUARE ROOT ROUTINE
1Ø X=X+1 WHILE X=SQR(X↑2)
2Ø PRINT X
```

9.7.5 The UNTIL Statement Modifier

The form:

<statement> UNTIL *<condition>*

is used to repeatedly execute the statement until the statement becomes true; which is to say, while the statement is false. For example:

```
1Ø X=X+1 UNTIL X<>SQR(X↑2)
```

is the same as

```
1Ø X=X+1
2Ø IF X=SQR(X↑2) THEN 1Ø
```

9.7.6 Multiple Statement Modifiers

More than one modifier can be used in a single statement. Multiple modifiers are processed from right to left. For example:

```
1Ø LET A=B IF A>Ø IF B>Ø
```

which is equivalent to:

```
1Ø IF B>Ø THEN IF A>Ø THEN A=B
```

or

```
1Ø IF B>Ø AND A>Ø THEN LET A=B
```

or

```
1Ø IF B=Ø THEN 4Ø
2Ø IF A=Ø
3Ø LET A=B
```

A two dimensional matrix (m by n) can be read a row at a time as follows:

```
75  READ A(I,J) FOR J=1 TO M FOR I=1 TO N
```

which is equivalent to:

```
75  MAT READ A(M,N)
```

and to:

```
1Ø FOR I=1 TO M
2Ø FOR J=1 TO N
3Ø READ A(I,J)
4Ø NEXT J
5Ø NEXT I
```

Also see section 9.8.3, interaction of FOR and IF.

9.8 ADDITIONAL PRINT STATEMENT FEATURES

9.8.1 PRINT-USING Statement

If elaborate formatting is required for an output operation, the PRINT-USING statement is available. The statement is of the form:

line number PRINT{#<*expression*>,}USING<*string*>,<*list*>

If the expression is omitted, the output is assumed to be directed to the terminal. The string can be either a string constant or a string variable and is interpreted as an exact image of the line to be printed, using the following notation:

a. An exclamation point (!) identifies a one character string field. The string is specified in the <list> part of the PRINT statement. For example:

   ```
   1Ø PRINT USING "!!!", "AB","CD","EF"
   ```

   causes

   ```
   ACE
   ```

   to be printed. The first character from each of the first three string constants or variables is printed. Any other characters beyond the first are ignored.

b. A character string field of two or more characters is indicated by spaces enclosed between backslashes. The backslash character is produced by typing SHIFT/L on the keyboard. If zero spaces are enclosed, a field two columns wide is assumed, one space indicates a field three columns wide, etc. For example:

```
   2Ø PRINT USING "\\\ \", "ABCD","EFGHI"
```

causes:

```
   ABEFGH
```

to be printed. The first two backslashes have no spaces enclosed, hence permit the printing of two characters (AB). The second two backslashes enclose two spaces and hence permit the printing of four characters (EFGH). Notice that no spaces are printed unless they are specifically planned.

c. Numeric fields are indicated with the # character. Any decimal point position can be specified and rounding is performed as necessary (not truncation). For example:

```
   1Ø PRINT USING "###.##", 12.345
```

will cause:

```
   12.34
```

to be printed, while:

```
   1Ø PRINT USING "####.", 12.345
```

causes:

```
   12.
```

to be printed. Numeric fields are right justified; that is, if a number does not fill the allotted spaces leading blanks precede the number. Where the field specified is too small for a constant or variable to be printed, the * character is used to fill the field. For example:

```
   1Ø PRINT USING "##", 1ØØ
```

causes:

```
   **
```

to be printed.

d. When the exponential form of a number is desired, the numeric field is followed by the string ↑↑↑↑ (four ↑ characters). This suffix allocates space in the line for E ±xx. Any arrangement of decimal points is possible. For example:

```
   5  F$="##↑↑↑↑      ######  ## #"
   1Ø A=1E4
   2Ø PRINT USING F$,A, A, A, A
```

prints the following:

```
   1E+Ø4    1ØØØØ  ** *
```

Thus, in order to print the line:

```
   A=1.23  B=2.45
```

where A=1.23111 and B=2.45457, format control can be used as in the following line:

```
  1Ø LET F$="A=##.##  B=##.##"
  2Ø PRINT USING F$, A,B
```

e. Notice that when the PRINT-USING statement is used, the usual PRINT statement punctuation characters (commas and semicolons) have no effect.

The PRINT USING statement is generally employed where a large amount of data is to be formatted for output in a particularly uniform manner.

9.8.2 PRINT Functions

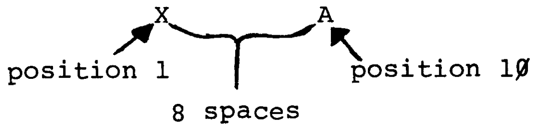
In order to aid in formatting simple and complex PRINT statements, the following functions are provided:

Function	Meaning
POS(n)	Returns the current position of the print head (imaginary for disk files) where n is the I/O channel number. POS(Ø) returns the value for the user's terminal.
TAB(x)	Tab to position x in the print record. For example, the Teletype paper has 72 printable columns. TAB(4) causes a movement of the print head to column 4 regardless of whether the print head is currently before or after print position 4. (If currently past position 4, position 4 of the next line will be selected.
CHR$(x)	Output the single ASCII character corresponding to the number x.

For example:

```
1Ø PRINT "X";TAB(1Ø);CHR$(65)
```

causes the following to be printed:



Other character string functions for use with the PRINT statement (or any other statement) are described in the section on Character String Functions, section 5.5.

9.9 INPUT LINE STATEMENT

The INPUT LINE statement causes the input of a string of characters (either from a specified file or the terminal) from the beginning of one line up to the LINE FEED character to be read into the string variable specified. The statement is of the form:

line number INPUT LINE {#<*expression*>} <*string variable*>

Where the input is to be from the Teletype no *expression* is used to indicate the source of the input. For example:

```
75 INPUT LINE A$
```

causes the system to pause while the user types a line of input (on the terminal up to 72 characters) followed by the RETURN key. The RETURN key generates a carriage return/line feed, which is appended to the other data typed by the user. Thus, every character typed (including such characters as space and quote) is passed to the program, which can analyze the data using the normal string manipulation functions. This is different from the normal mode of inputting strings, since ",', and <space> are normally disregarded, although not here. For example:

```
1Ø INPUT LINE A$: PRINT "LINE:" A$;
2Ø INPUT A$: PRINT "STRING:"; A$
3Ø GOTO 1Ø
? ABCD
LINE:ABCD
? ABCD
STRING:ABCD
? A B C D
LINE: A B C D
? A B C D
STRING:ABCD
?"ABCD"
LINE: "ABCD"
? "ABCD"
STRING:ABCD
↑C
READY

9Ø OPEN"DTA4:DATA" FOR INPUT AS FILE  4
95 INPUT LINE #4, B$
```

causes the input of a line of data from the file DATA on DECtape unit 4 through I/O channel 4.

9.10 SYSTEM FUNCTIONS

RSTS-11 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 9.2

Table 9.2

SYSTEM FUNCTIONS

Function	Meaning	Sample Usage
DATE$(Ø)	returns the current day, month, and year~~, and time in~~ the form: 2-MAR-71 ~~11:ØØ~~	1Ø PRINT DATE$(Ø)
DATE$(N)	returns a character string corresponding to the Julian date N+70,000 DATE$(1) = "Ø1-JAN-7Ø" DATE$(24Ø) = "Ø5-MAY-7Ø"	155 PRINT X%(I),DATE$(I)
TIME$(Ø)	returns the current time of day as a character string as follows: TIME$(Ø) = "Ø5:3Ø PM"	75 IF TIME$(Ø) ≥"Ø5:45 PM" PRINT "TIME TO QUIT"
TIME$(N)	returns a string corresponding to the time at N minutes before midnight, for example: TIME$(1) = "11:59 PM" TIME$(144Ø) = "12:ØØ AM" TIME$(721) = "11:59 AM"	1Ø PRINT #4,TIME$(I),I
TIME(Ø)	returns the clock time in seconds since midnight.	1Ø IF TIME(Ø)>432ØØ THEN PRINT "AFTERNOON"
TIME(1)	returns the central processor (CPU) time used for this job in Ø.1 second quanta.	1Ø IF TIME(1)>3Ø THEN STOP
TIME(2)	returns the connect time (time during which the user has been logged into the system) for this job in minutes.	1Ø IF TIME(2)>18ØØ THEN STOP
SLEEP(X)	this function is used as a statement and causes the currently running program to be dismissed until either X seconds have elapsed or a line is typed at the user's terminal, whichever comes first. No error is generated if input is not received.	75 SLEEP (1Ø)

Table 9.2 (Cont.)

Function	Meaning	Sample Usage
WAIT(X)	this function is used as a statement and has no immediate effect. Failure to supply input on a terminal within X seconds after an INPUT statement is executed will result in the generation of error condition #15.	4Ø ON ERROR GOTO 1ØØ 45 WAIT(15) 5Ø INPUT "ANSWER";X ... 1ØØ IF ERR=15 THEN RESUME
WAIT(Ø)	causes the system to wait as long as necessary for input from the terminal.	7Ø WAIT(Ø)

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:

line number READ *<list of variables>*
line number DATA *<list of values>*

The list of variables can include floating point, integer, subscripted, or character string variables. The list of values must correspond in type with the variables to which the value will be assigned (the exception is that integer and floating point values are interchangeable, although they are stored according to the type of the variable).

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 lowest, etc. The location of DATA statements in a program is irrelevant, although for simplicity they are usually kept toward the end of the program. (The DATA statements must occur in the proper numeric sequence, however.) A DATA statement must be the only statement on a line, although a READ statement can occur anywhere on a line.

If a READ statement is out of data, an error message is printed and program execution is terminated. (This error can be treated through the ON ERROR GOTO statement, section 9.4.)

Quotes are necessary in DATA statements only around string items which contain a comma or where embedded blanks within the string are significant.

Matrices are read from DATA statements via the MAT READ statement of the form:

line number MAT READ *<list of matrices>*

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:

