pdp11

BASIC-PLUS LANGUAGE MANUAL

DEC-11-ORBPB-A-D

digital

BASIC-PLUS LANGUAGE MANUAL

DEC-11-ORBPB-A-D

July, 1975

digital equipment corporation · maynard. massachusetts

CONTENTS

CONTENTS (Cont.)

CONTENTS (Cont.)

CONTENTS (Cont.)

CONTENTS (Cont.)

CONTENTS (Cont.)

CONTENTS (Cont.)

FIGURES

PREFACE

This manual describes the BASIC-PLUS programming language. Information 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.

While it is always good programming practice to use the % character to indicate integer format of variables, as described in Chapter 6, the sample programs in this manual do not always follow this convention. For the sake of clarity in illustrating various programming concepts, the % character is omitted occasionally in these examples, but should be included in user programs to save storage space as well as computing time.

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

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

First Printing, July 1975

CDP	DIGITAL	INDAC	PS/8
COMPUTER LAB	DNC	KA10	QUICKPOINT
COMSYST	EDGRIN	LAB-8	RAD-8
COMTEX	EDUSYSTEM	LAB-8/e	RSTS
DDT	FLIP CHIP	LAB-K	RSX
DEC	FOCAL	OMNIBUS	RTM
DECCOMM	GLC-8	OS/8	RT-11
DECTAPE	IDAC	PDP	SABR
DIBOL	IDACS	PHA	TYPESET 8
			UNIBUS

9/76-14

PART I

RSTS/E AND THE BASIC-PLUS LANGUAGE

This first Part describes the RSTS/E system, its hardware and user features, and the simplest level of the BASIC language. BASIC as described here 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. Part III describes the complete range of BASIC-PLUS I/O, including Record I/O and information on particular I/O devices.

As a language, BASIC is easy to learn. BASIC-PLUS provides many advanced features which allow BASIC to be a useful tool for the more experienced 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 Parts II and III allow the user to write more efficient code to better use machine time and core space.

CHAPTER 1

AN INTRODUCTION TO RSTS-11

In this manual, the RSTS-11 user need only be concerned with the writing and execution of correct programs in the BASIC-PLUS language. A description of the various RSTS-11 commands (NEW, OLD, LIST, RUN, etc.) can be found in the RSTS-11 System User's Guide.

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 RSTS-11. 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 from 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 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 INTRODUCTION TO TIME-SHARING

RSTS-11 is a time-sharing system. This means that when a user is working with RSTS, he has the illusion that he is the only user on the computer.

Many users can be on-line to RSTS at one time because RSTS controls the scheduling of execution times. RSTS has one or more users in core at one time. Users are brought into core from disk, allowed to execute for a short time, and returned to disk. This process is called swapping. RSTS takes note of the state at which execution stops and is able to resume operation at that point.

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

1.3 THE BASIC-PLUS PROGRAMMING LANGUAGE

BASIC is one of the simplest of all programming languages because of the small number of powerful but easily understood statements and commands and its easy application to problem solving. The wide use of BASIC in scientific, business, and educational installations attests to its value and straightforward application. (For a bibliography of texts on BASIC and other elementary computing texts, see Appendix G.)

BASIC is similar to many other programming languages in various respects 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, providing for an interactive man/machine relationship.

[1] The term "K" refers to 1Ø24 (decimal) words of storage in a computer Hence, 2K=2Ø48 words and 8K=8192 words.

BASIC-PLUS contains both the elementary statements used to write simple programs and many new advanced programming features and statements to produce more complex and efficient programs. The key word here is efficient. As the user progresses and gains programming experience, he will naturally find himself becoming more efficient and able to use 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.

1.4 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. Items in italic type (*formula*, *variable*, etc.) are supplied by the user according to rules explained in the text. Items in capital letters (LET, IF, THEN, etc.) must appear exactly as shown because they form the vocabulary of the BASIC language.

b. The term *line number* used in examples indicates that any line number is valid.

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

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

d. Square brackets indicate a choice of one element among two or more possibilities. For example:

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

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

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

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 <u>prints</u> on the teleprinter whereas the user <u>types</u> on the keyboard.

b. A statement is a single BASIC language instruction. Each BASIC program line is preceded by a line number and terminated by the RETURN key. A program line may contain a single statement or several statements separated by colons (see Section 2.3.1).

c. Commands cause BASIC to perform some operation immediately and are not preceded by a line number. A command is terminated by typing the RETURN key.

d. A user program consists of a series of statements written by a person using the BASIC-PLUS language.

e. The RSTS-11 terminal is in some cases an ASR-33 Teletype[1]. However, RSTS-11 can accommodate a wide variety of other terminals such as a DECwriter or VTØ5 display. The RSTS-11 user terminal is alternatively referred to as terminal, teleprinter, or keyboard, depending upon whether a part or the whole device is indicated. The use of terminals and other peripheral devices is described in the RSTS-11 System User's Guide.

f. The term BASIC is used interchangeably to indicate the BASIC language and the BASIC Interpreter (the system program which accepts and executes BASIC programs).

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

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[1] 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 RUNNH) 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 a word specifying the type of operation to be performed.

2.2 LINE NUMBERS

Each BASIC program line 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.

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

[1]The syntax of a language is the collection of rules governing the combination of language elements.

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

READY

RUNNH
INTEREST IN PERCENT? 7.5
AMOUNT OF LOAN? 2500
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        2207.44
 41.39         298.05        1909.39
 35.8          303.64        1605.75
 30.11         309.33        1296.42
 24.31         315.13        981.28
 18.4          321.04        660.24
 12.38         327.06        333.18
 6.25          333.19
LAST PAYMENT = 339.43

READY
```

Figure 2-1

Example BASIC Program

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

2.3 STATEMENTS

Each line number is followed by a BASIC statement. The first word of a BASIC statement identifies the type of statement and informs BASIC of the operation to be performed and 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 or a backslash. 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=X+1: PRINT X,Y,Z\ IF Y=2 GOTO 10
```

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

Any statement can be used anywhere in a multiple-statement line except as noted in the discussion of the individual statements.

2.3.2 A Single Statement on Multiple Lines

A single statement can be continued on successive lines 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:

```
10 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.

Where the LINE FEED key is used, it must occur between the elements of a BASIC statement. That is, a BASIC verb or the designation of a subscripted array element (see Section 3.6.2), for example, cannot be broken with a LINE FEED.

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

is acceptable where a LINE FEED follows Ø, but:

```
10 IF A
1=0 THEN 100
ILLEGAL CONDITIONAL CLAUSE
```

is not acceptable nor is:

```
10 IF A1=0 THEN 1
00
MODIFIER ERROR AT LINE 10
```

and each illegal form generates an error message. A number of multiword elements are processed as one word and cannot be broken by a LINE FEED. For example, AS FILE, FOR INPUT AS FILE, FOR OUTPUT AS FILE, GO TO, INPUT LINE, and ON ERROR GO TO are each treated by the system as one word.

2.4 SPACES AND TABS

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
```

or

```
10   L  ETB  = D * 2 + 1
```

The above statements are identical in effect.

TABS, like spaces, are used to make a program easy to read. An example follows:

```
10 FOR K=1 TO 3
20      FOR I=1 TO 10
30              FOR J=1 TO 10
40              A(I,J) = K/(I+J-1)+A(I,J)
50              NEXT J
60      NEXT I
70 NEXT K
```

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, relational, or logical operators.

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

Arithmetic Expressions	Logical Expressions
4	X<Y
A7*(B↑2+1)	((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. Appendix F explains the integer and floating-point number formats. Numeric constants are written using decimal notation, as follows:

```
+2
-3.675
1234.56
-123456
-.ØØØØØ1
```

The following are not acceptable numeric constants 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 in later sections.

Scientific notation allows further flexibility in number representation. Numeric constants can be written using the letter E to indicate "times ten to the power," thus:

.ØØØ123456	can be written in BASIC as 123.456E-6
123456ØØØØ.	can be written in BASIC as 123456E4
-123456789ØØ.	can be written in BASIC as -1.2345679E1Ø

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

The range of floating-point numbers is (approximately) as follows:

X=Ø or in the range $10^{-38} < \text{ABS(X)} < 10^{+38}$

2.5.2 Variables

A variable is a data item whose value can be changed by the program. 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. It is only necessary to assign a value to a variable when an initial value other than zero is required. However, good programming practice would be to set variables equal to Ø wherever necessary. This ensures that later changes or additions will not misinterpret values.

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:

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

BASIC-PLUS permits the operator ** in place of ↑ to denote the exponentiation operation. For example:

```
A**B
```

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

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

When more than one operation is to be performed in a single formula, 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:

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

```
(A+(B*(D↑2)))
```

the innermost parenthetical quantity is calculated first.

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

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

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

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 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 expressions easier to read. Ambiguities can 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); in conditional FOR loops (see Section 8.6); and in IF, UNLESS, WHILE and UNTIL clauses (see Sections 3.5, 8.5, and 8.7) where it is necessary to compare values. The relational symbols are as follows (where A and B are variables or expressions):

Mathematical Symbol	BASIC Symbol	Example	Meaning
$=$	=	A=B	A is equal to 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.

The term "approximately equal to" means that the two quantities look the same when printed. 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. This last decimal place within the computer does not always cause two numbers to have a different value when printed. Numbers are carried internally at greater than 6 digits of precision, but are rounded to 6 digits for output or a $\approx$ comparison. Thus, two numbers identical when rounded to 6 digits of precision are approximately equal, whereas two numbers equal to the internally carried limits of precision are truly equal (=).

2.5.5 Logical Operators

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

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

IMP	A IMP B	The logical implication of A and B. A IMP B is false if and only if A is true and B is false; otherwise the value is true.
EQV	A EQV B	A is logically equivalent to B. A EQV B has the value TRUE if A and B are both true or both false, and has the value false otherwise.

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

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

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

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

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

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

A	NOT A
T	F
F	T

CHAPTER 3

ELEMENTARY BASIC 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 and features explained in Parts II and III.

The reader should understand that any problem which can be solved with the more advanced techniques can also be solved with the simpler statements, although the solution may not be as efficient. As long as the user understands the details of his problem he can 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 word REMARK can be abbreviated to REM for typing convenience, and the message itself can contain any printing characters 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 program execution.) Typical REM statements are shown below:

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

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

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

          140 !COMMENT
```

In every statement other than the DATA statement, BASIC ignores everything on the line following the exclamation mark. An exclamation mark must not appear on the same line as a DATA statement unless it is part of an item in the DATA statement. (Tabs are useful for inserting space between the statement and comment parts of a line to improve readability.)

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 10 and 11 are REM statements. Line 20 is one REM statement broken into two lines with the LINE FEED key. Line 30 is one comment (begun with a !) and broken into two lines with the LINE FEED key.

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

          20 REM THIS PROGRAM COMPUTES THE
          ROOTS OF A QUADRATIC EQUATION

          30 ! 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:

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

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

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

```
50 LET A=35
```

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

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

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

than

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

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

The LET statement can be used anywhere in a multiple statement line, for example:

```
1Ø X=44: Y=X↑2+Y1: B2=3.5*A
```

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

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

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

3.3 PROGRAMMED INPUT AND OUTPUT

This Section describes the techniques used in performing BASIC program I/O (an abbreviation for the term Input/Output which includes the processes by which data is brought into and sent out of the computer). The most elementary forms of the PRINT, INPUT, READ, and DATA statements are presented here so that the user is able to create simple BASIC programs.

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

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

3.3.1 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 *<variable list>*

A DATA statement is of the form:

line number DATA *<value list>*

A READ statement causes the variables listed to be assigned sequential values 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. Each time a READ statement is encountered in the program, the data block supplies the next value. If the data block runs out of data, the program is assumed to be finished and an OUT OF DATA message is printed by BASIC.

READ and DATA statements appear as follows:

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

Note that only numbers are used in this particular DATA statement. (Input of string data is treated in Section 5.3.) The assignments performed by line 15Ø are as follows:

```
 X=4
 Y=2
 Z=1.7
X1=6.73E-3
Y2=-174.321
Q9=3.1415927
```

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 or the last statement on a line, while a READ statement can be placed anywhere in a multiple statement line.

NOTE

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

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

line number RESTORE

For example:

```
30 RESTORE
```

causes the next READ statement following line 3Ø to begin reading data from the first DATA statement in the program, regardless of where the last data value was found.

The same variable names can be used the second time through the data or not, as is most convenient, since the values are being read as though for the first time. In order to skip unwanted values, dummy variables must be read. In the following example, BASIC 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 45.

```
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
```

3.3.2 PRINT Statement

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

line number PRINT {*list*}

where the list can contain expressions, text strings, or both. As the 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 results. Any expression within the list is evaluated before a value is printed. Consider the following program:

```
LISTNH
10 LET A=1: LET B=2: LET C=3+A
20 PRINT
30 PRINT A+B+C

READY

RUNNH

 7

READY
```

All numbers are printed in the form:

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

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

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

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

```
RUNNH
 1

 6

READY
```

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

BASIC considers the terminal printer to be divided into five zones of fourteen spaces each[1]. When an item in a PRINT statement is followed by a comma, the next value to be printed appears in the next available print zone. For example:

```
10 LET A=3: LET B=2
20 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, since a 72-character line has five print zones.

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

```
LISTNH
10 LET A=1: LET B=2
20 PRINT A,B,,A+B

READY

RUNNH
 1             2                           3

READY
```

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

```
LISTNH
10 A=1:B=2:C=3
20 PRINT A,:PRINT B: PRINT C

READY

RUNNH
 1             2
 3

READY
```

[1]Terminals with greater than 83 columns have additional print zones in units of fourteen spaces.

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

```
LISTNH
10 LET A=1: B=2: C=3
20 PRINT A;B;C;
30 PRINT A+1;B+1;C+1
40 PRINT A,B,C

READY

RUNNH
 1  2  3  2  3  4
 1             2              3

READY
```

The PRINT statement can be used to print a message, either alone or together with the evaluation and printing of numeric values. Characters are 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
10 PRINT "TIME'S UP"
20 PRINT '"NEVERMORE"'

READY

RUNNH
TIME'S UP
"NEVERMORE"

READY
```

As another example, consider the following line:

```
40 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; spaces appear in the printout exactly as they are typed within the quotation marks.

When a comma separates a text string from another PRINT list item, the item is printed at the beginning of the next available print zone. Semicolons separating text strings from other items are ignored. Thus, the previous example could be expressed as:

```
40 PRINT "AVERAGE GRADE IS" X
```

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

The following example demonstrates the use of the formatting characters, and ; with text strings:

```
120 PRINT "STUDENT NUMBER"X,"GRADE ="G;"AVE. ="A;
130 PRINT "NO. IN CLASS ="N
```

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

```
STUDENT NUMBER 119050       GRADE = 87 AVE. = 85.44 NO. IN CLASS = 26
```

3.3.3 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 Part III.) Depending upon how many values are to be accepted by the INPUT command, the programmer may wish to send himself a message reminding him what data is to be typed at what time (this can be done with the PRINT or INPUT statement).