```
1Ø  DIM A(2Ø,2Ø), B(5Ø)
2Ø  MAT READ A, B(35)
```

The above lines read values for the 20 x 20 matrix A and 35 out of the possible 50 values for the B matrix (remaining elements are zero). Data is read in 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:

line number RESTORE

For example:

```
85  RESTORE
```

causes the next READ statement following 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 3.9 for an example program using the RESTORE statement.

The RESTORE statement can be placed in any position on a multiple statement line.

10.3 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:

line number INPUT{#*<expression>*,} *<variable list>*

In many cases the simpler form:

line number 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. If sufficient values are not typed, the system prints another ?; if too many values are typed, excess values are ignored. This last form also allows the user to intersperse strings to be printed between the variables to be input. For example:

```
1Ø  INPUT "YOUR NAME IS";N$, "AGE"; A
```

when executed would allow the following interaction at the terminal (the underlined characters are typed by the user):

```
YOUR NAME IS? JEAN
AGE? 23
```

The format:

line number INPUT #<*expression*>, <*variable list*>

causes input to be read from the file or device indicated in the expression by the number it was given when it was opened. (See section 7.2 or 10.5 for a description of the OPEN statement.) If the value of the expression is non-zero and the specified file is open as using the Teletype as an input device, then no ? character is printed at the terminal when input is requested. For example:

```
75 OPEN "TTY:" FOR INPUT AS FILE 2
8Ø INPUT #2, A
```

The system then pauses while the user types a numeric value for the variable A, although no prompting ? or character string message is or can be printed on the terminal.

Another format of the INPUT statement allows for the entering of an entire line of data as a single character string entity, regardless of embedded spaces or punctuation. This format is:

line number INPUT LINE{#<*expression*>,}<*string variable*>

For example:

```
        25  INPUT LINE A$
```

would pause and allow the user to enter characters on the terminal keyboard. The end of the line being input is the carriage return/ line feed sequence. As another example:

```
        2Ø  OPEN "DSK:F2" FOR INPUT AS FILE 7
        25  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) to input a line of characters up to the next LINE FEED character. (See Table 7-1 for the size of buffers available for each device.)

The MAT INPUT statement is used to input the values of a predimensioned matrix from a specified input device. Where no particular device is specified, the input is accepted from the user terminal. For example:

```
        2ØØ  MAT INPUT A(2Ø)
```

causes 20 floating point values to be accepted as elements of the matrix A. A statement of the form:

line number MAT INPUT {#<*expression*>,}<*variable list*>

causes the input to be read from a file or device indicated by the expression and previously opened.

```
        45  DIM B(1Ø,25)
        5Ø  OPEN"DTA2:DTA1" FOR INPUT AS FILE 1
        55  MAT INPUT #1, B(1Ø,25)
```

The above lines cause the file DATA1 on DECtape 2 to be open 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. Where input is from the user terminal a ? is printed; however, reference to another device 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 PRINT STATEMENT

In its simplest form, the PRINT statement looks as follows:

line number PRINT

This causes a carriage return/line feed to be performed on the user terminal. The format:

line number 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; where an element is not a simple variable or constant, it is evaluated before a final value is printed. The list can also contain a character string between quotes which will be printed exactly as it appears.

NOTE

> If a character string is enclosed in a PRINT statement with an initial quote and no terminating quote, the terminating quote is considered to be the end of that PRINT statement. For example:
>
> ```
> 1Ø PRINT "NAME IS A$
> 1Ø PRINT "NAME IS A$"
> 2Ø PRINT "NAME IS" A$
> ```
>
> Line 1Ø is shown in two equivalent forms. Line 2Ø 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:

```
1Ø  A=1: B=2: C=3
15  PRINT A; A+B+C, C-A, "END"
```

when executed, will cause the following line to be printed:

```
1  6       2                END
```

A terminal line is considered to be divided into five print zones of fourteen 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 fifth 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:

a. if a 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 space if it is positive. The number is then followed by a single space.

b. formatted character strings (explained later) followed by a semicolon, are printed with no preceding or trailing spaces, except as explained in (a) above.

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 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 include:

a. suppression of leading and trailing zeros to the right of a decimal point. Where a number is an integer, printing of the decimal point is also suppressed.

b. at most seven significant digits are printed.

c. most numbers are printed in decimal format unless the user indicates otherwise. Numbers too large or too small to be printed in decimal format are printed in exponential format.

d. character string constants are printed without leading or trailing spaces.

e. extra commas cause print zones to be skipped.

f. semicolons separating character string constants from other list items are optional; if used, their significance is as already defined; if not used, no extra spaces separate the character string from the value to be printed.

Output can be directed to a device other than the Teletype with the following command:

line number PRINT #<*expression*>,<*list*>

Where the expression is the number of a previously opened output file, out of the 12 possible opened files (see section 10.5). For example:

```
1Ø  OPEN "PTP:" FOR OUTPUT AS FILE 3
5Ø  PRINT #3, B, D, A+7, FNX(B)
```

causes four values to be punched onto paper tape by the high speed punch which is opened for output as file 3 out of the possible 12 files.

In order to perform formatted output, the following statement is used:

line number PRINT #<*expression*>, USING <*string*>, <*list*>

where the expression (which is optional) indicates the file or device which is the destination of the output; the string is either a string constant or a string variable which is an exact image of the line to be printed; and the list is a list of items to be printed. The string constant is constructed according to the following rules:

a. an exclamation point identifies a one character string field. The string is specified in the list within the PRINT statement. For example:

```
1Ø PRINT USING "!!!", "AB", "CD", "DE"
```

which causes:

```
ACE
```

to be printed at the user's terminal.

b. a variable string field of two or more characters is indicated by spaces enclosed between backslashes. The backslash character (\) is produced by typing SHIFT/L on the keyboard. Enclosing no spaces indicates a field two columns wide, one space is equivalent to three columns wide, etc.

```
2Ø PRINT USING " \\\  \", "ABCD", "EFGHI"
```

causes

```
ABEFGH
```

to be printed at the user's terminal.

c. numeric fields are indicated with the # character. Any decimal point arrangement can be specified, rounding is performed as necessary. For example:

```
3Ø  PRINT USING "###.##", 12.345
```

causes

```
12.36
```

to be printed on the user's terminal

```
4Ø  PRINT USING "####", 12.345
5Ø  PRINT USING "####.", 12.345
6Ø  PRINT USING "##", 1ØØ
```

cause

```
  12
  12.
  **
```

to be printed on the user's terminal.

d. when the exponential form of a number is desired, the numeric field is followed by the string ↑↑↑↑ which allocates space for E±ØØ. Again, any arrangement of decimal points is permitted. For example:

```
  5   F$= "##↑↑↑↑ ######  ## #"
  1Ø  A=1ØØØØ.
  2Ø  PRINT USING F$, A,A,A,A
```

causes

```
  1E+Ø4    1ØØØØ  **  *
```

to be printed at the user's terminal.

As another example:

```
 5  LET A=1.32111: B=2.45457
1Ø  LET F$= "  A=##.##  B=##.##"
2Ø  OPEN "LPT:" FOR OUTPUT AS FILE 4
3Ø  PRINT #4, USING F$, A,B
```

would cause:

```
A= 1.32  B= 2.45
```

to be printed on the line printer. Notice that when the PRINT USING statement is used, commas and semicolons have no effect on the output formatting.

The MAT PRINT statement allows for easy printing of a predimensioned matrix or matrices. The statement is of the form:

line number MAT PRINT #<*list of matrices*>,

For example:

```
15  DIM A(16), B8(5,10);C%(32),E1(32)
25  MAT PRINT A(15), B$(3,7), C%(32), E1
```

If the elements of the list of matrices are the unsubscripted names of the matrices, the entire matrix is printed. If the elements are subscripted, then the subscript 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:

```
  5  DIM A(1Ø,1Ø),B(1Ø,2Ø)
 12Ø  MAT PRINT A:              !MATRIX A IS PRINTED IN PACKED FORMAT
 13Ø  MAT PRINT B(1Ø,1Ø)        !1Ø*1Ø MATRIX PRINTED, 5 VALUES PER
                                ! LINE
```

Row and column matrices can also be printed. For example:

```
  5  DIM A(5), B(1Ø)
 5Ø  MAT PRINT A;
 6Ø  MAT PRINT B
```

Line 50 causes A to be printed as a row matrix, closely packed; line 60 causes B to be printed as a column matrix. The form:

```
 7Ø  MAT PRINT A,
```

would cause the matrix A to be printed as a row matrix, five values per line.

10.5 OPEN STATEMENT

The OPEN statement has two formats:

line number OPEN <*string*> FOR $\begin{bmatrix}\text{INPUT}\\\text{OUTPUT}\end{bmatrix}$ AS FILE<*expression*>

line number OPEN <*string*> AS FILE <*expression*>

The first form is used to open files for input or output of variables through use of INPUT and PRINT statements. The second form is generally used to open files for input and output of virtual core matrices through the MAT INPUT and MAT PRINT statements.

The string in the OPEN statement is either a character string constant or variable and represents the device on which a file is to be opened. In the case of a disk or DECtape file, the name of the file is also indicated. The various device names are as follows:

MTn:		magtape unit n
DF:	~~DSK:~~	system disk
DTn:	~~DTAØ~~: to ~~DTA~~7:	DECtape units 0 to 7
PR:	~~PTR:~~	high-speed paper tape reader
PP:	~~PTP:~~	high-speed paper tape punch
LP:	~~LPT:~~	line printer
CR:	~~CDR~~:	card reader
	MSR:	mark sense card reader
KB:	~~TTY:~~	user terminal
KBn:	~~TTYn:~~	terminal n within the system
	$	system library

Examples of acceptable character strings are shown below:

"DSK:FOO"	specifies the file FOO on the system disk
"DTA4:MMM:	specifies the file MMM on DECtape unit 4
"PTP:"	specifies the high-speed punch
"PTP:FOO"	specifies the high-speed punch, FOO is ignored
"FOO"	specifies the file FOO on the system disk, equivalent to DSK:FOO (disk is the default storage device)
"DSK:FOO.TMP"	specifies the file FOO on the system disk with the extension.TMP

The expression in the OPEN statement is the internal file designator, an integer number (constant or variable) between 1 and 12. Internal file designator Ø is always open and is equivalent to referencing the user's terminal.