The INPUT statement is of the form:

line number INPUT <*list*>

For example:

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

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

In the example program following, four questions are asked at execution time: INTEREST IN PERCENT?, AMOUNT OF LOAN?, NUMBER OF YEARS?, and NO. OF PAYMENTS PER YEAR?. The programmer knows which value is requested and proceeds to type and enter the appropriate value.

```
LISTNH
10 REM PROGRAM TO COMPUTE INTEREST PAYMENTS
15 INPUT "INTEREST IN PERCENT"; J
20 LET J=J/100
25 INPUT "AMOUNT OF LOAN"; A
30 INPUT "NUMBER OF YEARS"; N
35 INPUT "NO. OF PAYMENTS PER YEAR"; M
40 N=N*M: I=J/M: B=1+I: R=A*I/(1-1/B↑N)
45 PRINT: PRINT "AMOUNT PER PAYMENT =";R
50 PRINT "TOTAL INTEREST     =";R*N-A
55 PRINT: B=A
60 PRINT "INTEREST    APP TO PRIN    BALANCE OF PRIN"
65 L=B*I: P=R-L: B=B-P
67 PRINT L,P,B
70 IF B>=R GOTO 65
75 PRINT B*I,R-B*I
80 PRINT "LAST PAYMENT WAS "B*I+B
85 END

READY

RUNNH
INTEREST IN PERCENT? 9
AMOUNT OF LOAN? 2500
NUMBER OF YEARS? 2
NO. OF PAYMENTS PER YEAR? 4

AMOUNT PER PAYMENT = 344.961
TOTAL INTEREST     = 259.688

INTEREST    APP TO PRIN    BALANCE OF PRIN
 56.25         288.711       2211.29
 49.754        295.207       1916.08
 43.1119       301.849       1614.23
 36.3202       308.641       1305.59
 29.3758       315.585       990.007
 22.2752       322.686       667.321
 15.0147       329.946       337.375
 7.59093       337.37
LAST PAYMENT WAS  344.966

READY
```

As in the previous program, the question mark generated by BASIC 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 RUNNH and includes a verbal description of the numbers. This verbal description on the output is optional with the programmer, although it has a definite advantage in ease of use and understanding.

When the correct number of variables 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 single or 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
```

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

3.4 UNCONDITIONAL BRANCH, GOTO STATEMENT

The GOTO statement 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 thus possible to jump forward or 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 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 should always be the last statement on the line, since any statement following the GOTO on the same line is never executed. For example:

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

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:

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

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

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

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

Both types of condition, when evaluated, are either true or false; no numeric value is associated with the results of an IF statement. The relational and logical operators are described in Sections 2.5.4 and 2.5.5 and are presented in Appendix A for reference.

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

In the above line the quantities A*B and B*(B+l) 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+l) is greater than A*B, D4 is not incremented and control passes immediately to the next line following line 75.

When a line number follows the word THEN, the IF-THEN statement is the same as the IF-GOTO statement. The word THEN can 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 value 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.)

```
LISTNH
10 LET A=A+1: X=A↑2
20 IF A>4 GOTO 55
25 PRINT X
30 PRINT "VALUE OF A IS" A
40 GOTO 10
55 END

READY
```

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

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

READY
```

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

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

a. expressions in parentheses
b. intrinsic or user-defined functions
c. exponentiation (↑)
d. 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         !ASSIGNMENT BY A LET STATEMENT
40 IF X>Y IMP Y>Z THEN PRINT "QED"
```

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

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

```
10 IF A=1 THEN PRINT A;:PRINT "TRUE CASE": GOTO 20
15 PRINT "NOT = 1"
```

where A≠1, the following line is printed:

```
NOT =1
```

where A=1, the following line is printed:

```
1 TRUE CASE
```

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

```
20 IF A>B THEN IF B>C THEN PRINT "B>C": GOTO 30
25 PRINT "A<=B"
```

Only in the case where "B>C" is printed is the statement GOTO 3Ø seen and executed.

3.6 PROGRAM LOOPS

Loops were first mentioned 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 in a minimal amount of time. To write a loop, the programmer must ensure that the series of statements is repeated until a terminal condition is met.

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

```
10 PRINT 1, SQR(1)
20 PRINT 2, SQR(2)
30 PRINT 3, SQR(3)
        .
        .
        .
990 PRINT 99, SQR(99)
1000 PRINT 100, SQR(100)
1010 END
```

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

```
10 LET X=1
20 PRINT X,SQR(X)
30 LET X=X+1
40 IF X<=100 THEN 20
50 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. The condition in line 40 is now false so control 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 (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. termination condition, an exit test which, when satisfied, completes the loop (line 40 above). Execution continues to the program statements following the loop (line 50 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 passes to the line following the associated NEXT statement. The variable in the FOR statement, K in the preceding example, is known as the control variable.

The control variable must be unsubscripted, although a common use of such loops is to deal with subscripted variables using the control variable as the subscript of a previously defined variable (this is explained in further detail in Section 3.6.2). The expressions in the FOR statement can be any acceptable BASIC expression as 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 expression value to the variable and checks to see if the variable is still less than or equal to the terminal expression value. When the variable exceeds the terminal expression value, control falls through the loop to the statement following the NEXT statement.

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

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

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

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

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

```
10 FOR I = 10 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 is only executed once since the value of I=44 has been reached and the termination condition is satisfied.

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

```
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.

ACCEPTABLE NESTING TECHNIQUES

Two Level Nesting

```
FOR I1 = 1 TO 1Ø
FOR I2 = 1 TO 1Ø
NEXT I2
FOR I3 = 1 TO 1Ø
NEXT I3
NEXT I1
```

Three Level Nesting

```
FOR I1 = 1 TO 1Ø
FOR I2 = 1 TO 1Ø
FOR I3 = 1 TO 1Ø
NEXT I3
FOR I4 = 1 TO 1Ø
NEXT I4
NEXT I2
NEXT I1
```

UNACCEPTABLE NESTING TECHNIQUES

```
FOR I1 = 1 TO 1Ø
FOR I2 = 1 TO 1Ø
NEXT I1
NEXT I2
```

```
FOR I1 = 1 TO 1Ø
FOR I2 = 1 TO 1Ø
FOR I3 = 1 TO 1Ø
NEXT I3
FOR I4 = 1 TO 1Ø
NEXT I4
NEXT I1
NEXT I2
```

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

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

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

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

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

```
10 FOR I=1 TO 10 STEP 5:  NEXT I: PRINT "I=";I
```

causes:

```
I= 6
```

to be printed when executed.

Neither the 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
```

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.

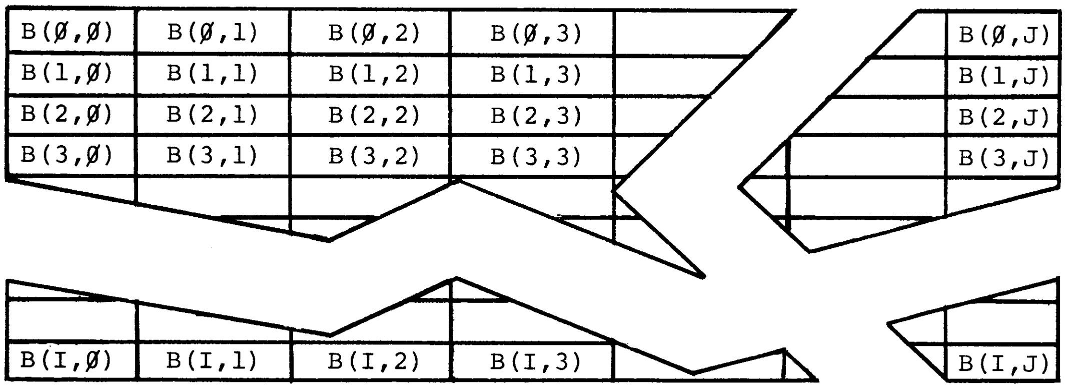
The name of a subscripted variable is any acceptable BASIC variable name followed by one or two integer expressions in parentheses. For example, a list might be described as A(I) where I goes from 1 to 5 as shown below (all matrices are created with a zero element, even though that element is never specified):

A(Ø), A(1), A(2), A(3), A(4), A(5)

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

A(Ø)
A(1)
A(2)
A(3)
A(4)
A(5)

A two dimensional matrix B(I,J) can be defined in a similar manner and graphically illustrated as follows:



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

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

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

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

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

For example:

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

Only integer values (such as 5 or 5070) can be used in DIM statements to define the size of a matrix. Any number of matrices can be defined in a single DIM statement as long as their representations are separated by commas.

If a subscripted variable is used without appearing in a DIM statement, it is assumed to be dimensioned to length 10 in each dimension (that is, having eleven elements in each dimension, Ø through 1Ø). However, all matrices should be correctly dimensioned in a program. DIM statements are usually grouped together among the first lines of a program.

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

```
LISTNH
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 he may make use of the extra variables set up within the matrix. He could, for example, say DIM A(5,9) to obtain a 6 x 10 matrix which would then be referenced beginning with the A(Ø,Ø) element.

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

Additional information on matrices can be found in Chapter 7.

A DIM statement can be placed anywhere in a multiple statement line. A DIM statement can appear anywhere in the program and need not appear prior to the first reference to an array, although DIM statements are generally among the first statements of a program to allow them to be easily found if any alterations are later required.

3.7 MATHEMATICAL FUNCTIONS

Within the course of a user's programming experience, he encounters many cases where relatively common mathematical operations are performed. The results of these common operations can often be found in volumes of mathematical tables; i.e., sine, cosine, square root, log, etc. Since it is this sort of operation that computers perform with speed and accuracy, such operations are built into BASIC. 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, intrinsic functions, such as:

```
SIN(23*PI/18Ø)
LOG(144)
```

are substituted.

The various mathematical functions available in BASIC-PLUS are detailed in Table 3.1.

Table 3-1
Mathematical Functions

Function Code	Meaning
ABS(X)	returns the absolute value of X
SGN(X)	returns the sign function of X, a value of 1 preceded by the sign of X, SGN(Ø)=Ø
INT(X)	returns the greatest integer in X which is less than or equal to X, (INT(-.5)=-1)
FIX(X)	returns the truncated value of X, SGN(X)*INT(ABS(X)),(FIX(-.5)=Ø)
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(X)	returns the arctangent (in radians) of X
SQR(X)	returns the square root of X
EXP(X)	returns the value of e↑X, where e=2.71828...
LOG(X)	returns the natural logarithm of X, $\log_e X$
LOG1Ø(X)	returns the common logarithm of X, $\log_{1Ø} X$
PI	has a constant value of 3.1415927
RND(X)	returns a random number between Ø and 1; the same sequence of random numbers is generated each time a program is run requiring the use of the random number generator. The value of X is ignored.
RND	alternate form for calling the random number function.

Most of these functions are self-explanatory. Those which are not 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, Ø if X is 0, and -1 if X is negative. For example: SGN (3.42) = 1, SGN (-42) = -1, and SGN(23-23) = Ø.

```
LISTNH
10 REM - SGN FUNCTION EXAMPLE
20 READ A,B
25 PRINT "A:"A,"B:"B
30 PRINT "SGN(A):"SGN(A),"SGN(B):"SGN(B)
40 PRINT "SGN(INT(A)):"SGN(INT(A))
50 DATA -7.32, .44
60 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*1Ø↑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.

NOTE

↑C in the above program terminates program execution. See the RSTS-11 System User's Guide.

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.

```
LISTNH
10 REM - RANDOM NUMBER EXAMPLE
25 PRINT "RANDOM NUMBERS"
30 FOR I=1 TO 30
40 PRINT RND(0),
50 NEXT I
60 END

READY

RUNNH
RANDOM NUMBERS
 .771027        .78183         .75174         .473969        .781555E-1
 .203217        .5159          .266449        .955597        .335541
 .412872        .457367        .283508E-1     .538025E-1     .676575E-1
 .921722        .921417        .233002        .105255        .534515
 .259796        .748138        .150665        .170746        .668488
 .474213        .828888        .705414        .772491        .286224

READY
```

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

```
40 PRINT INT(10*RND(0)),
```

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

```
RUNNH
RANDOM NUMBERS
 7              7              7              4              0
 2              5              2              9              3
 4              4              0              0              0
 9              9              2              1              5
 2              7              1              1              6
 4              8              7              7              2

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.

Since the parameter X in RND(X) is ignored, there is an alternate means of calling the random number generator having no arguments: RND. The following line is, therefore, acceptable:

```
40 PRINT RND
```

Similarly, if a number in the range (A,B) is desired, the formula:

```
(B-A)*RND+A
```

can be used.

3.7.2 RANDOMIZE Statement

The RANDOMIZE statement is written as follows:

line number RANDOMIZE

or, alternatively:

line number RANDOM

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

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

```
LISTNH
15 RANDOMIZE
20 FOR I=1 TO 5
25 PRINT "VALUE" I " IS" RND(0)
30 NEXT I
35 END

READY

RUNNH
VALUE 1  IS .797943
VALUE 2  IS .300079
VALUE 3  IS .618988
VALUE 4  IS .132141E-1
VALUE 5  IS .508392

READY
```

```
RUNNH
VALUE 1  IS .273041
VALUE 2  IS .225372
VALUE 3  IS .894867
VALUE 4  IS .340851
VALUE 5  IS .991303

READY
```

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. For example:

```
FNA
FNA1
```

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

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

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

```
10  DEF FNA(S) = S↑2
```

causes a later statement:

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

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

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

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

The two following programs

<u>Program #1</u>:

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

READY
```

<u>Program #2</u>:

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

READY
```

cause the same output:

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

READY
```

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:

```
10 DEF FNA(X) = X^2+3*X+4
20 DEF FNB(X) = FNA(X)/2 + FNA(X)
30 DEF FNC(X) = SQR(X+4)+1
```

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

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

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

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

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

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

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

```
LISTNH
1    ! MODULUS ARITHMETIC PROGRAM
5    ! FIND X MOD M
10   DEF FNM(X,M) = X-M*INT(X/M)
15   !
20   ! FIND A+B MOD M
25   DEF FNA(A,B,M) = FNM(A+B,M)
30   !
35   ! FIND A*B MOD M
40   DEF FNB(A,B,M) = FNM(A*B,M)
41   !
45   PRINT
50   PRINT "ADDITION AND MULTIPLICATION TABLES, MOD M"
55   INPUT "GIVE ME AN M";M
60   PRINT: PRINT "ADDITION TABLES MOD "M
65   GOSUB 800
70   FOR I=0 TO M-1
75   PRINT I;"  ";
80  FOR J=0 TO M-1
85   PRINT FNA(I,J,M);
90   NEXT J: PRINT: NEXT I
100  PRINT: PRINT
110  PRINT "MULTIPLICATION TABLES MOD " M
120  GOSUB 800
130  FOR I=0 TO M-1
140  PRINT I;"  ";
150  FOR J=0 TO M-1
160  PRINT FNB(I,J,M);
170  NEXT J: PRINT: NEXT I
180  STOP
800  !SUBROUTINE FOLLOWS:
810  PRINT: PRINT TAB(4);0;
820  FOR I=1 TO M-1
830  PRINT I;: NEXT I: PRINT
840  FOR I=1 TO 2*M+3
850  PRINT "-";: NEXT I: PRINT
860  RETURN
870  END

READY
```

Figure 3-1

Modulus Arithmetic

```
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  6  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 (Cont.)

Modulus Arithmetic

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

```
10 DEF FNA (X) = X*2 + X/2
20 PRINT FNA(3,2)
```

will cause an error message:

```
ARGUMENTS DON'T MATCH AT LINE 20
```

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

```
10 DEF FNA = X^2
50 R1 = FNB()
```

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

```
10 DEF FNZ(X)=X^2
20 LET A=2
30 PRINT FNZ(2+A)
```

line 30 causes 16 to be printed.