For example:

```
1Ø  OPEN "PTP:" FOR OUTPUT AS FILE 3
11  PRINT #3, "BEGINNING OF DATA FILE"
```

The above sequence opens the high-speed punch for output on internal file designator 3 which is then used in a PRINT statement to cause BEGINNING OF DATA FILE to be punched in ASCII code on the paper tape. The paper tape punch can only be opened for output, a message is printed if an attempt is made to open that device for input.

```
1Ø  LET I$ = "LPT:": E=1
11  OPEN I$ FOR OUTPUT AS FILE E
12  PRINT #E, A,SQR(A) FOR A=1 TO 1Ø
```

The above sequence uses variable names in the OPEN statement rather than constants and causes a table to be printed on the line printer.

```
        1Ø  OPEN "DTA2:REM.DTA" FOR INPUT AS FILE 1
        11  INPUT #1, A, D4, B$, C%
```

The above sequence opens the file REM with the extension .DAT for input on internal file designator 1 and then accepts as input four values.

When input is accepted from the user terminal, a ? is printed prior to the acceptance of data input by the system. In order to request input without the prompting character (which is frequently useful as a grammatical device), the user terminal (TTY:) can be opened on some internal file designation other than Ø. (The ? is only generated for input requests on internal file #Ø.) For example:

```
        1Ø  OPEN "TTY:" AS FILE 1
        11  PRINT "TYPE YOUR NAME"
        12  INPUT #1, A$
```

results in the following sequence:

```
        TYPE YOUR NAME
        JOHN DOE
```

When using the

line number OPEN *<string>* AS FILE *<expression>*

form, a normal data file can be opened for input and output or, more often, the statement is used to indicate the presence of a virtual data matrix on the system disk. This virtual data storage facility allows the user to individually address and update elements within a disk file in a random (non-sequential) manner and allows the user to address more data storage area than is available in core at the installation. The OPEN statement is used ~~following~~ preceding a DIM statement defining the virtual data matrix. The DIM statement is of the form:

line number DIM #*<expression>*,*<list>*

For example:

```
        1Ø  OPEN "DATA1" AS FILE #1
        11  DIM #1, A(1ØØØ,12ØØ), B$(1ØØØ), C%(5ØØ)
```

The OPEN statement gives the name of the file as DATA1 on the system disk.(only the disk can be used for virtual core storage). The internal file number associated with that particular disk file is 1. The matrices A, B$, and C% are then defined as virtual matrices referenced through internal file designator 1.

Floating point numbers, integers, and strings can all reside in virtual core matrices. More than one matrix can be specified in one virtual core file, as shown above.

String matrices, which is to say string variables, are handled slightly differently in virtual storage than when stored in core. In core matrices are of variable length from Ø to any arbitrary length. Strings in virtual core are of fixed length. All elements of a given matrix have the same length (even though all or part of any number of elements can be blank).

The length of virtual core matrices, then, varies from 1 character to 512 characters. Although any length can be specified, the system forces lengths to be a power of two. That is, the actual length of a string matrix element in virtual core is one of the following:

1, 2, 4, 8, 16, 32, 64, 128, 256, 512

When the user indicates a value other than one of the above, the next higher value is automatically assigned. For example:

```
1Ø  DIM #1, X$(1Ø) = 65
1Ø  DIM #1, X$(1Ø) = 128
```

the two lines above are exactly the same in function. The length is specified in the DIM statement, as shown above. For example:

```
15  DIM #1, A$(1ØØ) = 32, B$(1ØØ) = 4, C$(1ØØ) = 16
```

defines three virtual core matrices which can later be referenced in a program where

A$ consists of 101 strings of 32 characters each;
B$ consists of 101 strings of 4 characters each;
C$ consists of 101 strings of 16 characters each.

Each program accessing a particular virtual array file must have one (but only one) DIM statement describing the organization of that file.

In order for the user to reference any element in a virtual core matrix, the matrix must be defined and associated with some internal file designator, as follows:

```
10  OPEN "FOO" AS FILE 1
11  DIM #1, A(1000), B(100), C$(500)=256
```

MAT INPUT and MAT PRINT statements as well as a statement of the form:

Not necessary to actually read the file to reference the array

```
255  INPUT #1, A(255)
```

can then be used referencing the predefined matrices. Any element in virtual core can be addressed independently of the remaining elements of that matrix.

In general, the action of the two different types of OPEN statement can be summarized as follows:

```
20  OPEN "FOO" FOR INPUT AS FILE 1
30  OPEN "FEE" FOR OUTPUT AS FILE 2
```

In the above statements, if a file being opened for input does not exist, an error message is returned. If a file being opened for output does not exist, it is created. If a file for output already exists and is not write-protected, it is deleted and recreated.

```
40  OPEN "FII" AS FILE 3
```

If FII already exists as a file (on the system disk, in this case), then the existing file is used; if there is no file names FII, one is created. (Other devices than the system disk can be specified for this format. However, virtual core matrices can only be referenced on system disk files. Other devices can be referenced for I/O under the rules specified.)

Virtual array files must be closed before chaining!

10.6 CLOSE STATEMENT

The CLOSE statement is used to terminate I/O to or from a device, removing the internal file designation. Once a file has been closed, it can be reopened at any time for I/O on any internal file designator. All files are automatically closed at the end of program execution. The format of the CLOSE statement is as follows:

line number CLOSE *<expression>*

where the expression is the internal file designator. More than one file can be closed with one CLOSE statement. Closing a file before the end of program execution has the advantage of freeing more core storage space to open other files. An example of the CLOSE statement follows:

```
25  CLOSE  4
3Ø  CLOSE  1,  5
```

PART III

USING RSTS-11

This section deals with the interaction between the user and the RSTS-11 terminal, how to enter and edit user programs, and how to give commands to BASIC-Plus.

Clearly, the listing, running, saving, or compiling of a program differ in kind from the individual statements composing the program. System commands perform this type of operation. Other operations include assigning of peripheral devices for program input and/or output, the determining of the length of the current program or number of user files available, renaming and replacing of current files, and the enabling and disabling of the echo feature on the RSTS-11 terminal.

A system command can be given at any point after the system has printed READY and before the user issues a RUN command.

CHAPTER 11

BASIC-PLUS SYSTEM COMMANDS

11.1 ON-LINE WITH BASIC-PLUS

11.1.1 Project-Programmer Numbers and Passwords

Before the user attempts to use the RSTS-11 system, the system manager or an instructor will assign him both a unique project-programmer number and a password. The project-programmer number might, for example, look as follows:

```
100,101
```

The number 100 above is the project number (possibly held by a group of people having a common interest); and the number 101 above is the programmer number (held by only one person within the project group). Thus, each individual's project-programmer number is different. This allows the assignment of protection codes to user files for various relationships among users (see section 7.7 describing the NAME-AS statement).

The user is also assigned a password. This password is an alphanumeric code particularly assigned to an individual user. This password is never printed on the terminal and, hence, allows for a measure of security in limiting the use of the computer system.

11.1.2 HELLO Command

Equipped with the codes to obtain access to the system, the user should find a terminal and turn the LINE-OFF-LOCAL knob to LINE. This puts the terminal on-line to RSTS-11, that is, opens a line of communication between the computer and the terminal.

Once the terminal is on-line, type the word:

```
HELLO
```

followed by the RETURN key. This tells RSTS-11 that a user wishes to join the system. RSTS-11 will print a number sign (#) at the left margin of the paper and wait for the user to type his project-programmer number and the RETURN key. The system responds by printing:

```
PASSWORD:
```

and waiting for the user to type his password followed by the RETURN key. These characters are not printed at the console. If the codes are acceptable to the system, the message:

```
RSTS VØ1A -JOB Ø1 TTYØ 12-MAY-71 12:3Ø PM
MESSAGES OF THE DAY ARE:
```

is printed. If the codes are incorrect, the error message "INVALID ENTRY - TRY AGAIN" is printed and the user can try again.

The entire process of entering the system would look as follows (although the RETURN key is typed to enter a line to the system it does not echo on the terminal paper except to perform a carriage return/line feed operation):

```
HELLO

# 1ØØ,1Ø1
PASSWORD:

RSTS VØ1A -JOB Ø1 TTYØ 12-MAY-71 12:3Ø PM
MESSAGES OF THE DAY ARE:
NEW OR OLD--
```

Once successfully logged onto the system, the user can type "NEW" to create a new program, "OLD" to retrieve a program previously saved, or any other command in this chapter.

11.1.3 BYE Command

Whenever the user is ready to leave the terminal, he types the command:

```
BYE
```

followed by the RETURN key. This tells RSTS-11 that the user has requested to be dismissed from the system. RSTS-11 then removes from core and disk any temporary files which had been created by the system for the user. Any files created by the user and still remaining open on disk (or any I/O device) are closed and saved for the next session.

Before leaving the terminal, the user should turn the LINE-OFF-LOCAL knob to OFF. (Turning the knob to LOCAL means that the terminal has power, but is not connected to the system. It then operates as a typewriter.)

11.2 CREATING A USER PROGRAM

In order to create a new user program, at any time a user can issue the NEW command as follows:

```
NEW
```

followed by the RETURN key). The system responds by printing:

```
NEW PROGRAM NAME--
```

to which the user responds by typing the name of the new program. When typing a new BASIC program, the file name extension .BAS (for BASIC) is added to the name by the system.

Alternatively, the user can give the command NEW followed by the program name, to avoid having the system prompt the typing of the program name:

```
NEW FOO
```

is equivalent to

```
NEW
NEW FILE NAME--FOO
```

When the NEW command is given, it:

a. Deletes any program currently in core, and
b. Causes RSTS to remember the new program name.

```
NEW DTAØ:FOO
```

is meaningless. All checking for duplicate files occurs when the SAVE command is given.