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

```
10 DEF FNX(X)=X^2
20DEF FNX(X)=X+X
ILLEGAL FN REDEFINITION AT LINE 20
```

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

```
10 DEF FNA (X) = 4 +2
20 LET R = FNA(10)+1
30 PRINT R
40 END
RUNNH
 7
```

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

3.8 SUBROUTINES

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

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

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

```
LISTNH
1   REM - THIS PROGRAM ILLUSTRATES GOSUB AND RETURN
10  DEF FNA(X)= ABS(INT(X))
20  INPUT A,B,C
30  GOSUB 100
40  LET A=FNA(A)
50  LET B=FNA(B)
60  LET C=FNA(C)
70  PRINT
80  GOSUB 100
90  STOP
100 REM - THIS SUBROUTINE PRINTS OUT THE SOLUTIONS
110 REM - OF THE EQUATION:  AX↑2 + BX + C = 0
120 PRINT "THE EQUATION IS   " A "*X↑2  + " B "*X  + " C
130 LET D=B*B - 4*A*C
140 IF D<>0 THEN 170
150 PRINT "ONLY ONE SOLUTION... X "; -B/(2*A)
160 RETURN
170 IF D<0 THEN 200
180 PRINT "TWO SOLUTIONS...X =";
185 PRINT (-B+SQR(D))/(2*A); "AND X ="; (-B-SQR(D))/(2*A)
190 RETURN
200 PRINT "IMAGINARY SOLUTIONS... X = (";
205 PRINT -B/(2*A) "," SQR(-D)/(2*A) ") AND  (";
207 PRINT -B/(2*A) ","; -SQR(-D)/(2*A) ")"
210 RETURN
900 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 )
STOP AT LINE 9Ø

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.

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

```
5Ø GOSUB 2ØØ
```

Control is transferred to line 2ØØ in the user program. The first line in the subroutine can be a remark or any executable statement.

3.8.2 RETURN Statement

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

line number RETURN

which causes control to return to the statement following the original GOSUB statement. A subroutine is always exited via a RETURN statement.

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

3.8.3 Nesting Subroutines

Subroutines can be nested; that is, one subroutine can call another subroutine. If the execution of a subroutine encounters a RETURN statement, it returns control to the line following the GOSUB which called that subroutine. Therefore, a subroutine can call another subroutine, even itself. Subroutines can be entered at any point and can have more than one RETURN statement. It is possible to transfer to the beginning or 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 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 STOP AND END STATEMENTS

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

line number END

The line number of the END statement should be the largest line number in the program, since any lines having line numbers greater than that of the END statement are not executed and are not retrieved by the OLD command (although they are saved with the SAVE command).

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 causes:

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

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

Execution of a STOP or END statement causes the message:

```
READY
```

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

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, and intrinsic matrix functions. Also described is the immediate mode of operation which causes BASIC to treat single statements as commands.

In general, the new techniques presented here allow the user to write programs which conserve core space and reduce execution time. With the ability to manipulate character strings, the user can write sophisticated programs to handle a wide range of data.

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

CHAPTER 4

IMMEDIATE MODE OPERATIONS

4.1 USE OF IMMEDIATE MODE FOR STATEMENT EXECUTION

It is not necessary to write a complete program to use BASIC-PLUS. Most of the statements discussed in this manual can either be included in a program for later execution or be given on-line as commands, which are immediately executed by the BASIC processor. 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"
```

produces 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 further input.

4.2 PROGRAM DEBUGGING

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

In order to facilitate debugging a program, the user can place STOP statements liberally throughout the program. Each STOP statement causes the program to halt, printing the line number at which the STOP occurred; at which time the user can examine various data values, perhaps change them in immediate mode, and then give the

```
CONT
```

command to continue program execution. However, a syntax error in immediate mode or one of several other conditions could prevent continuation of program execution with the CONT command.

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

The user can also halt program execution at any time by typing CTRL/C. Immediate mode can then be used to examine and/or change data values. Typing the CONT command resumes program execution. Whenever execution cannot be continued, the message:

```
CAN'T CONTINUE

READY
```

is printed upon entering the CONT command.

4.3 MULTIPLE STATEMENTS PER LINE

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

```
A=1: PRINT A
ILLEGAL IN IMMEDIATE MODE

READY
```

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

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

READY
```

4.4 RESTRICTIONS ON IMMEDIATE MODE

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

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

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

4.5 PROGRAM INTERRUPTION BY CTRL/C

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

```
↑C
READY
PRINT LINE
 3ØØ
READY
```

CHAPTER 5

CHARACTER STRINGS

5.1 CHARACTER STRINGS

The previous chapters describe the manipulation of numerical information; 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 any combination of the characters in Table 5-2.

Without realizing it, the reader has already encountered character strings. Consider the following program which prints the name of a month, given its number:

```
LISTNH
10 INPUT "TYPE A NUMBER BETWEEN 1 AND 12";N
15 IF N<1 OR N>12 THEN PRINT "NUMBER OUT OF RANGE":GOTO 10
20 IF N>3 THEN PRINT "THE" N "TH MONTH IS ";
25 IF N=1 THEN PRINT "THE FIRST MONTH IS JANUARY"
30 IF N=2 THEN PRINT "THE SECOND MONTH IS FEBRUARY"
35 IF N=3 THEN PRINT "THE THIRD MONTH IS MARCH"
40 IF N=4 THEN PRINT "APRIL"
45 IF N=5 THEN PRINT "MAY"
50 IF N=6 THEN PRINT "JUNE"
55 IF N=7 THEN PRINT "JULY"
60 IF N=8 THEN PRINT "AUGUST"
65 IF N=9 THEN PRINT "SEPTEMBER"
70 IF N=10 THEN PRINT "OCTOBER"
75 IF N=11 THEN PRINT "NOVEMBER"
80 IF N=12 THEN PRINT "DECEMBER"
85 END

READY

RUNNH
TYPE A NUMBER BETWEEN 1 AND 12? 9
THE 9 TH MONTH IS SEPTEMBER

READY
```

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

5.1.1 String Constants

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

```
105   LET Y$ = "FILE4"
33    B1$ = 'CAN'
80    IF A$ = "YES" GOTO 250
```

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

5.1.2 Character String Variables

Variable names can be introduced for simple strings and for both lists and matrices composed of strings (which is to say one and two dimensional string matrices). Any legal name followed by a dollar sign ($) character is a legal name for 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) are list and matrix string variables.

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)
```

cannot. Likewise,

```
A$(N)   and    A$(M,N)
```

cannot both occur in the same program.

Just as numeric variables are automatically initialized to Ø when a program is run, string variables are initialized to a null string containing zero characters (the character string constant "").

5.1.3 Subscripted String Variables

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

```
1Ø DIM S1$(5)
```

indicates the S1$ is a string matrix with six elements, 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 including the zero element) in each direction. Note that the dimension of a string matrix specifies the number of strings and not the number of characters in any one string. For example, if the first statements in a program are:

```
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.

5.1.4 String Size

A character string can contain any number of characters limited only by the amount of core storage available. 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 9.6.2).

5.1.5 Relational Operators

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.

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.
==	A$ == B$	The strings A$ and B$ are identical. This operator is not available prior to Version 5B (RSTS/E) systems.

In any string comparison (except ==), trailing blanks are ignored. That is to say "YES" is equivalent to "YES ". Where two strings of unequal length are compared, the shorter is padded with trailing blanks to the length of the longer string. A null string (of length zero) is considered to be completely blank and is less than any string of length greater than zero unless that string consists of all blanks in which case the two strings are equivalent.

5.2 ASCII STRING CONVERSIONS, CHANGE STATEMENT

Individual characters in a string can be referenced through use of the CHANGE statement. The CHANGE statement permits the user program to transform (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
10 DIM X(3)
15 LET A$ = "CAT"
20 CHANGE A$ TO X
25 PRINT X(0);X(1);X(2);X(3)
30 END

READY

RUNNH
 3  67  65  84

READY
```

X(1) through X(3) take on the ASCII values of the characters in the string variable A$. The first element of X, X(Ø), becomes the number of characters present in A$. 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 is less than or equal to the dimension of the list. Notice that line 10, above, created a 4-element array, X. A DIM statement must be used in this instance; otherwise, the system creates a default 121-element array, leading to possible illogical results.

Table 5-2

ASCII Character Codes

Decimal Value	ASCII Character	RSTS Usage	Decimal Value	ASCII Character	RSTS Usage	Decimal Value	ASCII Character	RSTS Usage
Ø	NUL	FILL character	43	+		86	V	
1	SOH		44	,		87	W	
2	STX		45	-		88	X	
3	ETX	CTRL/C	46	.		89	Y	
4	EOT		47	/		9Ø	Z	
5	ENQ		48	Ø		91	[	
6	ACK		49	1		92	\	
7	BEL	BELL	5Ø	2		93	]	
8	BS		51	3		94	^	or ↑
9	HT	HORIZONTAL TAB	52	4		95	_	or ←
1Ø	LF	LINE FEED	53	5		96	`	Grave accent
11	VT	VERTICAL TAB	54	6		97	a	
12	FF	FORM FEED	55	7		98	b	
13	CR	CARRIAGE RETURN	56	8		99	c	
14	SO		57	9		1ØØ	d	
15	SI	CTRL/O	58	:		1Ø1	e	
16	DLE		59	;		1Ø2	f	
17	DC1		6Ø	<		1Ø3	g	
18	DC2		61	=		1Ø4	h	
19	DC3		62	>		1Ø5	i	
2Ø	DC4		63	?		1Ø6	j	
21	NAK	CTRL/U	64	@		1Ø7	k	
22	SYN		65	A		1Ø8	l	
23	ETB		66	B		1Ø9	m	
24	CAN		67	C		11Ø	n	
25	EM		68	D		111	o	
26	SUB	CTRL/Z	69	E		112	p	
27	ESC	ESCAPE[1]	7Ø	F		113	q	
28	FS		71	G		114	r	
29	GS		72	H		115	s	
3Ø	RS		73	I		116	t	
31	US		74	J		117	u	
32	SP	SPACE	75	K		118	v	
33	!		76	L		119	w	
34	"		77	M		12Ø	x	
35	#		78	N		121	y	
36	$		79	O		122	z	
37	%		8Ø	P		123	{	
38	&		81	Q		124	|	Vertical Line
39	'		82	R		125	}	
4Ø	(		83	S		126	~	Tilde
41	)		84	T		127	DEL	RUBOUT
42	*		85	U				

[1]ALTMODE (ASCII 125) or PREFIX (ASCII 126) keys which appear on some terminals are translated internally into ESCAPE.

NOTE

The decimal values 128 through 255 can appear in character strings. For most practical purposes, the characters represented by N and N+128 (decimal) are the same. However, the characters CHR$(N) and CHR$(N+128) do not test as equal if compared. Users should be careful when performing output of these values since they may have some significance in certain device-dependent operations (see Chapter 12).

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

```
LISTNH
10 DIM A(65)
15 READ A$
20 CHANGE A$ TO A
25 FOR I=0 TO A(0)
30 PRINT A(I);:NEXT I
35 DATA ABCDEFGHIJKLMNOPQRSTUVWXYZ
40 END

READY

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

Notice that A(Ø) = 26.

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

```
LISTNH
10 FOR I=0 TO 5
15 READ A(I)
20 NEXT I
25 DATA 5,65,66,67,68,69
30 CHANGE A TO A$
35 PRINT A$
40 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 of a numeric list to character string conversion:

```
LISTNH
10 DIM V(128)
15 INPUT "HOW MANY CHARACTERS";V(0)
20 FOR I=1 TO V(0)
25 INPUT V(I)
30 NEXT I
35 CHANGE V TO A$
40 PRINT A$
50 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:

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

causes the following assignments to be made:

```
A$ = the character string "17"
B  = 14
C  = 13.4
reading D as CAT causes the message ILLEGAL NUMBER AT LINE 1Ø
to be printed.
```

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

```
LISTNH
10 READ A$,B$,C$,D$,E$
20 PRINT A$;B$;C$;D$;E$
30 PRINT A$,B$,C$,D$,E$
40 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 last statement on a line. See also the MAT READ statement which reads matrices (either numeric or string), Section 7.2.

NOTE

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

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

```
10 INPUT "YOUR NAME";N$;"YOUR AGE";A
```

is functionally equivalent to:

```
30 PRINT "YOUR NAME";
35 INPUT N$
40 PRINT "YOUR AGE";
45 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:

```
10 INPUT LINE A$
```

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

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

```
10 INPUT LINE "TEXT";A$
SYNTAX ERROR AT LINE 10
```

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

a. Carriage return/line feed, generated by typing the RETURN key (appends the ASCII values 13 and 1Ø to the character string);

b. Line feed, generated by typing the LINE FEED key (appends the ASCII values 1Ø, 13 and Ø to the character string); or

c. ESCAPE, generated by typing the ESCAPE, ALT MODE or PREFIX key, depending upon the terminal (appends an ASCII 27 to the character string).

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
10 X=1.0:Y=2.01:A$="A="
20 PRINT A$;X" B="Y
30 PRINT "DONE"
40 END

READY
```

```
RUNNH
A= 1  B= 2.01
DONE

READY
```

Semicolons separating character string constants from other list items are optional. For example, in line 20 (above) note that the variable Y is not separated from the character string " B=" by a semicolon.

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

5.5 STRING FUNCTIONS

Like the intrinsic mathematical functions (e.g., SIN, LOG), BASIC-PLUS contains various functions for use with character strings. These functions allow the program to concatenate two strings, access part of a string, determine the number of characters in a string, generate a character string corresponding to a given number or vice versa, search for a substring within a larger string, and perform other useful operations. (These functions are particularly useful when dealing with whole lines of alphanumeric information input by an INPUT LINE statement.) The various functions available are summarized in Table 5-3.

5.5.1 User-Defined String Functions

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

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

```
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. Line 80 is <u>unacceptable</u>:

```
10 DEF FNA$(A$) = CHR$(LEN(A$)+1)
80 LET Z=FNA$(4)
```

The message:

```
ARGUMENTS DON'T MATCH AT LINE 80
```

is printed.

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

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

READY

RUNNH
 123

READY
```

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

"(space)123(space)"

Table 5-3

String Functions[1]

Function Code	Meaning

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

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

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

LEN(A$)	Indicates the number of characters in the string A$ (including trailing blanks). For example:
	`PRINT LEN(A$)`
	` 26`

+	Indicates a concatenation operation on two strings. For example "ABC"+"DEF" is equivalent to "ABCDEF". "12"+"34"+"56" is equivalent to "123456".
CHR$(N%)	Generates a one-character string having the ASCII value of N (see Table 5-2). For example: CHR$(65) is equivalent to "A". Only one character can be generated.
ASCII(A$)	Generates the ASCII value of the first character in A$. For example, ASCII("X") is equivalent to 88, the ASCII equivalent of X. If B$ = "XAB", then ASCII(B$) = 88.
DATE$(N%)	where N=Ø, this function returns the current date in the form: 12-Aug-72 This quantity can be printed on output by simple reference to the function. It should be noted that dates are output using both upper and lower case letters. When the output device is not capable of generating lower case letters, the ASCII values still imply lower case. Where N≠Ø, the function translates N into a date string. (See Section 8.8.)

[1] A$ in the immediate mode examples is assumed to be: "ABCDEFGHIJKLMNOPQRSTUVWXYZ".

Table 5-3 (Cont.)

String Functions

Function Code	Meaning

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$, and the character position if B$ is found to be in A$ (character position is measured from the start of the string with the first character counted as character 1). For example:
	`PRINT INSTR(5%,A$,"OP")`
	`15`
	If B$ is a null string (B$ = ""), the INSTR function returns the value 1. The null string is a proper substring of any string and is treated conventionally as the first element of A$ in null string search operations. In addition, if both A$ and B$ are null strings, the INSTR function returns the value 1

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

NUM$(N)	Indicates a string of <u>numeric</u> characters representing the value of N as it would be output by a PRINT statement. NUM$(n)=(space)n (space) if n>Ø and NUM$(n)=-n(space) if n<Ø. For example:
	`PRINT NUM$(1.ØØ)"A"`
	` 1 A`

VAL(A$)	Computes the numeric value of the string of <u>numeric</u> characters A$ (may include digits, +, -, . and E). If A$ contains any characters not acceptable as numeric input with the INPUT statement, an error results. For example:
	`PRINT VAL("1.5E1")`
	`15`

TIME$(N)	Where N=Ø, this function returns the current time-of-day as a string of the form: 1:30 PM where N<>Ø, the function translates N into a time string (See section 8.8). If the system was generated using the 24-hour time option, 1:30 PM is returned as 13:30.

Table 5-3 (Cont.)

String Functions

Function Code	Meaning
STRING$(N1,N2)	Creates a string of length N1 and characters whose ASCII decimal value is N2. For example, to create a string Y$ composed of 10 space (blank) characters CHR$(32%), execute the following statement: `Y$ = STRING$(10,32)` See Table 5-2 for the decimal values of ASCII characters.
CVT$$ (S$,M%)	Converts the source character string S$ according to the decimal value of the integer M%. For a complete explanation of this function, see Section 12.5.