Following the creation of a new file with an acceptable file name, the user can begin to type his program, beginning each line with a line number.

The user has the option of typing the RETURN key instead of indicating a file name. This will cause BASIC to create a file called NONAME which can be referenced later as NONAME. At any time, this name can be changed (see sections 7.7 and 11.5.5). Only one file with the name NONAME can exist at any one time for a given user. The creation of the file NONAME is shown below (the

RETURN key, although typed, does not echo):

```
NEW
NEW PROGRAM NAME--

READY
```

If the SAVE command is now given, it will create NONAME.BAS as a file.

11.3 RECALLING AN OLD PROGRAM

When the user desires to recall the source file of an old BASIC program (previously saved on a storage device), he gives the OLD command as follows:

```
OLD
```

to which the system replies:

```
OLD PROGRAM NAME--
```

The user then types the name of the old BASIC file containing the program. Alternatively, the user can indicate the old file name without prompting, as follows:

```
OLD TAXES
```

which calls the old file TAXES from the disk. If the file is not available on the disk or if it is protected against that user, an appropriate message is printed.

If a file name is preceded by the $ character, the file with the given name is taken from the system library. For example:

```
OLD $DOG
```

calls the file DOG from the system library. The system manager or group instructor will generally provide users with a list of files in the system library which are available for their access. (Many of these files are protected against change by users.)

Where no file name is indicated, BASIC looks for the file NONAME (which could have been created by the user or the system, see section 11.2). For example:

```
OLD
OLD PROGRAM NAME--

READY
```

Whatever had been stored in the file named NONAME is now in core and available to the user.

The OLD command can only retrieve BASIC source programs. Compiled programs can be run but not changed. Any program called with the OLD command can be edited by the user at the terminal.

11.4 EDITING OF USER PROGRAMS

During the course of typing a program at the terminal or after a program is seen to be incorrect, changes can be made in the text of a program. These changes are made in what is called the editing phase of BASIC, between the time when the system prints READY and the time when the user types RUN. (During this time, system commands and Immediate Mode statements can be executed.)

The simplest type of correction is done during the typing of a line before the line is entered to the system with the RETURN key. For example:

```
1Ø DEF FUN(X)=
```

If the user realizes he has typed FUN instead of FNU, he can type the RUBOUT key once for each character to be erased. The RUBOUT key causes the erased character to be echoed on the user terminal between back slashes as they are erased. For example:

```
ABC<RUBOUT><RUBOUT>DEF
```

Typing the above is printed on the terminal as follows:

```
ABC\CB\DEF
```

If the RETURN key is typed at the end of the above line, the system would receive it as follows:

```
          ADEF
```

The letters B and C have been erased.

If the user decides that his easiest course is to delete the entire line, and he has not yet typed the RETURN key, then he can type CTRL/U (hold down CTRL and U keys), which performs this function. If the RETURN key has been typed, then the line may merely be retyped; the second version will replace the first in the computer memory.

11.4.1 DELETE Command

The DELETE command is used to remove one or more lines from the user program currently in core. For example:

```
          DELETE 1ØØ
```

causes line number 100 to be deleted. (The user should first be certain that no other line references line number 100 unless that line is to be replaced.)

```
          DELETE 1ØØ-2ØØ
```

causes all the program lines between and including line numbers 100 and 200 to be deleted. If 100 and/or 200 do not exist in the program, any lines within the range from 100 to 200 are deleted.

If several groups of lines are to be deleted, then the user can type:

```
          DELETE 1ØØ-2ØØ, 3ØØ-4ØØ, 1ØØØ-11ØØ
```

which deletes all lines between 100 and 200, 300 and 400, and 1000 and 1100.

If only one line is to be deleted, it may be more convenient merely to type the line number and the RETURN key

```
          1Ø
```

which is equivalent to:

```
DELETE 1Ø
```

11.4.2 LIST Command

The LIST command is used to obtain a clean printed copy of all or part of the user's current program. This is especially useful during and after an editing session in which the original program is changed.

In order to obtain a printed copy of the entire program as it currently exists within the system, type:

```
LIST
```

In order to list a single line, type:

```
LIST 1ØØ
```

to type line 100.

In order to list a section of the program, type:

```
LIST 1ØØ-2ØØ
```

which will cause the listing of the entire program from line number 100 to line number 200 inclusive.

In each of the above cases, BASIC prints a program header containing the program title, date, and time. If this header material is not desired (as it might not be for normal editing), the command may be given as LISTNH to delete the header material. To summarize:

LIST Command	Meaning
LIST	List the entire user program as it currently exists.
LISTNH	Same as LIST, but without a program header.
LIST n	List line n.
LISTNH n	List line n without a program header.
LIST nl-n2	List lines nl through n2
LISTNH nl-n2	List lines nl through n2, inclusive, without a program header.

A ? is printed at the left of each line which BASIC-Plus considers to be in error. For example:

```
?1Ø PPRINT A+B
```

11.4.3 CONT Command

As explained in section 4.2 on Program Debugging, the STOP statement can be used to cause halts at various points in a user program. Immediate Mode examination of values or changes can be made to the program, followed by the resumption of program execution. Once a program has been stopped, it can be restarted at the point at which execution stopped by giving the

```
CONT
```

command followed by typing the RETURN key. When the CONT command is given,execution continues with the next executable statement following the STOP.

11.5 MANIPULATING USER PROGRAMS

The commands in this section enable the user to compile, save, run, and rename his files. These are all operations performed on a program as a whole (either in core or as a file) and are used once a complete program has been prepared at the terminal.

11.5.1 RUN Command

The RUN command is used to cause the execution of any BASIC program, either source or compiled. (Source programs are stored as the user typed them; compiled programs are files described in section 11.5.7.)

In order to run the program currently in core, the user simply types:

```
RUN
```

This causes the execution of the program in core. A program header is printed after the RUN command is given, consisting of the program name, date, and time. If this information is not desired, the command

```
RUNNH
```

should be given. RUNNH executes the current program without printing the header material.

Where it is desired to run a program not in core, the command:

```
RUN FILENAME
```

can be given. This command causes BASIC to search for the file FILENAME on the disk, load, compile it (if necessary), and run it if it is found.

If FILENAME.BAS (source) and FILENAME.BAC (compiled) both exist, BASIC will execute FILENAME.BAC since it requires less time. In order to retrieve and execute FILENAME.BAS instead of FILENAME.BAC, it is necessary to issue separate OLD and RUN commands. The file is then available for any editing to be performed.

Where the file to be run is not present on the disk but on another storage device, the format:

```
RUN DEV:FILENAME
```

where DEV: is the designation of the storage device. For example:

```
RUN PTR:
```

reads a BASIC program from the high speed reader and runs it.

As with the OLD command, the character $ appearing before the name of a file indicates that the file resides in the system library. If the file can be accessed by the user, he will be allowed to run it (see section on file protection, 7.7). Only files with the extensions .BAS and .BAC can be RUN.

11.5.2 SAVE Command

The SAVE command is used to store BASIC source programs on the disk as follows:

```
SAVE
```

The program currently in core is saved under its file name with the extension .BAS. If a file of the same name exists, then SAVE returns the error message:

```
FILE EXISTS - USE 'REPLACE'
```

Where the current name of the file is not the desired name, the format:

```
SAVE FILENAME
```

can be used, which saves the program currently in core under the name FILENAME.BAS.

In cases where the desired storage device is not the disk, the format:

```
SAVE DEV:FILENAME
```

is used where DEV: indicates the device designation. The file is stored as FILENAME.BAS. For example:

```
SAVE DTA4:ROPE
```

saves the file ROPE.BAS on DECtape 4.

The SAVE command is used only with source files and cannot be used with compiled files. When a program is saved, under some name, the program is still in core to be used or ignored as the user wishes.

To obtain a listing of his program on the line printer, the user can type:

SAVE LP~~T~~:

To punch a tape of his program, the user can type:

SAVE P~~T~~P:

11.5.3 UNSAVE Command

The UNSAVE command is used to remove a file from a storage device. The form:

```
UNSAVE FILENAME.BAC
```

removes the file FILENAME.BAC from the disk. (Unless specified, the device assumed is the disk.) If the command is given:

```
UNSAVE FILENAME
```

BASIC attempts to remove first the file FILENAME.BAS and then the file FILENAME.BAC. Unless only one of the two is specified in the command, both are removed if they are found.

To indicate another device, the form:

```
        UNSAVE DEV:FILENAME
```

is used, where DEV: is the device designation, for example:

```
        UNSAVE DTA4:FOO
```

removes the file(s) FOO.BAS and FOO.BAC from DECtape 4 if they are found.

11.5.4 CHAIN Command, Program Overlaying

If a user program is too large to be loaded into core and run in one operation, the user can segment the program into two or more overlays. Each overlay section is assigned a name and control can be transferred from one section to another with the CHAIN command.

The CHAIN command is of the form:

CHAIN <*string*>{<*line number*>}

in which the string is the name of the next overlay section and the line number specifies the line number at which to begin execution. If no line number is specified, execution begins with the lowest numbered line. For example:

```
        CHAIN "PHASE2" 2Ø
```

causes the current program segment to be overlayed with segment "PHASE2". Execution begins with line 2Ø in the new segment.

Communication between various program segments can be done by means of the user's file area.

When the CHAIN command is executed, all the user's open files are closed, the new segment is loaded, and execution continues.

~~Files must be CLOSED by stmt. before CHAINING, if file is to be read correctly by next program.~~

11.5.5 RENAME Command

The RENAME command causes the name of the program currently in core to be changed to the specified name. For example:

```
RENAME NEWNAM
```

The old name of the program in core is discarded and it is now known as NEWNAM If the SAVE command is given:

```
SAVE
```

the file NEWNAM.BAS would be stored on the disk.

11.5.6 REPLACE Command

The REPLACE command is used when the program in core has the same name as a file on the disk and the user wishes the program in core to become the new file with that name. The command is simply of the form:

```
REPLACE FILENAME
```

REPLACE is like SAVE, but destroys without notice the old copy of same file, if it exists.

11.5.7 COMPILE Command

Normally RSTS-11 reads each line of a user's program as it is typed and, if acceptable, translates the line into a form more easily understood by the PDP-11 computer. As lines within the program are altered, only those lines which are changed need to be recompiled (i.e., translated). When the SAVE command is given, only the source version of the program (i.e., the text that is typed in response to the LIST command) is retained in the specified place. In response to the OLD command, BASIC reads the text from a file and compiles it in much the same manner as is done when the program is read from the user's keyboard.

Once a program is completely developed and debugged, it may be desirable to avoid the time-consuming practice of compiling the program every time it is fetched from the library. For this reason, the COMPILE command has been provided. This command permits the user to save an image of his compiled program, rather than (or in addition to) the source text of the program. This compiled program may be called in from the disk and executed with a minimum of overhead by use of the RUN command (see section 11.5.1).

Due to the transformation that takes place when a program is compiled, a file with the extension .BAC can only be executed, it cannot be edited. Therefore, the user can issue the RUN command with respect to these compiled files, but the file cannot be brought into core with the OLD command.

If the current file name (i.e., that which is typed in the heading of a listing) is "FILEØ1", then the command

```
COMPILE
```

will save the compiled program in a file named FILEØ1.BAC. If another name is desired for the compiled file, it may be specified.

```
COMPILE FOO
```

will generate a file named FOO.BAC.

11.6 SYSTEM STATUS REPORTS

11.6.1 LENGTH Command

The LENGTH command returns the length of the user's current core program. For example:

```
LENGTH
2K OF CORE USED
```

At least 2K of core (1K = ~~4096~~ 1024 words) is reserved for each user with a maximum of 12K per user. The maximum size of a user program depends upon how much core is physically on the system and how much core the system administrator permits any single user to "own".

11.6.2 CATALOG Command

Giving the CATALOG command causes the user's file directory to be printed [redacted] (where no device is specified the list of user files is in the disk account printed on the terminal). For example:

```
CATALOG
FOO      .BAS      4     48       4-APR-71      29-MAR-71      10:50 AM
 ↑         ↑       ↑      ↑           ↖             ↗              ↑
name   extension  size  protection          creation date          time
                                  access date
```

To obtain a CATALOG of files on a device other than the disk, one can give the command

```
CATALOG DEV:
```

For example:

```
CATALOG $
```

lists the files in the system library;

```
CATALOG [1ØØ,1Ø1]
```

lists the files owned by user account 100,101; and

```
CATALOG DTØ4:
```

lists the files on DECtape unit 4.

11.7. USING INPUT/OUTPUT DEVICES

11.7.1 TAPE Command

The TAPE command is used to disable the terminal echo feature when reading a paper tape with the low-speed (terminal) reader. The command is given as follows:

```
TAPE
```

followed by the RETURN key. The tape is then inserted in the low-speed reader and the reader control switch set to START.

Prior to giving the TAPE command, the user will have set up conditions such that the system expects the program. For example, giving the following commands:

```
NEW FOO          [redacted]
TAPE
```

causes the system to await the new program file FOO which is to be entered to the system via the terminal tape reader. Giving the TAPE command disables the echo feature so that the program is not listed on the terminal as it is read. The same function would be served by the following commands:

```
OLD FOO
TAPE
```

11.6.2 KEY Command

Since nothing input at the terminal is echoed once the TAPE command is given, the KEY command is used to enable the user to see what he is typing again. The command is typed as:

```
KEY
```

although it will not be printed on the terminal paper. Once the RETURN key is typed, all other characters typed at the keyboard are echo printed. It may sometimes be necessary to type ↑C to have RSTS-11 "hear" the KEY command.

11.7.3 ASSIGN Command

The ASSIGN command is used to reserve an I/O device for the use of a single programmer. The command is given in the form:

```
ASSIGN DEV:
```

where DEV: is the device designator (see Table 11.1). If the device is present on the system and available for use, the system returns the message:

```
READY
```

If the device is not available for use, the message:

```
DEVICE NOT AVAILABLE NOW
```

is returned. For example:

```
ASSIGN LPT:
READY
ASSIGN PTP:
DEVICE NOT AVAILABLE NOW
```

TABLE 11.1

DEVICE DESIGNATORS

Code	Device
P~~T~~R:	high-speed paper tape reader
P~~T~~P:	high-speed paper tape punch
C~~D~~R:	card reader
MSR:	mark sense card reader
LP~~T~~:	line printer
DT~~A~~Ø: to DT~~A~~7:	DECtape units Ø to 7
DF ~~DSK~~:	disk
KBn ~~TTYn~~:	Teletype n in the system

11.7.4 DEASSIGN Command

The DEASSIGN command is used to release the specified device to the device pool within the system (for use by other users). If no device is specified, all assigned devices are released from that project-programmer number. For example:

```
DEASSIGN LPT:
```

releases the line printer.

```
DEASSIGN
```

releases all devices previously assigned by that user. If a DEASSIGN command is not given before the user leaves the system, an automatic DEASSIGN is performed when the user gives the BYE command.

11.8 SPECIAL CONTROL CHARACTERS

11.8.1 RETURN Key

Typing the RETURN key echoes as a carriage return/line feed operation on the terminal, as long as the terminal is not in TAPE mode.

11.8.2 ESCAPE or ALT MODE Key

The ESCAPE key also terminates the current typed line and causes it to be entered to the system. However, the ESCAPE key echoes on the terminal paper as a $ character and does not perform a carriage return/line feed.

11.8.3 LINE FEED Key

The LINE FEED key is used to continue the current logical line of input on an additional physical line. The LINE FEED key does not echo on the terminal paper but does perform a carriage return/line feed operation when used with the BASIC-Plus system.

Line feeds will produce errors in the user's programs if included in constants (including string constants), verbs, or user specified names for variables or functions.

11.8.4 RUBOUT Key

The RUBOUT key is used as an eraser for the current line. If typed in TAPE mode, the RUBOUT key is ignored; otherwise, it causes the character most recently typed to be deleted. The erased characters are shown on the terminal paper between back slashes. For example:

```
1Ø LEF X=X*X
```

could be corrected by typing the RUBOUT key 7 times (to remove the F) and typing the remainder of the line correctly. The line would look as follows on the terminal paper:

```
1Ø LEF X=X*X\X*X=X F\T X=X*X
```

and would appear to the system as:

```
1Ø LET X=X*X
```

In cases where the mistake is toward the beginning of a line, it may be easier to simply retype the entire line. For example:

```
1Ø LEF X=X*X
1Ø LET X=X*X
```

Once the second line is entered to the system, the first line numbered 10 is deleted.

11.8.5 CTRL/C

By typing a CTRL/C (hold down the CTRL key and type the C key, release both), the user causes BASIC to return to command mode, where commands can be given or editing done. CTRL/C stops whatever BASIC was doing at the time and returns control of the system to the user.

11.8.6 CTRL/U

The CTRL/U combination deletes the current input line. This is useful when a long command has been typed and is no longer wanted. Rather than use the RUBOUT key repeatedly, CTRL/U cancels the entire line. This feature can be used when typing either commands or statements. The entire physical line is deleted.

11.8.7 CTRL/O

The CTRL/O combination suppresses output on the Teletype until the next time CTRL/O is typed (or CTRL/C is typed). When a program produces a large amount of output (usually in tabular form), the user may not wish to wait for the printing of the complete information. CTRL/O enables the user to monitor the output while not stopping it completely. Typing CTRL/O while output is occurring still allows the computer to output the data, but the Teletype does not print it. This speeds up the output process, since the Teletype is a rather slow device. The second time CTRL/O is typed, the output is again sent to the Teletype for as long as the user wishes.

CTRL/C, on the other hand, will completely stop the output. Think of CTRL/O as a switch, the first setting of which creates a condition and the second setting releases the condition.

11.8.8 TAB Character

The TAB character or CTRL/I combination allows the user to insert a tabular format into his typed material. When entering a program to the system, the TAB character allows formatting such as is shown in section 9.5. The BASIC editor considers each line as being broken into tab stops eight spaces apart across the line. Typing the TAB character causes the printing head to move to the next of those stops on the line.

I^c also makes tabbing in text entry.

If using a model 33 Teletype, the TAB echoes as spaces. The model 35 Teletype has built-in hardware tabs.

11.8.9 CTRL/P

The CTRL/P combination is used as a switch to engage or disengage the software tabs created with a model 33 Teletype. If the printer does not position itself at the next tab stop in response to a TAB, then CTRL/P should be typed to "toggle" this switch.

11.8.10 CTRL/B

The CTRL/B combination is used as a switch to stop and start echoing at the user terminal. This feature is useful when operating remotely from a half-duplex Teletype.

RSTS-11 assumes that the terminal user is at a full duplex Teletype, which requires that characters typed at the user terminal be echoed by the system at the user's printer. Some Teletypes provide this local echo function as a part of their normal functioning. Such Teletypes are easily recognized, since each character typed by the user is echoed twice: once by the Teletype and once by the RSTS system. When this symptom is seen, typing CTRL/B will cause RSTS to suppress its Teletype character echo, so that only the local echo is printed. Conversely, RSTS may think a Teletype is half duplex when actually it is full duplex; in this case, typed characters are not echoed at all. Typing CTRL/B will then alter the assumption RSTS makes, and resume Teletype echoing.

11.8.11 CTRL/Z

The CTRL/Z combination is used to mark the end of a file; when inputting data from a file, a CTRL/Z character marks the end of the recorded data. The message "END OF FILE ON DEVICE" is given when a ↑Z is detected.