CHAPTER 6

INTEGER AND FLOATING POINT OPERATIONS

Numbers on the system can be represented and manipulated in either integer or floating point format as described in Section 2.5.1. The implications of representing numbers in a certain format and the resultant benefits are described in this chapter. Certain operations involving integer numbers are more efficient if performed using a forced one-word integer format. The specification of a forced integer format and the possible integer operations are described in Section 6.1 through 6.6. The results of performing operations by mixing the formats are described in Section 6.7. Operations using standard floating point arithmetic and floating point scaled arithmetic are performed as described in Section 6.8.

6.1 INTEGER CONSTANTS AND VARIABLES

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

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

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

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

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

```
A%=-1.1
```

causes A% to be assigned the value -1.

6.2 INTEGER ARITHMETIC

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

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

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

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

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

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

takes less storage space and executes faster than:

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

6.3 INTEGER I/O

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

```
LISTNH
10 READ A, B%, C, D%, E
20 PRINT A, B%, C, D%, E
30 DATA 2.7,3,4,5.7,6.8

READY

RUNNH
DATA FORMAT ERROR AT LINE 10

READY
```

when line 3Ø is changed to

```
     3Ø DATA 2.7,3,4,5,6.8
```

the following is printed:

```
     RUNNH
      2.7            3              4              5              6.8

     READY
```

6.4 USER DEFINED INTEGER FUNCTIONS

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

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

```
       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 as the system performs the necessary conversions. However, strings cannot be used where numbers are required (or vice versa).

```
       75 DEF FNA%(X%) =X%-1%
       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 or integer valued expressions can be used within IF statements in any place that a logical expression can appear. An integer value of Ø% corresponds to the logical value FALSE, and any non-zero value is defined to be TRUE. The logical operators (AND, OR, NOT, XOR, IMP, EQV) operate on logical (or integer) data in a bitwise manner. The integer -1% (which is represented internally as sixteen binary ones) is normally used by the system 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 in an IF statement:

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

READY

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

READY
```

6.6 LOGICAL OPERATIONS ON INTEGER DATA

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

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

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

A	B	A AND B
1	1	1
1	∅	∅
∅	1	∅
∅	∅	∅

A	B	A OR B
1	1	1
1	∅	1
∅	1	1
∅	∅	∅

A	B	A XOR B
1	1	∅
1	∅	1
∅	1	1
∅	∅	∅

A	B	A EQV B
1	1	1
1	∅	∅
∅	1	∅
∅	∅	1

A	B	A IMP B
1	1	1
1	∅	∅
∅	1	1
∅	∅	1

A	NOT A
1	∅
∅	1

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

```
PRINT 85% AND 28%
 2∅

READY
```

Each bit in the internal representation of 85% is combined with each corresponding bit in the internal representation of 28% according to the rules in the AND truth tables. By consulting the AND (logical product) truth table, it can be seen that a bit is generated in the bit position of the result only if both bits are 1 in the corresponding bit position of the integer values 85% and 28%. The resultant value of 2∅ printed by BASIC is the integer value of the bits set in the internal representation of the logical product.

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

```
PRINT 85% OR 28%
 93

READY
```

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

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

```
LISTNH
10 C% = 85% AND 28%
20 PRINT C%

READY

RUNNH
 20

READY
```

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

```
10 L% = W% AND 255%
```

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

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

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

6.7 MIXED MODE ARITHMETIC

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

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

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

READY
```

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

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

```
PRINT 1%/2; 1/2%; 1/2
 Ø  .5  .5

READY
```

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

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

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

READY
```

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

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

READY
```

The explicit per cent sign and the decimal point determine the format of the result and enables the user to control the result.

6.8 FLOATING POINT AND SCALED ARITHMETIC

Floating point numbers occupy either two 16-bit words or four 16-bit words of storage in memory. With the single precision package, 2 words are used; with the double precision package, 4 words are used. Appendix F describes the internal format of the two packages.

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

The system performs output of numeric results of floating point calculations as described in Section 2.5.1. To perform output of numbers larger than six digits, the user can tailor the format as described in Section 10.4.1 for the PRINT USING statement.

Since all fractional numbers cannot be represented exactly in binary notation, certain calculations in floating point result in an accumulated error. For example, the following calculation, run in standard four-word floating point, results in an accumulated error.

```
LISTNH
10 X = 0
20 X = X + .01 FOR I%=1% TO 10000%
30 PRINT X - 100: END

READY

RUNNH
-.177636E-11

READY
```

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

To perform decimal calculation on a system having the double precision floating point (4-word) math package, the user can employ the scaled arithmetic feature to avoid accumulated error. Systems with two word precision do not have scaled arithmetic. The user can specify the number of decimal places in fractional numbers by use of the SCALE command. (See the description of the SCALE command in Section 2.8 of the RSTS-11 System User's Guide).

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

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

A scale factor between 1 and 6 determines the accuracy of fractional numbers. For example, with a scale factor of 2 in effect, the following statement, upon input,

```
        X = .Ø1
```

causes the system to move the decimal point 2 places to the right. If any rounding is necessary, the system does it at this point. The system then converts the result, 1, to a floating point representation. Similarly, .1 becomes 10 internally; and all numbers less than .005 become Ø.

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

```
LISTNH
1Ø X = Ø
2Ø X = X + .Ø1 FOR I%=1% TO 1ØØØ%
3Ø PRINT X - 1ØØ: END

READY

RUNNH
 Ø

READY
```

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

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

With a scale factor in effect, the system handles output by PRINT and PRINT USING statements in the standard manner. The PRINT statement still handles 6 digits or less and uses the E format for numbers larger than 6 digits. The PRINT USING statement formats numbers according to the specified string.

The mathematical functions described in Section 3.7 can be used in conjunction with the scaled arithmetic feature. With a non-zero scale factor in effect, the system automatically descales the number passed, computes the value of the function, and converts, with any necessary rounding, the value returned to an appropriately scaled floating point integer.

CHAPTER 7

MATRIX MANIPULATION

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

7.1 BASIC-PLUS ARRAY STORAGE

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

```
10 DIM A(10,10)
```

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

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

7.2 MAT READ STATEMENT

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

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:

```
10 DIM A(20,20)
20 MAT READ A
```

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

```
20 MAT READ A(5,15)
```

a five by fifteen matrix would be read and the matrix A would be redimensioned.

7.3 MAT PRINT STATEMENT

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

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

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

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

```
    10 DIM A(10,10),B(20,20)

    120 MAT PRINT A;           !PRINT 10*10 MATRIX,PACKED FORMAT
    130 MAT PRINT B(N,M),      !PRINT N*M MATRIX, 5 ELEMENTS
                               !PER LINE
```

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

```
        MAT PRINT V
```

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

```
        MAT PRINT V,
```

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

```
        MAT PRINT V;
```

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

```
        LISTNH
        10 DIM A(7),X(5)
        20 MAT READ A,X
        30 MAT PRINT A;:PRINT:MAT PRINT X
        40 DATA 21,22,23,24,35,36,37,51,52,53,54,55
        50 END

        READY

        RUNNH
         21  22  23  24  35  36  37

         51
         52
         53
         54
         55
```

7.4 MAT INPUT STATEMENT

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

line number MAT INPUT <*list of matrices*>

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

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

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

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

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

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

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

```
50 DIM A(20,20),B%(2,2)
     .
     .
     .
100 MAT INPUT A(2,1)
110 MAT INPUT B%,C$
```

The matrix A is redimensioned in line 1ØØ. The INPUT statement proceeds to accept input until the entire matrix has been read or the RETURN or ESCAPE delimiter is encountered. Several lines can be input by terminating the physical keyboard line with a line feed to indicate continuation on the following line.

Following the input of a matrix, the two variables NUM and NUM2 contain the number of elements input. NUM contains the number of rows input or, for a one dimensional matrix, the number of elements entered. NUM2 contains the number of elements in the last row. For example, the following program inputs a variable size matrix (up to 10x10):

```
50 DIM A(10,10)
100 INPUT "TYPE MATRIX DIMENSIONS";N,M
110 MAT INPUT A(N,M)
120 !CHECK TO SEE IF ENTIRE MATRIX WAS ENTERED
130 IF NUM*NUM2=N*M THEN 1000
140 PRINT "YOU DIDN'T ENTER THE WHOLE MATRIX"
150 GOTO 100
      •
      •
      •
```

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

```
100 MAT INPUT "TEXT" A%
SYNTAX ERROR AT LINE 100
```

7.5 MATRIX INITIALIZATION STATEMENTS

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

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

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

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

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

```
10 DIM A(10,10),B(15),C(20,20)
20 MAT A=ZER          !SETS ALL ELEMENTS OF A=0
30 MAT B=CON(10)      !SETS FIRST 10 ELEMENTS OF B=1
40 MAT C=IDN(10,10)
```

It should be noted that these instructions do not set row zero or column zero.

7.6 MATRIX CALCULATIONS

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

7.6.1 Matrix Operations

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

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

```
110 DIM A(50), B(25), C(50)
120 MAT C=A+B
RUNNH
MATRIX DIMENSION ERROR AT LINE 120
READY
```

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

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

Multiplication of conformable matrices is indicated as follows:

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

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

Scalar multiplication of a matrix is performed as follows:

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

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

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

```
        120 MAT C=A
```

7.6.2 Matrix Functions

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

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

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

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

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

```
        200 MAT A = INV(X): D1=DET
        210 MAT B = INV(A): D2=DET
        220 IF D1=1/D2 GOTO 340 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; however, inversion destroys the previous contents of these elements. The operation MAT A = INV(A) is legal.

CHAPTER 8

ADVANCED STATEMENT FEATURES

8.1 DEF STATEMENT, MULTIPLE LINE FUNCTION DEFINITIONS

In Chapter 3 the DEF statement is described as 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 function definition is as follows:

line number DEF FN<*identifier*><*(dummy arguments)*>

<*body of definition*>

line number FNEND

The multiple-line DEF function is distinguished from the one-line user functions by the absence of an equal sign following the function name on the first line. (From zero to five arguments of any type or mixture of types can be used.) The value returned by the function is the value of FN<*identifier*> at the time the FNEND statement is encountered. Somewhere within the multiple-line definition there 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. (There may be more than one such statement, as in the example below.)

The function example 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:

```
10 DEF FNM(X,Y)
20 LET FNM=X
30 IF Y<=X THEN 50
40 LET FNM=Y
50 FNEND
```

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

```
LISTNH
10 DEF FNF(M%)
20 IF M%=1% THEN FNF=1 ELSE FNF=M%*FNF(M%-1%)
30 FNEND
35 INPUT "VALUE FOR FACTORIAL";M
40 PRINT M"FACTORIAL EQUALS"FNF(M)
50 END

READY

RUNNH
VALUE FOR FACTORIAL? 4
 4 FACTORIAL EQUALS 24

READY
```

Any variable referenced in the body of a function definition which is not an argument of that multiple line DEF function has its current value in the user program. Multiple-line DEF functions can be nested (one multiple-line definition can reference another multiple-line definition 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; attempts by the program to modify them are cancelled when the function exits to its calling program:

```
LISTNH
10 DEF FNB(X)
20 X=0: FNB=10
30 FNEND
40 A=1: B=FNB(A)
50 PRINT A,B
60 END

READY

RUNNH
 1             10

READY
```

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

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

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

```
LISTNH
10 DEF FNA$(A,B,C%)
20 IF A>B GOTO 40
30 FNA$=CHR$(A+1): GOTO 50
40 FNA$=CHR$(A+C%)
50 FNEND
60 INPUT "VALUES FOR A,B,C%";A,B,C%
70 PRINT "FNA$(A,B,C%) = "FNA$(A,B,C%)
80 END