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) E(5,1)
		N2(8) V8(3,3)

Integer Matrix	any integer variable name followed by one or two dimension elements in parentheses	A%(2) I%(3,5) E3%(4) R2%(2,1)
Character String Matrix	any character string variable name followed by one or two dimension elements in parentheses	C$(1) S$(8,5) A2$(8) V1$(4,2)

A.2 SUMMARY OF OPERATORS

Type	Operator		Operates Upon
Arithmetic	↑	exponentiation	numeric variables and constants
	-	unary minus	
	*,/	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	
Logical	NOT	logical negation	relational expressions composed of string or numeric elements with relational operators
	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 8.5.1 for further details.
	*	multiplication of conformable matrices	
	*	scalar multiplication of a matrix, see Section 8.5.1	

A.3 SUMMARY OF FUNCTIONS

Type	Function	Explanation
Mathematical	ABS(X)	returns the absolute value of X
	ATN(X)	returns the arctangent of X in radians
	COS(X)	returns the cosine of X in radians
	EXP(X)	returns the value of e X, where e=2.1418
	FIX(X)	returns the truncated value of X, SGN(X)*INT(ABS(X))
	INT(X)	returns the greatest integer in X which is less than or equal to X
	LOG(X)	returns the natural logarithm of X, log$_e$X
	LOG1Ø(X)	returns the common logarithm of X, log$_{1Ø}$X
	PI	has a constant value of 3.1415926
	RND(X)	returns a random number between 0 and 1
	SGN(X)	returns the sign function of X, a value of 1 preceded by the sign of X
	SIN(X)	returns the sine of X in radians
	SQR(X)	returns the square root of X
	TAN(X)	returns the tangent of X in radians
Print	POS(X)	returns the current position of the print head for the device X, Ø is the user's Teletype. (This value is imaginary for disk files.)
	TAB(X)	moves print head to position X in the current print record, regardless of current position.
String	CHR$(X)	returns a character string having the ASCII value of X. Only one character is generated.
	ASCII(A$)	returns the ASCII value of the first character in the string A$.
	LEFT(A$,N)	returns a substring of the string A$ from the first character to the Nth character (the leftmost N characters).
	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).
INSTR(N1,A$,B$) VAL (A$) NUM$ (N)	MID(A$,N1,N2)	returns a substring of the string A$ starting with character N1 and being N2 characters long (the characters between and including the N1 to N1+N2-1characters).
SPACE$(N)	LEN(A$)	returns the number of characters in the string A$, including trailing blanks.
System	DATE$(Ø)	returns the current date in the following format: 2-MAR-71
	DATE$(N)	returns a character string corresponding to the Julian date N+70,000 DATE$(1) = "Ø1-JAN-7Ø" DATE$(24Ø) = "Ø5-MAY-7Ø"
	TIME$(Ø)	returns the current time of day as a character string as follows: TIME$(0) = "Ø5:3Ø PM"

Summary of Functions (Cont.)

Type	Function	Explanation
	TIME$(N)	returns a string corresponding to the time at N minutes before midnight, for example: TIME$(1) = "11:59 PM" TIME$(144Ø) = "12:ØØ AM" TIME$(721)= "11:59 AM"
	TIME(Ø)	returns the clock time in seconds since midnight, as a floating point number.
	TIME(1)	returns the central processor time used by the current job in seconds.
	TIME(2)	returns the connect time (during which the user is logged into the system) for the current job in seconds.
	SLEEP(X)	dismiss this job for X seconds
	WAIT(X)	causes the currently running program to be dismissed for either X seconds or until a line is typed at the user terminal, whichever comes first. Generates error condition #15 if wait is exhausted.
	WAIT(Ø)	causes the system to wait for input from the user terminal with no time limit.
Matrix	TRN(X)	returns the transpose of the matrix X, see section 8.5.2.
	INV(X)	returns the inverse of the matrix X, see section 8.5.2.
	DET	following an INV(X) function evaluation, the variable DET is equivalent to the determinant of X.

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 *exp*	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
argument(s) or *arg*	Dummy variable names
statement	Any legal BASIC-Plus statement
string	Any legal string constant or variable
protection	Any legal protection code as described in section 7.7
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.

```
                                                                 Manual
Statement Formats and Examples                                   Section

REM                                                               3.1
     line number  REM <message>
     line number <statement> !<message>
              1Ø  REM  THIS IS A COMMENT
              15  PRINT   !PERFORM A CR/LF

LET                                                               3.2
     line number  {LET}<var> =  <exp>
              55  LET A=4Ø: B=22
              6Ø  B,C,A=4.2      !MULTIPLE ASSIGNMENT

DIM                                                               3.6.2
     line number  DIM <var(dimension(s))>
              1Ø  DIM A(2Ø), B$(5,1Ø), C%(45)

     line number  DIM #<exp>, <var(dimension(s))>=<exp>           7.5
              75  DIM #4, A$(1ØØ)=32,B(5Ø,5Ø)

RANDOMIZE                                                         3.6.3
     line number  RANDOMIZE
              55  RANDOMIZE

IF-THEN, IF-GOTO                  [THEN<statement>    ]           3.5
     line number  IF <cond>       [THEN<line number>  ]
                                  [GOTO<line numbe >  ]
              55  IF A>B OR B>C THEN PRINT "NO"
              6Ø  IF FNA(R)= B THEN 25Ø
              95  IF L<X↑2 AND L<>Ø GOTO 345

IF-THEN-ELSE                                                      9.5
                                  [THEN<statement>   ]
     line number  IF <cond>       [THEN<line number> ]  {ELSE<statement>   }
                                  [GOTO<line number> ]  {ELSE<line number> }

              3Ø  IF B=A THEN PRINT "EQUAL" ELSE PRINT "NOT EQUAL"
              5Ø  IF A>N THEN 2ØØ ELSE PRINT A
              75  IF B= R THEN STOP ELSE 8Ø
```

FOR

```
     line number   FOR<var>=<exp>TO<exp>{STEP<exp>}                   3.6.1
              2Ø   FOR I=2 TO 4Ø STEP 2
              55   FOR N=A TO A+R
```

FOR-WHILE, FOR-UNTIL

```
                                                     [WHILE]           9.6
     line number   FOR <var> = <exp>{STEP<exp>}      [UNTIL] <cond>
              84   FOR I = 1 STEP 3 WHILE I<X
              74   FOR N = 2 STEP 4 UNTIL N>A OR N=B
              05   FOR B= 1 UNTIL B>1Ø
```

NEXT

```
                                                                      3.6.1
     line number   NEXT <var>
              25   NEXT I
              6Ø   NEXT N
```

DEF, single line

```
                                                                      3.9.3
     line number   DEF FN <var(arg)>=<exp(arg)>                       5.5.1
              2Ø   DEF FNA(X) = X↑2 + X                               6.4
```

DEF, multiple line

```
                                                                      9.1
     line number   DEF FN<var> (arg)
             <statements>
     line number   FN <var>= <exp>
     line number   FNEND
              1Ø   DEF FNF(M)     !FACTORIAL FUNCTION
              2Ø   IF M=1 THEN FNF=1 ELSE FNF=M*FNF(M-1)
              3Ø   FNEND
```

GOTO

```
                                                                      3.4
     line number   GOTO <line number>
             1ØØ   GOTO 5Ø
```

ON-GOTO

```
     line number   ON <exp> GOTO <list of line numbers>               9.2
              75   ON X GOTO 95, 15Ø, 45, 2ØØ
```

GOSUB

```
                                                                      3.8.1
     line number   GOSUB <line number>
              9Ø   GOSUB 2ØØ
```

ON-GOSUB

```
     line number   ON <exp> GOSUB <list of line numbers>              9.3
              85   ON FNA(M) GOSUB 2ØØ, 25Ø, 4ØØ, 375
```

RETURN

```
     line number   RETURN
             375   RETURN                                             3.8.2
```

CHANGE

```
                           [<numerical var>]    [<string var>     ]   5.2
     line number   CHANGE  [<string var>   ] TO [<numerical var>  ]
              25   CHANGE A$ TO X
              7Ø   CHANGE M TO R$
              75   CHANGE B TO B$
```

```
OPEN                                                          7.2
                                        ┌INPUT ┐              7.5
    line number  OPEN <string>{FOR      │      │ } AS FILE <exp>  10.5
                                        └OUTPUT┘

             1Ø  OPEN "PTP:" FOR OUTPUT AS FILE B1
             2Ø  OPEN "FOO" AS FILE 3
             3Ø  OPEN "DTA4:DATA.TR" FOR INPUT AS FILE 1Ø

CLOSE                                                         7.6
    line number  CLOSE <list of exp>                         10.6
            1ØØ  CLOSE 2
            255  CLOSE 1Ø, 4, N1


READ                                                          3.3
    line number  READ <list of variables>                     3.9
             25  READ A, B$, C%, F1, R2, B(25)               10.1

DATA                                                          3.3
    line number  DATA <list of values>                        3.9
            3ØØ  DATA 4.3, "STRING",85,1E+4,49,75.Ø4         10.1

RESTORE                                                       3.3
    line number  RESTORE                                      3.9
            125  RESTORE                                     10.2

PRINT                                                         3.3
                                                              3.11
    line number  PRINT { {#<exp>,}<list>}                     5.4
             25  PRINT  !GENERATES  CR/LF                     6.3
             75  PRINT "BEGINNING OF OUTPUT";I,A*I            7.3
             45  PRINT #4, "OUTPUT TO DEVICE" FNM(A)↑2;B;A   10.4

PRINT USING                                                   9.8.1
    line number  PRINT {#<exp>,}USING <string>,<list>        10.4
             54  PRINT USING "##.##",A
             55  PRINT #3, USING"\\###.##  \\##↑↑↑↑", "A=",A,"B=",B

INPUT                                                         5.3
    line number  INPUT {#<exp>,}<list>                        3.10
             25  INPUT "TYPE YOUR NAME ", A$                  7.4
             55  INPUT #8, A, N, B$                          10.3

INPUT LINE                                                    9.9
    line number  INPUT LINE {#<exp>,} <string>               10.3
             4Ø  INPUT LINE  R$
             75  INPUT LINE  #1, E$


NAME-AS                                                       7.7
    line number  NAME <string> AS <string>{<protection>}
            455  NAME "NONAME" AS "FILE1"
            27Ø  NAME "DTA4:MATRIX" AS "MATA1"<48>

KILL                                                          7.8
    line number  KILL <string>
             45  KILL NONAME
```

```
ON ERROR GOTO                                                        9.4
      line number  ON ERROR GOTO <line number>
               1Ø  ON ERROR GOTO 5ØØ
              525  ON ERROR GOTO  !DISABLES ERROR ROUTINE
              526  ON ERROR GOTO Ø  !DISABLES ERROR ROUTINE

RESUME                                                               9.4
      line number  RESUME <line number>
             1ØØØ  RESUME      !OR RESUME Ø ARE EQUIVALENT
              655  RESUME  2ØØ

STOP                                                                 3.12
      line number  STOP
               75  STOP

END                                                                  3.12
      line number  END
              545  END
```

Matrix Statements

```
MAT READ                                                             8.1
      line number  MAT READ <list of matrices>
               55  DIM A(2Ø), B$(32), C%(15,10)
               9Ø  MAT READ A, B$(25), C%

MAT PRINT                                                            8.2
      line number  MAT RRINT{#<exp>,} <matrix name>
               1Ø  DIM A(2Ø), B(15,2Ø)
               9Ø  MAT PRINT A;          !PRINT 10*10 MATRIX, PACKED
               95  MAT PRINT B(1Ø,5)     !PRINT 10*5 MATRIX, FIVE
                                         !ELEMENTS PER LINE
               97  MAT PRINT #2, A;      !PRINT ON OUTPUT DEVICE 2

MAT INPUT                                                            8.3
      line number  MAT INPUT {#<exp>,} <list of matrices>
               1Ø  DIM B$(4Ø), F1%(35)
               2Ø  OPEN "DTA3:FOO" FOR INPUT AS FILE 3
               3Ø  MAT INPUT #3, B4, F1%

MAT initialization                       [ZER]                       8.4
      line number  MAT <matrix name> =   [CON] { dimension(s) }
                                         [IDN]
               1Ø  DIM B(15,1Ø), A(1Ø), C%(5)
               15  MAT C% = CON          !ALL ELEMENTS OF C%(I)=1
               2Ø  MAT B = IDN(1Ø,1Ø)    !IDENTITY MATRIX 1Ø*1Ø
               95  MAT A = ZER           !CLEARS MATRIX ELEMENTS TO Ø
```

CHAIN   l.n. CHAIN < string > { < line no. > }

Statement Modifiers

```
IF                                                                   9.7.1
     <statement>  IF <condition>
             1Ø   PRINT X IF X<>Ø

UNLESS                                                               9.7.2
     <statement>  UNLESS<condition>
             45   PRINT A UNLESS A=Ø

FOR                                                                  9.7.3
     <statement>  FOR <var> = <exp> TO <exp>{STEP<exp>}
             75   LET B$(I) = "PDP-11" FOR I = 1 TO 25
             8Ø   READ A(I) FOR I=2 TO 8 STEP 2

WHILE                                                                9.7.4
     <statement>  WHILE <condition>
             1Ø   LET A(I) = FNX(I) WHILE I<45.5

UNTIL                                                                9.7.5
     <statement>  UNTIL <condition>
            115   IF B>Ø THEN A(I)=B UNTIL I>5
```

APPENDIX B

BASIC-Plus Command Summary

Command	Explanation	Section
HELLO	Indicates to RSTS that a user wishes to log onto the system. Allows the user to input project-programmer number and password.	11.1.2
BYE	Indicates to RSTS that a user wishes to leave the terminal. Closes and saves any files remaining open for that user.	11.1.3
NEW	Clears the user's area in core and allows the user to input a new program from the terminal. A program name can be indicated following the word NEW or when the system requests it.	11.2
OLD	Clears the user's area in core 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 system disk.	11.3
DELETE	Allows the user to remove one or more lines from the program currently in core. Following the word DELETE the user types the line number of the single line to be deleted or two line numbers separated by a comma indicating the first and last line of the section of code to be removed. The word DELETE by itself deletes entire current program.	11.4.1
LIST	Allows the user to obtain a printed listing at the user terminal of the program currently in core, 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 comma will list the lines between and including the lines indicated.	11.4.2
LISTNH	Same as LIST, but does not print header containing the program name and current date.	11.4.2
CONT	Allows the user to continue execution of the program currently in core following the execution of a STOP statement.	11.4.3
RUN	Allows the user to begin execution of the program currently in core. The word RUN can be followed by a file name in which case the file is loaded from the system disk, compiled, and run; alternatively, the device and file name can be indicated if the file is not on the system disk. A device specification without a file name will cause a program to be read from an input only device (such as high-speed reader, card reader).	11.5.1

Command	Explanation	Section
RUNNH	Same as RUN, but does not print header containing the program name and current date.	11.5.1
SAVE	Causes the program currently in core to be saved on the system disk under its current file name with the extension .BAS. Where the word SAVE is followed by a file name or a device and a file name, the program in core is saved under the name given and on the device specified. A device specification without a file name will cause the program to be output to any output only device (line printer, high-speed punch).	11.5.2
UNSAVE	The word UNSAVE is followed by the file name and extension of the file to be removed. If no device is specified, the disk is assumed.	11.5.3
CHAIN	Allows the user to combine a segmented program in user core. Following the word CHAIN is the name of the next overlay section (assigned with the SAVE command) and, optionally, a line number on which to begin execution if the lowest line number is not to be used.	11.5.4
RENAME	Causes the name of the program currently in core to be changed to the name specified after the word RENAME.	11.5.5
REPLACE	Same as SAVE, but allows the user to substitute a new program with the same name for an old program, erasing the old program.	11.5.6
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.	11.5.7
LENGTH	Returns the length of the user's current program in core.	11.6.1
CATALOG	Returns the user's file directory. Unless another device is specified following the word CATALOG, the disk is the assumed device.	11.6.2
TAPE	Used to disable the echo feature on the user terminal while reading paper tape via low-speed reader.	11.7.1
KEY	Used to re-enable the echo feature on the user terminal following the issue of a TAPE command.	11.7.2
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 terminal which issued the ASSIGN.	11.7.3

Command	Explanation	Section
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.	11.7.4

Special Control Character Summary

RETURN Key	Enters a typed line to the system, results in a carriage return/line feed operation at the user terminal.	11.8.1
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.	11.8.2
LINE FEED Key	Used to continue the current logical line on an additional physical line. Performs a carriage return/line feed operation.	11.8.3
RUBOUT Key	Deletes the last character typed on that physical line. Erased characters are shown on the terminal between back slashes.	11.8.4
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.	11.8.5
CTRL/U	Deletes the current typed line, echoes as ↑U and performs a carriage return/line feed.	11.8.6
CTRL/O	Used as a switch to suppress/enable output of a program on the user terminal.	11.8.7
TAB or CTRL/I	Performs a tabulation to the next of nine tab stops (eight spaces apart) which form the terminal printing line.	11.8.8
CTRL/P	Used as a switch to enable/disable the software tabs created with a Model 33 Teletype terminal.	11.8.9
CTRL/B	Used as a switch to stop/start the echo feature at the user's terminal.	11.8.10
CTRL/Z	Used as an end-of-file character.	11.8.11

APPENDIX C

Error Message Summary

The following messages are printed when an error of the specified type occurs. These errors can cause a transfer to a user-written subroutine if the ON ERROR-GOTO statement has been used (in which case the message is not printed). Many of the following messages are followed by the phrase AT LINE XXX where XXX indicates the line number at which the error occurred. The value of the variable ERR following execution of the ON ERROR-GOTO transfer is given below.

ERR	Error Message	Meaning
1	BAD DIRECTORY FOR DEVICE	File lookup is impossible on the device specified.
2	ILLEGAL FILE NAME	The file name given contains embedded blanks or illegal characters.
3	FILE IS CURRENTLY OPEN	Another user has opened this file.
4	NO ROOM ON DEVICE	Sufficient storage space on device specified is not available to store the file.
5	CAN'T FIND FILE	File is not present on indicated device.
6	NOT A VALID DEVICE	Device indicated is not present in the system.
7	I/O CHANNEL ALREADY OPEN	A file is already open using that internal file designator.
8	DEVICE NOT AVAILABLE	Another user has assigned that device for his use.
9	I/O CHANNEL NOT OPEN	No file has been opened on the particular internal file designator used.
10	PROTECTION VIOLATION	This file is not available for your project-programmer code.
11	END OF FILE ON DEVICE	Program attempted to read beyond the end of the file.
12	OPERATION ABORTED	Serious I/O failure.
13	DATA ERROR ON DEVICE	Parity error detected.
14	DEVICE OK?	Device is off-line or requires service.

ERR	Error Message	Meaning
15	TELETYPE WAIT EXHAUSTED	User did not respond at the terminal within the allotted time.
16	FILE OF SAME NAME EXISTS	Attempt to put a second file of the same name on the same device.
17	VIRTUAL CORE NOT ON DISK	Attempt to reference virtual core on a device other than the system disk.
18	VIRTUAL CORE EXCEEDED	Not enough disk space left free to hold the virtual core area required.
19	VIRTUAL ARRAY NOT OPENED	File containing virtual core matrix has not been opened before being referenced.
20	ILLEGAL I/O CHANNEL	Acceptable I/O internal device designators are integers from 1 to 12.
21	LINE TOO LONG	More than 256 byte record read from an I/O device.
22	FLOATING POINT ERROR	Underflow or overflow has occurred.
23	ARGUMENT TOO LARGE IN EXP	Inaccurate results will follow.
24	ARGUMENT TOO LARGE IN SIN	Inaccurate results will follow.
25	INTEGER ERROR	Overflow has occurred.
26	ILLEGAL NUMBER	A non-numeric character was encountered in a number.
27	TRANSCENDENTAL ERROR	Attempt to take log of Ø or of a negative number.
28	IMAGINARY SQUARE ROOT	User requested square root of a negative number. (SQR(ABS(X)) returned.
29	SUBSCRIPT OUT OF RANGE	Undefined element of a matrix referenced.
30	OUT OF DATA	User has exhausted DATA statement pool.
31	ON-STATEMENT OUT OF RANGE	The expression in an ON-statement was less than 1 or greater than the number of line numbers specified.
32	NOT ENOUGH DATA IN RECORD	User tried to input more data than present on the next logical record of the file being used.
33	ILLEGAL UUO FOR USER	User has executed a SYS function illegally or with a bad argument.