READY

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

READY

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

READY
```

8.2 ON-GOTO STATEMENT

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

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:

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

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 an error message to be printed (or a transfer to an ON ERROR-GOTO routine with ERR=58).

8.3 ON-GOSUB STATEMENT

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

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

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

An example of the statement follows:

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

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

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

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

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

8.4 ON ERROR GOTO STATEMENT

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

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

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

line number ON ERROR GOTO {<*line number*>}

This statement is placed in the program prior to any executable statements with which the error handling routine deals. If an error does occur, user program execution is interrupted and the user written error subroutine is started at the line number indicated. The variable ERR, available to the program, assumes one of the values listed in Table 8-1. Table 8-1 is also contained in Appendix C, the complete RSTS error message summary.

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

Table 8-1

User Recoverable Errors

(C) indicates that program execution continues, following printing of the error message, if an ON ERROR GOTO statement is not present. Otherwise, execution terminates and the system prints the READY message.

ERR	Message Printed	Meaning
1	BAD DIRECTORY FOR DEVICE	The directory of the device referenced is in an unreadable format.
2	ILLEGAL FILE NAME	The filename specified is not acceptable. It contains embedded blanks or unacceptable characters.
3	ACCOUNT OR DEVICE IN USE	The specified operation cannot be performed because the file is already open by some user. This message has a general "file in use" meaning.
4	NO ROOM FOR USER ON DEVICE	Storage space allowed for the current user on the device specified has been used or the device as a whole is too full to accept further data.
5	CAN'T FIND FILE OR ACCOUNT	The file specified or current user account numbers were not found on the device specified. This message has a general "not there" meaning.
6	NOT A VALID DEVICE	Attempt to use an illegal or nonexistent device specification
7	I/O CHANNEL ALREADY OPEN	An attempt was made to open one of the twelve I/O channels which had already been opened by the program.
8	DEVICE NOT AVAILABLE	The device requested is currently reserved by another user.
9	I/O CHANNEL NOT OPEN	Attempt to perform I/O on one of the twelve channels which has not been previously opened in the program.
10	PROTECTION VIOLATION	The current user is not allowed to perform the requested operation on the specified file. Input may have been requested from an output-only device or vice versa. This message has a general "can't do that" meaning.

ERR	Message Printed	Meaning
11	END OF FILE ON DEVICE	Attempt to perform input beyond the end of a data file; or a BASIC source file is called into memory and is found to contain no END statement.
12	FATAL SYSTEM I/O FAILURE	An I/O error has occurred on the system level. The user has no guarantee that the last operation has been performed.
13	USER DATA ERROR ON DEVICE	One or more characters may have been transmitted incorrectly due to a parity error, bad punch combination on a card or similar error.
14	DEVICE HUNG OR WRITE LOCKED	User should check hardware condition of device requested. Possible causes of this error include a line printer out of paper or high-speed reader being off-line.
15	KEYBOARD WAIT EXHAUSTED	Time requested by WAIT statement has been exhausted with no input received from the specified keyboard.
16	NAME OR ACCOUNT NOW EXISTS	An attempt was made to rename a file with the name of a file which already exists, or an attempt was made by the system manager to insert an account code which is already within the system.
17	TOO MANY OPEN FILES ON UNIT	Only one open DECtape output file is permitted per DECtape drive. Only one open file per magtape drive is permitted.
18	ILLEGAL SYS() USAGE	Illegal use of the SYS system function.
19	DISK BLOCK IS INTERLOCKED	The requested disk block segment is already in use (locked) by some other user.
20	PACK IDS DON'T MATCH	The identification code for the specified disk pack does not match the identification code on the pack.
21	DISK PACK IS NOT MOUNTED	No disk pack is mounted on the specified disk drive.
22	DISK PACK IS LOCKED OUT	The disk pack specified is mounted but temporarily disabled.
23	ILLEGAL CLUSTER SIZE	The specified cluster size is unacceptable.
24	DISK PACK IS PRIVATE	The current user does not have access to the specified private disk pack.
25	DISK PACK NEEDS 'CLEANING'	Non-fatal disk mounting error; use CLEAN system call.

ERR	Message Printed	Meaning
26	FATAL DISK PACK MOUNT ERROR	Fatal disk mounting error.
27	I/O TO DETACHED KEYBOARD	I/O was attempted to a hung up data-set or to the previous, but now de-tached, console keyboard for the job.
28	PROGRAMMABLE ↑C TRAP	ON ERROR-GOTO subroutine was entered through a program trapped CTRL/C. See a description of the SYS system function.
29	CORRUPTED FILE STRUCTURE	Fatal error in CLEAN system call.
30	DEVICE NOT FILE STRUCTURED	An attempt is made to access a device, other than a disk,DECtape, or magtape device, as a file-structured device. This error occurs, for example, when the user attempts to gain a directory listing of a non-directory device.
31	ILLEGAL BYTE COUNT FOR I/O	The buffer size specified in the RECORDSIZE option of the OPEN statement or in the COUNT option of the PUT statement is not a multiple of the block size of the device being used for I/O.
32	NO ROOM AVAILABLE FOR FCB	When the user accesses a file under programmed control in RSTS-11, a sys-tem control structure called an FCB requires one small buffer and one small buffer is not available for the FCB.
33	UNIBUS TIMEOUT FATAL TRAP	This hardware error occurs when an attempt is made to address nonexistent memory or an odd address using the PEEK function. An occurrence of this error message in any other case is cause for an SPR.
34	RESERVED INSTRUCTION TRAP	An attempt is made to execute an illegal or reserved instruction or an FPP instruction when floating point hardware is not available. (SPR)
35	MEMORY MANAGEMENT VIOLATION	This hardware error occurs when an illegal Monitor address is specified using the PEEK function. Generation of the error message in situations other than using PEEK is cause for an SPR.
36	SP (R6) STACK OVERFLOW	An attempt to extend the hardware stack beyond its legal size is encountered. (SPR)
37	DISK ERROR DURING SWAP	A hardware error occurs when a user's job is swapped into or out of memory. The contents of the user's job area are lost but the job remains logged into the system and is reinitialized to run the NONAME program. (SPR)

ERR	Message Printed	Meaning
38	MEMORY PARITY ERROR	A parity error was detected in the memory occupied by this job.
39	MAGTAPE SELECT ERROR	When access to a magtape drive was attempted, the selected unit was found to be off line.
40	MAGTAPE RECORD LENGTH ERROR	When performing input from magtape, the record on magtape was found to be longer than the buffer designated to handle the record.
41	NO RUN-TIME SYSTEM	Reserved.
42	VIRTUAL BUFFER TOO LARGE	Virtual core buffers must be no more than 512 decimal bytes long.
43	VIRTUAL ARRAY NOT ON DISK	A non-disk device is open on the channel upon which the virtual array is referenced.
44	MATRIX OR ARRAY TOO BIG	In-core array size is too large.
45	VIRTUAL ARRAY NOT YET OPEN	An attempt was made to use a virtual array before opening the corresponding disk file.
46	ILLEGAL I/O CHANNEL	Attempt was made to open a file on an I/O channel outside the range of the integer numbers 1 through 12.
47	LINE TOO LONG	Attempt to input a line longer than 255 characters (which includes any line terminator). Buffer overflows.
48	FLOATING POINT ERROR	Attempt to use a computed floating point number outside the range $\|1E{-}38\| \le n \le \|1E38\|$ excluding zero. If no transfer to an error handling routine is made, zero is returned as the floating point value. (C)
49	ARGUMENT TOO LARGE IN EXP	Maximum is in the range $-89 \le arg \le +88$. Value returned is zero. (C)
50	DATA FORMAT ERROR	A READ or INPUT statement detected data in an illegal format. For example, 1..2 is an improperly formed number, 1.3 is an improperly formed integer, and X" is an illegal string. (C).
51	INTEGER ERROR	Attempt to use a computed integer outside the range $-32767 \le n \le 32767$. If no transfer to an error handling routine is made, zero is returned as the integer value. (C)
52	ILLEGAL NUMBER	Integer or floating point overflow or underflow.
53	ILLEGAL ARGUMENT IN LOG	Negative or zero argument to log function. Value returned is the argument as passed to the function. (C)
54	IMAGINARY SQUARE ROOTS	Attempt to take square root of a number less than zero. The value returned is the square root of the absolute value of the argument. (C)

ERR	Message Printed	Meaning
55	SUBSCRIPT OUT OF RANGE	Attempt to reference an array element beyond the number of elements created for the array when it was dimensioned.
56	CAN'T INVERT MATRIX	Attempt to invert a singular matrix.
57	OUT OF DATA	The DATA list was exhausted and a READ requested additional data.
58	ON STATEMENT OUT OF RANGE	The index value in an ON-GOTO or ON-GOSUB statement is less than one or greater than the number of line numbers in the list.
59	NOT ENOUGH DATA IN RECORD	An INPUT statement did not find enough data in one line to satisfy all the specified variables.
60	INTEGER OVERFLOW, FOR LOOP	The integer index in a FOR loop attempted to go beyond 32766 or below -32766.
61	DIVISION BY Ø	Attempt by the user program to divide some quantity by zero. (C) If no transfer is made to an error handling routine, a Ø is returned as the result.

8.4.1 RESUME Statement

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 (which is placed at the end of the error handling routine, much like a RETURN statement in a normal subroutine). The RESUME statement causes the program 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 at the end of the error handling routine.

The format of the RESUME statement is as follows:

line number RESUME {<*line number*>}

For example:

```
2000 RESUME
2001 RESUME 100
```

The line 2ØØØ restarts the user program at the line in which the error was detected, and is equivalent to the statement:

```
2000 RESUME 0
```

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

```
50 A=A+1 : PRINT A : FOR M=1% TO 3% : INPUT X
```

If an error occurs in the INPUT statement, above, control is passed to the preceding FOR statement on the same line - not to the first statement of the line.

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

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

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

8.4.2 Disabling the User Error Handling Routine

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

```
        line number  ON ERROR GOTO Ø
```

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

```
        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.

Generally, the error handling subroutine detects and properly handles only a few different errors; it is useful to have the RSTS system handle other errors, if they occur. For this reason, RSTS allows the ON ERROR GOTO Ø statement 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 and a message printed as though no ON ERROR GOTO statement had been in effect.

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

The program below requests the user's name with the INPUT statement on line 30. The ON ERROR GOTO statement is previously executed on line 10.

```
10 ON ERROR GOTO 1000              !SET UP ERROR ROUTINE
20 WAIT(60)                        !WAIT 60 SEC. FOR REPLY
30 INPUT "YOUR NAME";N$            !GET STUDENT NAME
50 STOP
    .
    .
    .
1000 !THIS IS THE ERROR HANDLING ROUTINE
1010 IF ERR<>15 THEN ON ERROR GOTO 0  !WAIT ERRORS ONLY
1020 PRINT                         !SKIP TO NEW LINE
1030 PRINT "PLEASE TYPE YOUR NAME"
1040 PRINT "AND THEN HIT THE 'RETURN' KEY"
1050 RESUME                        !TRY AGAIN
```

In this example, if the call to the error subroutine was caused by some error other than the KEYBOARD WAIT EXHAUSTED error, the program would exit via the ON ERROR GOTO Ø in line 1Ø1Ø. This permits the appropriate error message to be printed on the user's terminal. Note that exiting via the RESUME at line 1Ø5Ø causes the INPUT statement to be restarted.

8.4.3 The ERL Variable

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

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

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

```
10 ON ERROR GOTO 100
20 INPUT "TYPE TWO NON-ZERO NUMBERS";A,B
30 LET X=A/B
40 LET X=X+B/A
50 PRINT X
60 STOP
   .
   .
   .
100 IF ERR<>61 THEN ON ERROR GOTO 0
110 PRINT "FIRST NUMBER WAS 0" IF ERL=40
120 PRINT "SECOND NUMBER WAS 0" IF ERL=30
```

If the LET statements in lines 3Ø and 4Ø were moved to some other line numbers, lines 11Ø and 12Ø would also require a change.

8.5 IF-THEN-ELSE STATEMENT

The IF-THEN statement allows the program to transfer control to another line or execute a specified statement depending upon a stated condition.

The IF-THEN-ELSE statement is the same as the IF-THEN statement, except that rather than executing the line following the IF statement, another line number or statement can be specified for execution where the condition is not met. The statement is of the form:

$$\textit{line number}\quad \text{IF}\langle\textit{condition}\rangle \left[\begin{array}{l}\text{THEN}\langle\textit{line number}\rangle\\ \text{THEN}\langle\textit{statement}\rangle\\ \text{GOTO}\langle\textit{line number}\rangle\end{array}\right]\left\{\begin{array}{l}\text{ELSE}\langle\textit{line number}\rangle\\ \text{ELSE}\langle\textit{statement}\rangle\end{array}\right\}$$

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 relational condition is tested; if it is true the THEN/GOTO part of the statement is executed. If the condition is false, the ELSE part of the statement is executed. Following the word ELSE is either a statement to be executed or a line number to which control is transferred.

As an example of an IF-THEN-ELSE statement:

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

An IF statement can follow either the THEN or ELSE clause in the above statement, making it possible to nest IF statement to any desired level. For example:

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

```
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"
30 END

READY

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

READY

RUNNH
? 3,6,1
B>A>C

READY
```

The use of the LINE FEED and TAB characters greatly improves the legibility of complex program statements such as line 20 above.

The IF-THEN-ELSE statement can appear anywhere in a multiple-statement line. However, if this statement is followed by any other statements, the following rules apply:

a. The physically last THEN or ELSE clause is considered to be followed by the next statement on the line:

```
10 IF A=1 THEN 100 ELSE PRINT A: PRINT "ONE"
```

   where A≠1, the value of A and the text string ONE are printed.

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

```
20 IF A>B THEN IF B<C THEN PRINT "B<C": GOTO 30
25 PRINT "A<B"
```

   Only in the case where "B<C" is printed is the statement GOTO 3Ø seen and executed.

   If either A≤B or B≥C, the line "A<B" is printed.

8.6 CONDITIONAL TERMINATION OF FOR LOOPS

In the simple FOR-NEXT loop described in Section 3.6.1, the format of the FOR statement is given as:

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<*relational expression*>

and

*line number*FOR<*variable*>=<*expression*>{STEP<*expression*>} UNTIL<*relational expression*>

The condition has the same structure as specified in an IF statement (see Section 3.5) and can be just as elaborate, if necessary. Before the loop is executed and at each loop iteration the condition is tested. The iteration proceeds if the result is true (FOR-WHILE) or false (FOR-UNTIL).

The difference between a FOR loop specified with a WHILE or UNTIL and one specified with a terminal value for the loop variable is worth noting, in order to avoid potential pitfalls in the usage of each. Consider the two loops in the program below:

```
LISTNH
10 FOR I=1 TO 10
15 PRINT I;
20 NEXT I
25 PRINT "I="I
50 FOR I=1 UNTIL I>10
55 PRINT I;
60 NEXT I
65 PRINT "I="I
75 END

READY

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

READY
```

Each of these loops prints the numbers from 1 to 10. When the loop at line 10 is done, however, the loop variable is set to the last value used (that is, 10). In the second loop beginning at line 50, the loop variable is set to the value which caused the loop to be terminated (that is, 11).

Next consider the two loops following:

```
LISTNH
10 X=10
20 FOR I=1 TO X
30 X=X/2: PRINT I,X
40 NEXT I
50 PRINT
60 X=10
70 FOR I=1 UNTIL I>X
80 X=X/2: PRINT I,X
90 NEXT I
95 END

READY

RUNNH
 1             5
 2             2.5
 3             1.25
 4             .625
 5             .3125
 6             .15625
 7             .078125
 8             .390625E-1
 9             .195313E-1
 10            .976563E-2

 1             5
 2             2.5

READY
```

In the case of the loop beginning with line 20, the iteration stops when I exceeds the initial value of X (that is, 10). Even though the value of X changes within the loop, the initial value of X determines the performance of the loop. In the second loop, the current value of X determines when the iteration ceases. Thus, after three iterations, I is greater than X in the second loop and the loop is terminated. (The STEP value when omitted, is still assumed to be 1.)

These forms of loop control are particularly useful in iterative applications where data generated during the loop execution determines loop completion.

Consider the problem of scanning a table of values until two successive elements are both 0, or the end of the table is reached:

```
      .
      .
      .
100 FOR I=1 UNTIL I=N OR X(I)=0 AND X(I+1)=0
115 NEXT I
      .
      .
      .
```

The following two programs also illustrate the FOR-UNTIL and FOR-WHILE constructions:

```
LISTNH
10 INPUT "LETTER IS";Y$
20 X$="": FOR I=1 UNTIL X$=Y$ OR X$="ZZZ"
30 READ X$: NEXT I
40 DATA A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P,Q,R,S,T,U,V,W,X,Y,Z,ZZZ
50 PRINT "LETTER IS NUMBER"I-1
90 END

READY

RUNNH
LETTER IS? C
LETTER IS NUMBER 3

READY

RUNNH
LETTER IS? Q
LETTER IS NUMBER 17

READY
```

```
LISTNH
10 INPUT "WORD";Y$
20 X$="": FOR I=1 WHILE X$<=Y$
30 READ X$: NEXT I
40 DATA A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P,Q,R,S,T,U,V,W,X,Y,Z,ZZZ
50 PRINT "WORD BEGINS WITH LETTER" I-2
90 END

READY

RUNNH
WORD? FIRST
WORD BEGINS WITH LETTER 6

READY

RUNNH
WORD? LAST
WORD BEGINS WITH LETTER 12

READY
```

8.7 STATEMENT MODIFIERS

To increase the flexibility and ease of expression within BASIC-PLUS, five statement modifiers are available (IF, UNLESS, FOR, WHILE, and UNTIL). These modifiers are appended to program statements to indicate conditional execution of the statements or the creation of implied FOR loops.

8.7.1 The IF Statement Modifier

The form:

<*statement*>IF<*condition*>

is analogous to the form:

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

For example:

```
10 PRINT X IF X<>Ø
```

is the same as:

```
10 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:

```
100 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:

```
70 PRINT X(I,J) IF I=J IF X(I,J)<>0
```

prints the value of X(I,J) only if the value of X(I,J) is non-zero and if I equals J. When there is more than one modifier on a line, the modifiers are executed in a right-to-left order. That is, the rightmost one is executed first and the leftmost is executed last. This situation is described by the term "nested modifiers".

An additional operational advantage of IF modifiers is illustrated in the discussion of FOR modifiers in Section 8.7.3.

8.7.2 The UNLESS Statement Modifier

The form:

<statement> UNLESS *<condition>*

causes the statement to be executed only if the condition is false. This particular form simplifies the negation of a logical condition. For example, the following statements are all equivalent:

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

Do not use the implicit GOTO statement with the UNLESS modifier; this may cause CATASTROPHIC ERRORS when the condition in the modifier are met. Instead, use the GOTO statement explicitly, as follows:

```
IF A=B THEN GOTO 600 UNLESS C=D
```

Do not use:

```
IF A=B THEN 600 UNLESS C=D
```

8.7.3 The FOR Statement Modifier

The form:

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

or, the form

*<statement>*FOR*<variable>*=*<expression>*{STEP*<expression>*} $\left[\begin{matrix}\text{WHILE}<\textit{expression}>\\ \text{UNTIL}<\textit{expression}>\end{matrix}\right]$

can be used to imply a FOR loop on a single line. For example (using none of the optional elements):

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

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

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

In cases where the FOR-NEXT loop is extremely simple, the necessity for both a FOR and a NEXT statement is eliminated. Notice that this implied FOR loop will only modify (and hence execute iteratively) one statement in the program. Any number of implied FOR loops can be used in a single program.

As in the case with all modifiers, a FOR modifier in an IF statement operates only on the THEN or ELSE clause with which it is associated, and never on the conditional expression to the left of the THEN. Thus, if it was desired to print all non-zero values in a matrix X(100), the following program would not operate properly:

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

since the implied FOR loop at line 20 applies only to the THEN PRINT... part of the statement, and not to the IF... part. The first value of X tested is X(100), since I remained at 100 from statement 15. To achieve the desired effect, it is only necessary to state line 20, not as an IF statement, but rather as a PRINT statement with nested modifiers; for example:

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

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

The WHILE and UNTIL clauses are explained in Section 8.6.

8.7.4 The WHILE Statement Modifier

The form:

<statement> WHILE *<condition>*

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

```
10 LET X=X↑2 WHILE X↑2<1E6
```

is equivalent to:

```
10 LET X=X↑2
15 IF X<1E6 THEN 10
```

The WHILE modifier (and the UNTIL modifier in Section 8.7.5) operates usefully only in iterative loops where the logical loop structure modifies the values which determine loop termination. This is a significant departure from FOR loops, in which the control variable is automatically iterated; a WHILE statement need not have a formal control variable. The following infinite loops never terminate:

```
10 X=X+1 WHILE I<1000
15 PRINT I,A(I) WHILE A(I)<>0
```

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

A successful application of the WHILE modifier is shown below:

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

As in the UNLESS modifier case, use the GOTO statement explicitly to avoid an error with the WHILE modifier.

8.7.5 The UNTIL Statement Modifier

The form:

<statement> UNTIL *<condition>*

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

```
10 X=X+1 UNTIL X<>SQR(X↑2)
```

is the same as:

```
10 X=X+1
20 IF X=SQR(X^2) THEN 10
```

As in the UNLESS modifier case, use the GOTO statement explicitly to avoid an error with the UNTIL modifier.

8.7.6 Multiple Statement Modifiers

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

```
10 LET A=B IF A>0 IF B>0
```

which is equivalent to:

```
10 IF B>0 THEN IF A>0 THEN A=B
```

or

```
10 IF B>0 AND A>0 THEN LET A=B
```

or

```
10 IF B<=0 THEN 40
20 IF A<=0 THEN 40
30 LET A=B
     .
     .
     .
```

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

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

which is equivalent to:

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

and to:

```
50 FOR I=1 TO N
55 FOR J=1 TO M
60 READ A(I,J)
65 NEXT J
70 NEXT I
```

Also see Section 8.7.3 which described the interaction of FOR and IF modifiers.

8.8 SYSTEM FUNCTIONS AND STATEMENTS

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 8-2.

Table 8-2

SYSTEM FUNCTIONS

Function	Meaning	Sample Usage
DATE$(Ø)	returns the current day, month and year, in the form: 2-Mar-72 Note that the date contains both upper and lower case characters (where lower case is not available on some terminals, only upper case letters are used but the ASCII values imply lower case).	PRINT DATE$(0) 10-AUG-72 READY
DATE$(N)	returns a character string corresponding to a calendar date. The formula used to translate between N and the date is as follows: (day of year)+[(number of years since 1970)*1ØØØ] DATE$(1)="Ø1-Jan-7Ø" DATE$(2Ø6Ø)="29-Feb-72"	155 PRINT X%(I),DATE$(I)
TIME$(Ø)	returns the current time of day as a character string as follows: TIME$(Ø)="Ø5:3Ø PM" or"17:3Ø "	75 IF TIME$(0) >= "05:45 PM" THEN PRINT "TIME TO QUIT"
TIME$(N)	returns a string corresponding to the time at N minutes before midnight, for example: TIME$(1)="11:59 PM" or "23:59" TIME$(144Ø)="12:ØØ AM" or "ØØ:ØØ" TIME$(721)="11:59 AM" or "11:59" N must be less than 1441 to return a valid string.	PRINT TIME$(1) 11:59 PM READY PRINT TIME$(1400) 12:40 AM READY
TIME(Ø)	returns the clock time in seconds since midnight.	25 IF TIME(0)>43200 THEN PRINT "AFTERNOON"
TIME(1)	returns the central processor (CPU) time used for this job in Ø.1 second quanta.	10 IF TIME(1)>30 THEN STOP
TIME(2)	returns the connect time (time during which the user has been logged into the system) for this job in minutes.	10 IF TIME(2)>1000 THEN STOP
TIME (3)	returns the number of kilo-core ticks (kct's) used by this job.	80 PRINT TIME(3)
TIME(4)	returns the device time for this job in minutes.	40 IF TIME(4)/60>2.5 THEN 90

Table 8-2 (Cont.)

Function	Meaning	Sample Usage
SWAP%(I%)	causes a byte swap operation to occur on the integer variable I%; returns the value of I% with the bytes swapped.	**1Ø PRINT CHR$(SWAP%(I%))**
RAD$(I%)	converts an integer to a 3-character string. This function is used to convert a value (expression in Radix-5Ø format) back into ASCII. Radix-5Ø is explained in Appendix D.	**55 PRINT RAD$(I%)**

There are also two special system statements that can be used within a BASIC-PLUS program: SLEEP and WAIT. Both statements allow the user to suspend his program for a stated interval.

The SLEEP statement is of the form:

line number SLEEP <*expression*>

SLEEP is used to dismiss the currently running program for the number of seconds indicated by the expression. At the end of this period the program is again runnable. Thus, the user is guaranteed at least this number of seconds idle time, possibly slightly more depending upon the number of jobs currently active on the system.

To awaken a job from a sleep before the specified number of seconds has expired, type a delimiter (RETURN, LINE FEED, FORM FEED or ESCAPE) at any of the job's terminals. The program segment shown below, however, can be used to override line terminating delimiters and provide a continuous SLEEP for a specified time.

```
1ØØ  T=TIME(Ø)
11Ø  SLEEP T+3Ø-TIME(Ø):
     IF TIME(Ø)-T<3Ø GOTO 11Ø
12Ø  INPUT X
```

In the above program, the INPUT statement is executed only if the time elapsed is equal to or greater than 3Ø seconds. Otherwise, if a delimiter is typed, the SLEEP is executed again for the length of time remaining in the original 3Ø seconds or until another line terminating character is typed.

A job is also awakened when it has declared itself a receiver and a message is queued for it through the SEND/RECEIVE system function calls. (The SEND/RECEIVE system function calls are privileged and are therefore documented in the System Manager's Guide.)

The WAIT statement is of the form:

line number WAIT <*expression*>

WAIT is used to set a maximum period for the system to wait for input from the user keyboard. If no delimiter is typed at the keyboard (RETURN, LINE FEED, ESCAPE) within the number of seconds specified by the expression, the program is restarted and a WAIT EXHAUSTED error occurs, which can be detected using ON ERROR-GOTO. The WAIT statement is used in conjunction with the INPUT statement. As an example:

```
LISTNH
10 ON ERROR GOTO 100
20 WAIT 15
30 INPUT "16 + 16 =";A
40 WAIT 0
50 IF A=32 THEN PRINT "RIGHT!"
         ELSE PRINT "NO, TRY AGAIN": GOTO 10
70 STOP
100 IF ERR<>15 THEN ON ERROR GOTO 0
110 PRINT "WAKE UP!"
120 RESUME 30
130 END

READY

RUNNH
16 + 16 =? WAKE UP!
16 + 16 =? 21
NO, TRY AGAIN
16 + 16 =? 32
RIGHT!
STOP AT LINE 70

READY
```

In this example line 1ØØ is executed only if the user fails to respond within 15 seconds. The use of WAIT Ø restores the terminal to its normal state in which no timeout occurs, but rather the system waits until a line is entered, however long that may take.

PART III

BASIC-PLUS DATA HANDLING

This part of the manual contains a complete description of all BASIC-PLUS data handling operations. A brief review is made of the simple forms of READ, DATA, PRINT, RESTORE and INPUT along with the more advanced forms of these statements. Formatted ASCII files, virtual core matrices, and Record I/O operations are described.

Advanced users are advised to consult the RSTS/E Programming Manual for a discussion of advanced data handling techniques and device dependent operations.

CHAPTER 9

DATA STORAGE CAPABILITIES

9.1 FILE STORAGE

Previously, techniques have been presented for entering data into a program as the program is written (via READ and DATA statements or from the user terminal while the program is executing (via the INPUT statement). Both techniques are inefficient when the amount of data to be read or written increases beyond a few items. In order to improve operation, BASIC-PLUS provides the user with facilities to define and manipulate Input/Output data files.

A BASIC-PLUS data file consists of a sequence of data items transmitted between a BASIC program and an external Input/Output device. The external device can be the user terminal, some other terminal, disk, line printer, card reader, magnetic tape device, DECtape, or high-speed paper tape equipment.

Each data file has both an external name by which it is known within the RSTS system (the name of the file on a disk storage device, for example) and an internal file designator (a channel number used to reference the file). An OPEN statement (see Section 9.2) is used to associate an external file specification with an internal file channel.

An external file specification contains some or all of the following information:

device:[proj,prog]filename.extension<protection>

If the *device* designator is not present in a file specification, the system device public structure is assumed. For non-file structured devices, only the *device* designator need be specified; any *filename, extension, project-programmer* codes, and *protection* code specified are ignored. Refer to Table 9-1. Where a *device* designator appears, it can be one of the following:

Table 9-1
Device Designations

Device Designation	Device
File structured Devices[1]	
DF:, DK:, DP:, DB:, or SY:	RSTS public disk structure as a whole
SYØ:	System disk
DFØ:	RF11 disk
DKØ: to DK7	RK11/RKØ5 disk cartridge units Ø through 7
DPØ: to DP7:	RP11/RPØ2/RPØ3 disk pack units Ø through 7
DBØ: to DB7:	RPØ4 disk pack units Ø through 7
DTØ: to DT7:[2]	DECtape units Ø through 7
MTØ: to MT7:[2,3]	Industry compatible magnetic tape units Ø through 7
Non-File Structured Devices	
PRØ:[2]	High-speed paper tape reader
PPØ:[2]	High-speed paper tape punch
LPØ: to LP7:[2]	Line printer units Ø through 7
CRØ:[2]	Card reader
KB:	Current user terminal
KBØ: to KB63:	Other user terminals on the system
DXØ: to DX7:	Floppy disk units Ø through 7

[1]DECtape, magtape, and user disks can also be used as non-file structured devices.
[2]Except for KB: and disks, device designation XX: is equivalent to XXØ:.
[3]MMØ: to MM7: are synonyms for MTØ: to MT7: when the magtape is a TMØ2/TU16.

For file-structured devices, each file is assigned a *filename* and *extension*. The *filename* is a string of one to six alphanumeric characters. The filename *extension* consists of a dot (.) followed by a one to three alphanumeric character string, usually specifying the file type. A null or blank *extension* is permitted, in which case the dot and filename extension field are omitted from the file designation. Refer to Table 9-2. The extensions recognized by the RSTS-11 system are as follows:

Table 9-2
Reserved File Extensions

Extension	Significance	Automatically Appended on Output	Assumed on Input
.BAS	indicates a BASIC-PLUS source program to be compiled; stored in ASCII format.	to BASIC-PLUS source programs stored with a SAVE or REPLACE command.	by the OLD command, also assumed by RUN, CHAIN and UNSAVE in the absence of a .BAC file of the same name.
.BAC	indicates a compiled BASIC-PLUS program; stored in a binary format, cannot be altered.	to BASIC-PLUS programs on which a COMPILE command is performed.	by the RUN, CHAIN and UNSAVE commands.
.TMP	indicates a temporary BASIC-PLUS file. These files are used while creating or editing a BASIC program. They are deleted when no longer needed.	no.	no.

The [*proj,prog*] field (containing the project and programmer numbers) identifies the owner of the file. If it is omitted the owner is assumed to be the current user. This field is meaningful only for disk and magtape files; it has no significance for DECtape files or files on non-file structured devices. The two numbers forming the field are decimal numbers between Ø and 254, separated by a comma, and enclosed in square brackets or parentheses.

NOTE

The PDP-11 DOS/BATCH Monitor uses octal UIC values in the range 1,1 to 376,376. Transferring magtape files between RSTS and DOS/BATCH causes an effective decimal-to-octal conversion between RSTS project-programmer number and DOS/BATCH UIC code. RSTS DECtape files are assigned a [1,1] UIC code.

Use of the $ character (dollar sign) in the project-programmer field indicates that the file is stored under the system library account ([1,2]).

When creating a disk file (with OPEN or OPEN FOR OUTPUT, see Section 9.2) or renaming a file (with the NAME AS statement, see Section 9.4) a *protection* field can be specified. Files can be read and/or write protected against three classes of users where distinctions are made on the basis of the project and programmer number of the user attempting to access the file. The three classes of users are:

a. owner;
b. group, all users having the same project number as the owner (termed the owner's group); and
c. others, all other users not in the owner's group

Table 9-3 is used to determine the value of the *protection* code to achieve the desired file protection.

Table 9-3
Protection Codes

Code	Meaning
1	read protect against owner
2	write protect against owner
4	read protect against owner's project
8	write protect against owner's project
16	read protect against all others who do not have owner's project number
32	write protect against all others who do not have owner's project number
64	compiled, run-only files
128	privileged program

Protection codes are stored within the system as character strings and consist of a decimal number within paired angle brackets. The decimal number is the sume of the desired combination of protection code values contained in Table 9-3. For example: a protection code of <48> denies read or write access to anyone logged into the system under an account number whose project number differs from the owner. The code < 48 > is the sum of 32 (write protect against all others) and 16 (read protect against all others). Similarly, the code <42> protects a file against <u>any</u> write operations (32=write protect against all others, 8=write protect against other project members, and 2=write protect against owner, 42=32+8+2).

The value 64 denotes a compiled file. It is added to the protection code by the system upon compilation. A protection code of <124> denies read or write access to everyone but the user. It is the sum of 64 (compiled file), 32 and 16 (write and read protect against all other projects), and 8 and 4 (write and read protect against others in owner's project). A protection code of <104> allows all users to run the file only (write access is denied), because the read protect values, 16 and 4, do not pertain.

The value 128 is used only with compiled files to designate a privileged program.[1] A protection code of <232>, for example, denotes a privileged program which can be run but not altered by all users. Only privileged users can assign protection codes which include the value 128.

Protection codes are normally specified only in the NAME-AS statement which allows the user to change the name and protection oode of any file which he has previously created (see Section 9.4). However, protection codes can be specified as an optional part of any filename. For example,

```
        OPEN "FILE.EXT<40>" AS FILE 1%
```

The file FILE.EXT is created under the current account with a protection code of <4Ø> .

[1]See the <u>RSTS/E Programming Manual</u> for a discussion of privilege.

9.1.1 Auxiliary Libraries

By specifying a certain special character, the user can designate an auxiliary library account on the system. This feature operates in a manner similar to the $ character signifying the system library account [1,2]. The following special characters designate the related auxiliary library accounts.

Character	Account
! (CHR$(33))	[1,3]
% (CHR$(37))	[1,4]
& (CHR$(38))	[1,5]
# (CHR$(35))	[proj,Ø]
@ (CHR$(64))	Assignable account

The user can refer to an account by using a character in any location where a project-programmer field is valid. Thus, a special character appearing in a file specification means that the file is stored in the related auxiliary library account.

The # character is unique because the system interprets it according to the account under which the user is running. For example, if the user is running under account [1Ø,2Ø] and specifies the # character, the system interprets it to mean account [1Ø,Ø]. This feature allows each project on the system to have its own library of files.

How to assign an account to the @ character is explained in Section 2.7.3 of the RSTS-11 System User's Guide.

9.1.2 Public and Private Disks

The concept of device names for disks was introduced with little explanation of when a disk is to be referenced by name (e.g., DK2:) and when simply by default. To clarify this, the concept of public disks and private disks must be explained.

A private disk is one that belongs to only a few user accounts, conceivably to a single user account. Files can be created under these accounts if the user account is one of the accounts on the disk. Also, files created under these accounts on a private disk can be read (or written) by other users only if the protection code of the file permits. A user who does not have an account on a private disk cannot create a file on it.

A public disk, on the other hand, is a disk on which any user can create files. Every user has an account on a public disk as soon as he references it. There is always at least one public disk, called the "system disk", on the system. All public disks together on a system are called the "public structure" because the system treats all of the public disks as one unit. For example, when a program creates a file in the public structure, that file is placed on the public disk which has the most unused space. This ensures proper distribution of files across the disks in the public structure. The actual determination of which disks on a particular system are public and which are private is left to the system manager. Therefore, this allocation varies from system to system.

Private disks are always referenced by a specific name (DK2:, for example). The public structure is normally referenced by default, although it has the specific name "SY:". The system disk is referenced explicitly by the name "SYØ:" and may be on any unit. Other disks on the system, whether public or private, are termed non-system disks. Referencing public disks by their specific names is not recommended: a file that exists elsewhere in the public structure might not be found or might even be deleted. The system does not allow a single user to store two files in the public structure under the same name.