The following messages are given when a fatal error has occurred; each is followed by the phrase AT LINE XXX where XXX is the line number where the error occurred. Execution stops after the printing of one of the following messages.

Error Message	Meaning
MAXIMUM CORE EXCEEDED	User program has overflowed one or more of the following: pushdown list, string storage, I/O buffer area, or matrix storage.
STATEMENT NOT FOUND	The line specified in a GOTO or GOSUB statement is not present.
ILLEGAL STATEMENT	An attempt was made to execute a statement that could not be compiled.
STOP	A STOP statement was the last statement executed, rather than the END statement. This message is not necessarily an error, but provides the user with the information on where program execution has stopped.
UNIMPLEMENTED CODE	The BASIC-Plus runtime interpreter cannot execute the compiled statement.
RETURN WITHOUT GOSUB	A RETURN statement was executed without a preceding GOSUB statement.
BAD NESTING OF GOSUB OR DEF	A DEF statement was encountered in the middle of a prior multi-line function definition.
UNDEFINED FUNCTION CALLED	A reference exists to a user function FNα, which has not been defined.

The following error messages are printed by the BASIC-Plus compiler while a program is being typed or commands are being given and allow the user to correct mistakes before a program is run. When the program is listed, lines containing these errors are preceded by a ? character.

Error Message	Meaning
ILLEGAL SYMBOL	An improper variable name was used.
ILLEGAL VERB	The word or term following the line number is not a recognizable BASIC statement designator.

Error Message	Meaning
ILLEGAL EXPRESSION	An expression was detected which does not conform to acceptable expressions in BASIC. See Section 2.5.
ILLEGAL MODE MIXING	Numbers (floating point or integer) cannot be combined in an expression with character strings.
ILLEGAL IF STATEMENT	Either the IF, THEN, or ELSE phrase is incorrect. See Sections 3.5 and 9.5.
ILLEGAL CONDITIONAL CLAUSE	The expression following IF in a conditional expression is not properly constructed.
ILLEGAL FUNCTION NAME	Function name used does not follow conventions established for user-defined functions.
ILLEGAL DUMMY VARIABLE	One or more dummy variables in a user-defined function are used elsewhere in that program.
ILLEGAL FN REDEFINITION	Attempt to redefine a user function. The first definition is retained and must be deleted before the second can be accepted.
ILLEGAL LINE NUMBER(S)	The line number used is out of the range for line numbers (1 n 32768) or the number is not an integer.
MODIFIER ERROR	Either the modifier phrase of the statement is incorrect or BASIC has interpreted some other error to be an incorrect attempt at a modifier phrase.
CAN'T COMPILE STATEMENT	The statement does not make sense to the BASIC compiler.
EXPRESSION TOO COMPLICATED	Rewrite the expression, using two or more statements.
ARGUMENTS DON'T MATCH	The number and/or type (numeric or string) of the arguments with which a function is called do not agree with the function definition.
TOO MANY ARGUMENTS	A maximum of eight arguments can be used with a user-defined function.
INCONSISTENT FUNCTION USAGE	Function is redefined with different number or kind of arguments. This message is a warning only, indicating that BASIC is aware of a change in the function.

Error Message	Meaning
ILLEGAL DEF NESTING	A DEF statement was detected within a multiple line function definition.
FOR WITHOUT NEXT	No NEXT statement can be found with a variable corresponding to the control variable in the FOR statement.
NEXT WITHOUT FOR	No FOR statement can be found with the control variable corresponding to the variable in the NEXT statement.
DEF WITHOUT FNEND	A multiple line function definition was encountered without a FNEND statement marking the end of the function.
FNEND WITHOUT DEF	An FNEND statement was found without a preceding DEF statement.
CONSTANT STRING NEEDED	A character string variable is not acceptable in this uage, a character string constant must be used.
TOO FEW ARGUMENTS	A device or file name was omitted in an ASSIGN or RENAME command.
SYNTAX ERROR	Elements within the statement have been arranged in the wrong sequence.
STRING IS NEEDED	A number was specified where a character string was expected.
NUMBER IS NEEDED	A character string was specified where a number was expected.
DATA TYPE ERROR	The type of term found in the DATA statement pool did not correspond to the type of variable about to be read by the current READ statement.
1 OR 2 DIMENSIONS ONLY	A matrix was specified with more than two subscripts.
PROGRAM LOST-SORRY	The user attempted to continue his program execution after receiving a MAXIMUM CORE EXCEEDED error.
RESUME AND NO ERROR	A RESUME command was given when no error condition existed. The CONT (continue) command should be used.
REDIMENSIONED ARRAY	A previously dimensioned matrix is being dimensioned again. The original dimension is maintained.
INCONSISTENT SUBSCRIPT USE	A matrix is being referenced with a different number of subscripts than it was defined to have.

Error Message	Meaning
ON-STATEMENT NEEDS GOTO	ON statement must be followed by a GOTO and a list of line numbers.
TEXT TRUNCATED	The user typed a program line with more than 255 characters.

The following errors are fatal and cause a termination of program execution, but the phrase AT LINE XXX is not printed.

Error Message	Meaning
WHAT?	BASIC-Plus did not understand the line entered.
NO END STATEMENT IN PROGRAM	When reading a file recalled by an OLD command, an end-of-file condition was detected before reading an END statement.
NOT ENOUGH CORE	A previously saved program is too large to run in core. See the CHAIN command section.
EXECUTE ONLY FILE	Not possible to LIST or SAVE a compiled program.
STATEMENT NOT FOUND	When chaining to a program, the starting line specified was not found.
PLEASE USE THE RUN COMMAND	Attempt to execute a statement in Immediate Mode which only makes logical sense in a program.
CAN'T CONTINUE	User changed program or execution was terminated by a fatal error.
END OF STATEMENT NOT SEEN	A badly formed statement has caused BASIC-Plus to miss detecting the end of that statement.
FILE EXISTS - USE 'REPLACE'	An attempt was made to SAVE a file which was already on the system disk. REPLACE will delete the old file and save the new file.
SWAP ERROR FOR JOB	A disk error occurred while moving the user's job to or from core. The user's program is lost.
PLEASE SAY HELLO	The user cannot use the RSTS-11 system until he has identified himself.
ILLEGAL IN IMMEDIATE MODE	This statement cannot be executed in Immediate Mode.
SORRY	System full; no add'l users can be accepted.

APPENDIX D

ASCII CHARACTER CODES

Character	ASCII Code No. (Decimal)	Character	ASCII Code No. (Decimal)
space	32	@	64
!	33	A	65
"	34	B	66
#	35	C	67
$	36	D	68
%	37	E	69
&	38	F	70
'	39	G	71
(	40	H	72
)	41	I	73
*	42	J	74
+	43	K	75
,	44	L	76
—	45	M	77
.	46	N	78
/	47	O	79
0	48	P	80
1	49	Q	81
2	50	R	82
3	51	S	83
4	52	T	84
5	53	U	85
6	54	V	86
7	55	W	87
8	56	X	88
9	57	Y	89
:	58	Z	90
;	59	[	91
<	60	\	92
=	61	]	93
>	62	↑	94
?	63	←	95

Additional symbols useful on output are as follows:

LF (line feed)	10
CR (carriage return)	13

The above list is not complete; there are 128 characters numbered 0 through 127.

ctrl char - letter code - 64

INDEX

HOW TO OBTAIN SOFTWARE INFORMATION

Announcements for new and revised software, as well as programming notes, software problems, and documentation corrections are published by Software Information Service in the following newsletters.

Digital Software News for the PDP-8 & PDP-l2
Digital Software News for the PDP-ll
Digital Software News for the PDP-9/l5 Family

These newsletters contain information applicable to software available from Digital's Program Library, Articles in Digital Software News update the cumulative Software Performance Summary which is contained in each basic kit of system software for new computers. To assure that the monthly Digital Software News is sent to the appropriate software contact at your installation, please check with the Software Specialist or Sales Engineer at your nearest Digital office.

Questions or problems concerning Digital's Software should be reported to the Software Specialist. In cases where no Software Specialist is available, please send a Software Performance Report form with details of the problem to:

Software Information Service
Digital Equipment Corporation
l46 Main Street, Bldg. 3-5
Maynard, Massachusetts 0l754

These forms which are provided in the software kit should be fully filled out and accompanied by teletype output as well as listings or tapes of the user program to facilitate a complete investigation. An answer will be sent to the individual and appropriate topics of general interest will be printed in the newsletter.

Orders for new and revised software and manuals, additional Software Performance Report forms, and software price lists should be directed to the nearest Digital Field office or representative. U.S.A. customers may order directly from the Program Library in Maynard. When ordering, include the code number and a brief description of the software requested.

Digital Equipment Computer Users Society (DECUS) maintains a user library and publishes a catalog of programs as well as the DECUSCOPE magazine for its members and non-members who request it. For further information please write to:

DECUS
Digital Equipment Corporation
l46 Main Street, Bldg. 3-5
Maynard, Massachusetts 0l754

READER'S COMMENTS

Digital Equipment Corporation maintains a continuous effort to improve the quality and usefulness of its publications. To do this effectively we need user feedback -- your critical evaluation of this manual.

Please comment on this manual's completeness, accuracy, organization, usability and readability.

____

____

____

____

Did you find errors in this manual? If so, specify by page.

____

____

____

____

____

How can this manual be improved?

____

____

____

____

____

Other comments?

____

____

____

____

____

Please state your position. ____________________ Date: __________

Name: ____________________ Organization: ____________________

Street: ____________________ Department: ____________________

City: ______________ State: ______________ Zip or Country__________

— — — — — — — — — — — — — — — — Fold Here —

— — — — — — — — — — — — — — Do Not Tear - Fold Here and Staple — — — — — — — — — — — — —

digital

Digital Equipment Corporation
Software Information Services
146 Main Street, Bldg. 3-5
Maynard, Massachusetts 01754