Private disks may be mounted and dismounted while the system is running. Normally, private disks are loaded only when needed, but public disks should be kept permanently mounted so that users can access all files during time sharing.

9.2 <u>OPEN STATEMENT</u>

The OPEN statement associates a file on a file-structured device or some non-file structured device with an I/O channel number internal to the BASIC program. BASIC-PLUS permits up to 12 files to be open at a given time, and therefore, permits internal file designators to be integers between 1 and 12.

The general form of the OPEN statement is as follows:

line number OPEN <*string*> $\left[\begin{matrix}\text{FOR INPUT}\\\text{FOR OUTPUT}\end{matrix}\right]$ AS FILE <*expression*>

One or more of the following specifications can be appended to the end of the statement (and are described in Sections 9.2.1 through 9.2.4).

[,RECORDSIZE <*expression*>] [,CLUSTERSIZE <*expression*>]

[,FILESIZE <*expression*>] [,MODE <*expression*>]

The *string* field is a character string constant, variable or expression that contains the external file specification (as described in Section 9.1) of the file to be opened. The AS FILE *expression* must have an integer value between 1 and 12, corresponding to the internal channel number on which the field is being opened.

There are three distinct forms for the OPEN command:

OPEN<*string*> FOR INPUT
OPEN<*string*> FOR OUTPUT
OPEN<*string*>

The form of the OPEN statement used determines whether an existing file is to be opened or a new file created.

a. An OPEN FOR INPUT statement causes a search for an already existing file (since the statement indicates the file is an input file). If no file is found, the CAN'T FIND FILE OR ACCOUNT error occurs.

```
50 OPEN "FILE.DAT" FOR INPUT AS FILE 1
```

b. An OPEN FOR OUTPUT statement causes a search for an already existing file which, if found, is deleted. A new file is then created.

```
75 OPEN "DATA.01<40>" FOR OUTPUT AS FILE 3
```

c. An OPEN statement without an INPUT or OUTPUT designation attempts to perform an OPEN FOR INPUT operation as described above. If this fails, a new file is created.

```
100 OPEN "MATR.TER" AS FILE 7
```

The OPEN statement does not control whether the program attempts to perform input or output on the disk file or whether read and/or write access to the file is granted[1]; these privileges are controlled by the file protection code.

If an assignable device (all devices other than disks are available or assignable to a single user at any given time) is referenced in any OPEN statement and that device is already in use by another user, a DEVICE NOT AVAILABLE error occurs.

When used with disk files, an OPEN FOR INPUT or OPEN FOR OUTPUT allows either read or write operations on the opened file. The system allows write access to a file if the protection code permits and if no other user has write access to the file. For example, if user 1 opens a file, he has read and write access. When user 2 opens the same file, he has read access only; a PROTECTION VIOLATION error occurs when he attempts to write on that file. When user 1 subsequently closes the file, no user has write access until the next open operation. User 3 can now open the file and obtain both read and write access, because no other user currently has write access to that file. On DECtape and magnetic tape devices, the FOR INPUT and FOR OUTPUT clauses restrict operations on that file to the type of operation specified.

NOTE

> Only one user can have write access to a file at a single time (unless MODE 1% is used, see Section 10.5.1); and user write access is always denied to a file with a .BAC extension, since compiled files can only be run.

[1]Magtape and DECtape are exceptions to this rule, see the RSTS/E System Programming Manual.

The next four sections in this manual describe the RECORDSIZE, CLUSTERSIZE, FILESIZE and MODE options of the OPEN statement. As these are sophisticated file handling tools, it is suggested that the novice user initially skip these sections and continue with Section 9.2.5.

9.2.1 RECORDSIZE Option

When any file is opened, the system creates a buffer area in the user's core space to buffer all I/O to and from the file. Normally the amount of space reserved is determined by the device, as each device has a default device buffer size as described in Table 9-4.

Table 9-4
Default Device Buffer Size

Device	Default Device Buffer Size
All disks	512 characters (or bytes)[1]
Floppy disk (DXn:)	128 characters (or bytes)
DECtape (DTn:)	510 characters (or bytes)[1]
Magtape (MTn:)	512 characters (or bytes)
High-speed reader (PR:)	128 characters (or bytes)
High-speed punch (PP:)	128 characters (or bytes)
Line printer (LP:)	128 characters (or bytes)
Card reader (CR:)	160 characters (or bytes)
User terminal (KB:)	128 characters (or bytes)

[1]The default buffer size may differ when the device is used as a non-file structured device.

With the RECORDSIZE option, the user program can allocate more buffer space than is provided by the default case. However, in some cases the particular device driver may not permit additional space to be used. Table 9-5 shows the buffer size alterations for specific devices.

Table 9-5
Use of RECORDSIZE

Device	Possible Buffer Alterations
Disk	The disk drivers permit use of any buffer size that is an integral multiple of 512 bytes.
DECtape	The DECtape driver uses only the first 510 bytes of the available buffer space (512 bytes for non-file structured DECtape).
Magtape	The magtape driver uses only enough bytes for one physical magtape record. 14 bytes ≤ physical record ≤ buffer size.
High-speed reader High-speed punch Line printer User Terminal	These non-file structured devices can use any selected buffer size greater than the default size.
Card reader	The card reader driver uses only enough bytes for one card's data.
Floppy disk	The floppy disk driver permits use of any buffer size that is an integral multiple of 128 bytes.

The RECORDSIZE option has significant advantages when used with magtape and disk files. RECORDSIZE permits non-file structured access to magtape records of any length. On a disk file, total throughput can be improved by using a larger buffer size, as this permits a single disk transfer to read a large quantity of data. Specify only an even number of bytes in the RECORDSIZE expression. For example:

```
    1ØØ OPEN "MASTER.DAT" FOR INPUT AS FILE 1%, RECORDSIZE 2Ø48%
```

If the file MASTER.DAT were on an RF11 disk and occupied a contiguous area on that disk, a 2Ø48-byte transfer would take about 33ms while four 512-byte transfers would take about 83ms (on the average). If the file did not reside in a contiguous disk area, the RSTS Monitor would break the 2Ø48-byte transfer into four 512-byte transfers. Even in this last case, the system overhead to perform the transfer would be less.

This example raises the question of how to ensure that a file occupies a contiguous disk area. This can be done by means of the CLUSTERSIZE option described in the following Section.

9.2.2 CLUSTERSIZE Option

The CLUSTERSIZE option is applicable only to disk files and only when these files are initially created with an OPEN or OPEN FOR OUTPUT statement. The CLUSTERSIZE specification is ignored if this is not the case.

The RSTS system divides each disk into a number of 256-word blocks. Each block is assigned a unique physical block number starting at 1[1]. Physical block numbers are assigned such that block n is physically contiguous with blocks n+1 and n-1.

A number of contiguous blocks taken together as a unit are called a cluster. RSTS permits clusters to be 1, 2, 4, 8, 16, 32, 64, 128 or 256 blocks long. When the disk is initialized (the process by which the disk is cleared for use on RSTS) a minimum cluster size can be established. This minimum cluster size (also called the pack cluster size) can be 1, 2, 4, 8, or 16 blocks.

For each file on the system, an entry is made in the owner's file directory (User File Directory or UFD) containing the filename, cluster size for the file, and a sequential list of blocks belonging to that file.

A UFD has a fixed maximum size which is determined when the UFD is created[2]. A UFD on any one disk cannot exceed 112 (decimal) blocks (28,672 words). If all files were a minimum size (7 or fewer clusters long) a UFD clustered as 16 would have room for a maximum of 1157 files. To keep the list of blocks belonging to the file as short as possible, the UFD contains a one-word entry for the first block of each cluster. Knowing the first block number of the cluster and the number of blocks in the cluster is sufficient to determine all of the blocks in the cluster.

[1]Block Ø of each disk is reserved for a bootstrap record and is not used by any file.

[2]The maximum size of a UFD is seven times the cluster size for that UFD, which is established when the account is created, and may be 1, 2, 4, 8 or 16 blocks. The figures given in the text assume a UFD cluster of 16.

Because of the size limit on the UFD, large files benefit from the specification of large cluster sizes. In an extreme example, the UFD would be completely filled by a single file of 24,283 blocks where the file cluster size is one block. However, with a cluster size of 256 blocks, only 128 words of the UFD are required to describe this file.

Since most user files are not extremely large, omitting the CLUSTERSIZE option when creating the file makes little practical difference. Omitting the CLUSTERSIZE option has the effect of assigning a cluster size equal to the pack cluster size for the disk on which the file resides.

Once a file is opened on an internal I/O channel, all I/O requests by the BASIC program are handled by means of a read or write call from BASIC-PLUS to the Monitor, directed to the nth logical block of the file. The RSTS system translates the logical block number into a physical block number. This is done by reading the list of physical clusters belonging to the file (as kept in the UFD) and finding the entry corresponding to the nth logical block. To minimize the overhead involved in reading the UFD, which is stored on the disk, part of this list of clusters belonging to a file is kept in memory. This part of the list is called the file window. The file window is composed of seven entries from the list of file clusters. Since each entry corresponds to one cluster of the file, with a file cluster size of one block, 7 blocks (or 1792 words) of the file are described by the in-core file window. These 7 blocks can then be read or written without accessing the complete list from the UFD stored on the disk. Similarly, with a file cluster size of 256 blocks, the file window describes the location of 1792 blocks of the file, or over 45Ø,ØØØ words. When performing random access I/O to virtual core arrays and RECORD I/O files, any of the 1792 blocks would be read or written without referencing the UFD.

As an example of the use of the CLUSTERSIZE option:

```
100 OPEN "MAT.DAT" FOR OUTPUT AS FILE 1%, CLUSTERSIZE 128%
```

In this case the file MAT.DAT is created with a cluster size of 128 blocks. Note that the file is initially 128 blocks long and is extended as needed in 128-block increments.

Since files with large cluster sizes must be extended by a whole cluster at a time and since clusters are always contiguous blocks, it may not always be possible to find sufficient contiguous free blocks to extend the file. The user should be aware of this possibility whenever he creates a file with a cluster size larger than the pack cluster size (the minimum cluster size for that disk).

As another example (typing LINE FEED following FILE 1%,):

```
100 OPEN "DATA" FOR OUTPUT AS FILE 1%,
RECORDSIZE 2048%, CLUSTERSIZE 4%
```

The RECORDSIZE option improves disk throughput when multiple blocks can be read or written in a single transfer (see Section 9.2.1). By creating the file with a cluster size of 4 (1Ø24 words or 2Ø48 characters per cluster) the user guarantees that logical blocks 1-4, 5-8, etc., of his file are physically contiguous on the disk.

9.2.3 FILESIZE Option

A disk file (and only a disk file) can be pre-extended by using the FILESIZE option in an OPEN statement, eliminating the need for a PUT statement. The format for the FILESIZE option is:

[*line number*]OPEN <*string*>[FOR OUTPUT] AS FILE<*expr*>[,FILESIZE <*expr*>]

For example:

```
100 OPEN "VALUES" FOR OUTPUT AS FILE 3%, FILESIZE 50%
```

The data file, VALUES is opened and automatically pre-extended to 50 256-word blocks.

9.2.4 MODE Option

The OPEN statement allows another option: the MODE field. The format of the OPEN statement, including the MODE field, is as follows:

[*line number*] OPEN <*string*> $\left[\begin{array}{l}\text{FOR INPUT}\\ \text{FOR OUTPUT}\end{array}\right]$ AS FILE <*expr*>

[,RECORDSIZE <*expr*>] [,CLUSTERSIZE <*expr*>] [,MODE <*expr*>]

The MODE option is used to establish device-dependent properties of the file. MODE Ø% is assumed by the system when MODE is omitted. For disk files, MODE indicates that the file is to be updated or appended (see Section 10.5). For non-file structured magtape operations, MODE establishes the density and parity settings for magtape. For line printer operations, MODE is used in conjunction with the optional forms control to establish the current form length. For card reader operation, MODE sets a read mode to correspond to specific data card formats.

9.2.5 File Structured Vs. Non-File Structured Devices

RSTS/E distinguishes between file structured (disk, DECtape and magtape) devices and non-file structured devices. When a file is to be found or created on a file structured device, the file specification string in the OPEN statement must include both a device designation (or default public structure) and a filename. On non-file structured devices, the device name alone identifies a file (filename and extension, if specified, are ignored). For example:

DTØ:	is insufficient information to specify a file.
DTØ:FRED	is sufficient to specify the file FRED on DECtape unit Ø.
PP:	uniquely specifies the high-speed punch.
PP:FILE	specifies a file on the high-speed punch, the filename is ignored.
DX1:	uniquely specifies floppy disk unit 1:

File specification syntax is such that the default device (the public disk storage area) need not be specified. For example:

```
    SY:QUIZ
```

is equivalent to:

```
    QUIZ
```

It is also possible to open a file structured device in non-file structured mode. For example:

```
    OPEN "DK2:" AS FILE 5%
```

is sufficient to open a disk cartridge in non-file structured mode.

9.3 CLOSE STATEMENT

The CLOSE statement is used to terminate I/O between the BASIC program and a peripheral device. Once a file has been closed, it can be reopened for reading or writing on any internal file designator.

All files must be closed before the end of program execution. Execution of a CHAIN statement automatically closes any open files, but does not cause the output of the last blocks to output files. The format of the CLOSE statement is as follows.

line number CLOSE <*expression*> [,<*expression*>...]

The *expression* indicated has the same value as the *expression* in the OPEN statement and indicates the internal channel number of the file to close. Any number of files can be closed with a single CLOSE statement; if more than one file is to be closed, the *expressions* are separated by commas. For example:

```
    255 CLOSE 2,4
    345 CLOSE 10
```

Line 255 above closes the files opened on internal I/O channels 2 and 4. Line 345 closes the file open on internal I/O channel 10.

9.4 NAME-AS STATEMENT, FILE PROTECTION AND RENAMING

The NAME-AS statement is used to rename and/or assign protection codes to a disk or DECtape file, and can only be used on a given file by someone logged into the system under the account number which owns the file. The format of the statement is as follows:

line number NAME<*string*>AS<*string*>

The specified file (the first *string* indicated) is renamed (as the second *string* indicated). When the file resides on a device other than the default device (system disk), the device must be specified in the first *string* and may optionally be specified in the second *string*. No filename extension assumptions are made by NAME-AS; the filename extension must be specified in both strings if any extension is present in the old filename or desired in the new filename. For example:

```
75 NAME "DT0:OLD.BAS" AS "NEW.BAS"
```

is equivalent to:

```
75 NAME "DT0:OLD.BAS" AS "DT0:NEW.BAS"
```

but the statement:

```
90 NAME "FILE1.BAS" AS "FILE2"
```

is not advised since FILE2 has no extension and could not subsequently be called into core via the OLD or RUN commands (which require filename extensions).

A file protection code can be specified within typed angle brackets as part of the second <*string*> although it is not required. If a new file protection code is specified, it is reflected in the protection assigned to the renamed file. If no new protection code is specified, the old protection code is retained. See Section 9.1 for a complete description of protection codes.

```
100 NAME "FILE.EXT" AS "FILE.EXT<40>"
```

changes only the protection code of the file FILE.EXT stored on the system disk.

```
200 NAME "DT0:ABC.BAS" AS "XYZ.BAS"
```

changes the name of the file ABC.BAS on DECtape unit Ø. Since no transfer of the file from one device to another can be performed with the NAME-AS statement, it is not necessary to mention DTØ: twice; that is, the device of the new filename need not be specified. However, an error is generated if a device other than the old device is specified.

```
120 NAME "NEW" AS "NEW1"
```

changes only the name of the disk file NEW. (To transfer a file between devices, use the PIP system program described in the RSTS-11 System User's Guide.)

9.5 KILL STATEMENT

The KILL statement is of the form:

$\left\{ \textit{line number} \right\}$ KILL <*string*>

and causes the file named *string* to be deleted from the user's file area. (The file can no longer be opened, but if it is already open the file remains available until it is closed.) For example, when the user has completed all work with the file XYZ (note that the filename has no extension) on the system disk, he could remove the file from storage by executing the following statement:

```
455 KILL "XYZ"
```

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.)

The KILL (and NAME-AS) statement can be issued in immediate mode. It should be noted that KILL is more general than UNSAVE, which is primarily used to delete source (.BAS) files (see the RSTS-11 System User's Guide). KILL can be used to delete any file, including a file with a null extension (which the UNSAVE command cannot delete).

9.6 CHAIN STATEMENT

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 separate programs. Such programs are called into core for execution by means of a CHAIN statement. Each program section is assigned a name and control can be transferred between any two programs. A CHAIN statement is of the form:

line number CHAIN <*string*> [<*line number*>]

and causes the program named by the *string* to be called, compiled (if necessary), and executed. The *line number*, if specified, designates the line at which the program is to be started. If the *line number* is omitted, the program is started at the lowest numbered line (as though a RUN command had been used). The CHAIN statement is the last statement executed in each program segment other than the last segment. For example:

```
1000 CHAIN "MAIN.BAC" 2000
```

causes the program MAIN.BAC to be loaded and started at line 2ØØØ. Notice that a filename extension is not required. The compiled form of the program is searched for and, if found, run. If the compiled form is not found, the non-compiled form is searched for and, if found, compiled and run. If neither form of the program is found, an error occurs.

On the other hand, if a filename extension is specified, and not found, an error occurs; in this case, no other form of the program is searched for.

Chaining to precompiled program files (.BAC files) is considerably more efficient that chaining to BASIC source program files since .BAS files require compilation upon each call.

Communication between chained programs is performed by means of user's files or core common.

When the CHAIN statement is executed, all open files for the current program are closed, the new program segment is loaded, and execution continues. Any files to be used in common by several programs should be opened in each program.

The CHAIN statement also implicitly closes all open I/O channels, which is slightly different from the actions performed as a result of a CLOSE statement. For example, the line printer drivers perform two top of form operations when the printer is closed with a CLOSE statement. To continue printing on the same piece of paper, do not execute a CLOSE statement on the line printer channel; the CHAIN statement is sufficient to close the printer without performing top of form operations.

Similarly, an explicit close of the paper tape reader clears the input buffers and prepares for a new tape. The implicit close performed by the CHAIN statement does not clear the buffers and the program subsequently chained may resume where the previous program stopped. An explicit close of the paper tape punch causes a trailer to be punched; the implicit close does not.

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. Data values must correspond in type with their respective variables, but the "%" character should not be included in integer values. Integer and floating point values are interchangeable, although they are stored according to the type of the variable. The use of quote marks is discussed below.

The data pool consists of all DATA statements in a program. Values are read starting with the DATA statement having the lowest line number and continuing to the next higher, etc. The location of DATA statements in a program is irrelevant, although for simplicity they are usually kept together toward the end of the program. (The DATA statements must occur in the proper numeric sequence, however.) A DATA statement must be the only statement on a line, although a READ statement can occur anywhere on a line. Comments are not permitted at the end of a DATA statement. If a READ statement is unable to obtain further data from the data pool, an error message is printed and program execution is terminated. (This error can be treated through the ON ERROR GOTO statement, Section 8.4.)

Quotes are necessary in DATA statements only around string items which contain a comma, when leading, trailing or embedded blanks within the string are significant, or when lower case letters are to be preserved. The data pool, composed of values from the program's DATA statements, is stored internally as an ASCII string list. When a numeric variable is read, the appropriate ASCII to numeric conversions are performed. When a string variable is read, the string is used as it appears in the DATA statement. If the item does not appear in quotes, then leading, trailing, and embedded spaces are ignored. If the item appears in quotes, the string variable is equated to the entire string within the quotes.

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

line number MAT READ < *matrix* >

This reads the value of each element of a predimensioned matrix from the data pool. Each element in the list of matrices indicates the maximum dimension of the matrix to be read (which cannot be greater than the dimensioned size of the matrix). Individual elements are separated by commas. For example:

```
10 DIM A(20,20),B(50)
20 MAT READ A
30 MAT READ 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 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 3.3.1 for an example program using the RESTORE statement.

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

10.3 PRINT STATEMENT

In its simplest form, the PRINT statement:

line number PRINT

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. When an element is not a simple variable or constant, the expression is evaluated before a value is printed. The list can also contain character strings between quotes which are printed exactly as typed between quotes.

NOTE

If a character string is enclosed in a PRINT statement with an initial quote and no terminating quote, a terminating quote is considered to follow the last character of that PRINT statement. For example:

```
10 PRINT "NAME IS A$
10 PRINT "NAME IS A$"
20 PRINT "NAME IS " A$
```

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:

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

when executed causes the following line to be printed:

```
1  6            2        END
```

A terminal line is considered to be divided into five[1] 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 an integer or floating-point variable, function, or expression is followed by a semicolon, the value is printed with a preceding minus sign if the number is negative, or a preceding space if it is positive. The number is then followed by a single space.

b. character strings and string variables followed by a semicolon are printed with no preceding or trailing spaces.

Any PRINT statement which does not end with a semicolon or comma character causes a skip to the next line after printing the elements in the list. The presence of the punctuation character at the end of the PRINT list causes the next PRINT statement to continue on the same line under the conditions already defined.

In general, the output rules for the PRINT statement are:

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

[1]The actual number of print zones is INT (n/14), where n is the size of the print line.

b. at most six significant digits are printed, unless PRINT-USING is used.

c. most numbers are printed in decimal format. 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; omitting punctuation has no effect on the output format in this case.

10.3.1 Formatted ASCII I/O

BASIC-PLUS permits access to data files by three methods:

a. Formatted ASCII;

b. Virtual core arrays, described in Chapter 11;

c. RECORD I/O, described in Chapter 12.

Formatted ASCII data files are the simplest method of data storage, involving a logical extension of the PRINT and INPUT statements to be used in conjunction with the OPEN statement.

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

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

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

where the *expression* has the same value as the expression in the OPEN statement (the internal file designator) and the *list* is a list of variable names, expressions, or constants as explained in the sections describing the PRINT and INPUT statements.

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

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

where the expression is the number of a previously opened output file, out of 12 possible open files (see Section 9.2). For example:

```
10 OPEN "PP:" FOR OUTPUT AS FILE 3%
50 PRINT #3%, B,3.14,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, of 12 possible files. Of course, the above program segment assumes that the function FNX and the variables A and B are defined elsewhere in the program.

10.3.2 Output to Non-Terminal Devices

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

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

where the *expression* is the internal channel number (the internal file designator) of a previously opened output file (see Section 9.2). The *list* of information to be output can include any of the output information described as applicable to the PRINT statement. For example:

```
10 OPEN "DATA1" FOR OUTPUT AS FILE 7%
20 PRINT #7%, "START OF DATA FILE"
```

The above lines open a file called DATA1 on the disk with internal channel number 7 (of 12 possible open files available 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
10 LET I$="LP:"
20 OPEN I$ FOR OUTPUT AS FILE 1%
30 PRINT #1%, I,SQR(I) FOR I=1% TO 5%
40 END
READY
RUNNH
READY
```

The result would appear on the line printer as follows:

```
1              1
2              1.4121
3              1.73205
4              2
5              2.23607
```

10.3.3 PRINT-USING Statement

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 internal channel number of the file or device which is the destination of the output; the *string* is either a string constant, string variable, or string expression which is an exact image of the line to be printed. This string is called a format field. The list is a list of items to be printed in the format specified by the format field. All characters in the string are printed as they appear except for the special formatting characters and character combinations described on the following pages. The *string*, or portions of the *string*, are repeated until the *list* is exhausted. The *string* is constructed according to the following rules:

Exclamation Point

An exclamation point in the format field identifies a one character string field. The variable string is specified in the < list> within the PRINT statement. For example:

```
10 PRINT USING "!!!", "AB", "CD", "EF"
```

which causes:

```
ACE
```

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

String Field

A variable string field of two or more characters is indicated in the format field by spaces enclosed between backslashes. The backslash character (\) is produced by typing SHIFT/L on some keyboards. Enclosing no spaces indicates a field two columns wide, one space is equivalent to a field three columns wide, etc. For example:

```
20 PRINT USING "\\\  \", "ABCD", "EFGHI"
```

causes

```
ABEFGH
```

to be printed at the user's terminal. The first two backslashes have no spaces enclosed, hence permit the printing of two characters (AB). The second two backslashes enclose two spaces and permit the printing of four characters (EFGH). No spaces are printed unless specifically planned.

Numeric Field

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

```
30 PRINT USING "###.##", 12.345
```

causes

```
12.35
```

to be printed on the user's terminal, while

```
40 PRINT USING "####", 12.345
50 PRINT USING "#### ", 12.345
60 PRINT USING "##", 100
```

causes

```
     12
     12.
    % 100
```

to be printed on the user's terminal. Numeric fields are right justified; that is, if a number does not fill the allotted space, leading blanks precede the number. When the field specified is too small for a constant or variable to be printed, the % character is printed to indicate the error. The number is then printed without reference to the format field. On the other hand, when the format field specified is more than 20 character spaces larger than required for a constant or variable to be printed, a PRINT-USING BUFFER OVERFLOW non-recoverable error may occur.

If the format field specifies a digit as preceding the decimal point, at least one digit is always output before the decimal point. If necessary, that digit is zero.

<u>Asterisks</u>

If a numeric field designation in the format field begins with **, any unused spaces in the format field are filled with asterisks. For example:

```
10 A=27.95: B=107.50: C=1007.50
20 PRINT USING "**##.##", A,B,C
```

prints the following:

```
**27.95
*107.50
1007.50
```

Notice that the ** characters act as two additional # characters as well as allowing asterisk fill.

Exponential format (see below) cannot be used in a field with leading asterisks. Negative numbers cannot be output using asterisk fill unless the sign is output following the number (see below).

Exponential Format

When the exponential form of a number is desired, the numeric format field is followed by the string ↑↑↑↑ (four ↑ characters) which allocates space for E-xx. Any arrangement of decimal points is permitted. For example:

```
5 F$="##^^^^######"
10 A=10000.
20 PRINT USING F$,A,A
```

causes

```
10E 03 10000
```

to be printed at the user's terminal.

All format positions are used to output a number with an exponent. The significant digits are left justified and the exponent is adjusted.

Trailing Minus Sign

If a numeric format field designation is terminated with a minus sign, the sign of the output number is printed following the number, rather than preceding it. A blank is printed to indicate a positive number.

```
10 A=-10.5
20 PRINT USING "##.##- ####.##", A,A
```

which prints:

```
10.50-  -10.50
```

Note that if the trailing minus is not used, space must be reserved in the numeric format field for the sign to precede the number.

Dollar Signs

If a numeric format field begins with $$, a dollar sign is printed immediately preceding the first digit of the number:

```
10 A=77.44: B=304.55: C=2211.40
20 PRINT USING "$$##.##", A,B,C
```

which prints:

```
 $77.44
$304.55
% 2211.4
```
(insufficient space to print C along with $ character)

Note that the $$ characters provide for the printing of two additional characters in the number. Since one character is a $, the effect is to allow for one additional # designation beyond the ones typed by the user.

Exponential format (see above) cannot be used in a field with leading dollar signs. Negative numbers cannot be output using the floating dollar character unless the sign is output following the number (see above).

Commas

If one or more commas appear to the left of the decimal point (if any) in a numeric format field, then commas are inserted every three digits to the left of the decimal point. A comma to the right of the decimal point is considered a printing character.

For example:

```
10 PRINT USING "#,######.## ####.#,#", 12345.5,123.456,1
```

prints the following:

```
     12,345.50   12%.5,1
```

<u>Insufficient Format</u>

If insufficient format characters are present in a field when a number is output, a % character is printed in the first position of the field followed by the number in standard format, usually causing the field to be widened to the right. The user is guaranteed his entire number. For example:

```
     10 PRINT USING "##.## ##.##", 12 345, -12.5
```

prints the following:

```
     12.35 %-12.5
```

Rounding occurs when digits are dropped at the right of numbers. If rounding causes the number to exceed the format allowed, the % character is used. For example:

```
     10 PRINT USING " ##     ##", .125, .999
```

prints the following:

```
     13    % .999
```

<u>Format Too Large</u>

If a numeric format field results in an attempt to output more significant digits than are available for the number, zeros are substituted for all digits following the last significant digit. Six significant digits are available with the 2-word, single precision math package and fifteen digits with the 4-word, double precision math package. A PRINT-USING BUFFER OVERFLOW error may occur when a numeric format field results in an attempt to output more than 20 characters <u>before</u> the first significant digit of the number.

PRINT Statement Punctuation

When the PRINT-USING statement is used, the usual PRINT statement punctuation characters (commas and semicolons) have no effect on the output format, except that a semicolon at the end of the PRINT list inhibits termination of the printed line.

```
10 PRINT USING "##    ##  ##", 1;2,3
```

prints the following:

```
1     2   3
```

As another example:

```
10 PRINT USING "#.##", 2.5;
20 PRINT "X"
```

prints

```
2.50X
```

As another example:

```
10 LET A=1.32111: B=2.45457
15 LET F$ = "  A=##.##  B=##.##"
20 OPEN "LP:" FOR OUTPUT AS FILE 4
25 PRINT #4, USING F$, A,B
```

would cause:

```
A= 1.32  B= 2.45
```

to be printed on the line printer.

10.3.4 MAT PRINT Statement

The MAT PRINT statement allows for easy printing of a predimensioned matrix. The statement is of the form:

line number MAT PRINT [#<*expression*>,]<*matrix*>

for example:

```
15 DIM A(16)
25 MAT PRINT A(15)
```

If the specified matrix name is unsubscripted, the entire matrix is printed. If the matrix specification is subscripted, the subscript(s) indicates the maximum size of the matrix to be printed.

The matrix name can be followed by a semicolon to indicate that the values are to be printed in a packed fashion, or by a comma to indicate that each element is printed in its own zone. For example:

```
10 DIM A(10,10),B(10,20)
20 OPEN "LP:" FOR OUTPUT AS FILE 1


120 MAT PRINT #1, A;    !PRINT MATRIX A IN PACKED FORMAT
130 MAT PRINT #1, B(10,10),    !10*10 MATRIX IS PRINTED,
                               !5 VALUES PER LINE
```

Row and column matrices can also be printed. For example:

```
10 DIM A(5),B(10)
20 OPEN "LP:" FOR OUTPUT AS FILE 1
30 MAT PRINT #1, A;            !PRINT ON ONE LINE ON CHANNEL 1
40 MAT PRINT #1, B             !PRINT IN COLUMN FORMAT ON
                               !CHANNEL 1
```

Line 3Ø causes A to be printed as a row matrix, closely packed; line 4Ø causes B to be printed as a column matrix. The form:

```
70 MAT PRINT A,
```

would cause the matrix A to be printed as a row matrix, five values per line (at the user terminal).

10.3.5 PRINT Functions

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

Function	Meaning
POS(X)	Returns the current position on the output line; where X is the I/O channel number. POS(Ø) returns the value for the user's terminal.
TAB(X)	Tab to position X in the print record. For example, a standard terminal has 72 printable columns numbered Ø through 71. TAB (4) causes sufficient spaces to be output to move the print head to column 4. If the print head is currently past position 4, no spaces are output.

For example:

```
    10 PRINT "X";TAB(10);POS(0)
```

causes the following to be printed:

```
          X         10
position Ø↑⏟↑position 1Ø
          9 spaces
```

10.4 INPUT STATEMENT

The INPUT statement allows data to be entered to a running program from an external device, the user's keyboard, disk, DECtape, paper tape reader, etc. The full form for this statement is:

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 of string or numeric variables. If sufficient values are not typed, the system prints another ?; if too many values are typed, separated by commas, excess values are ignored. The user can also insert printed messages between the variables to be input. For example:

```
10 INPUT "YOUR NAME IS";N$,"ACCOUNT NUMBER";A;"THANK YOU"
```

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

```
YOUR NAME IS? CLYDE
ACCOUNT NUMBER? 555
THANK YOU
```

ON ERROR GOTO statements can be used in a program to trap recoverable errors which occur during an input sequence. The errors shown below occur most frequently when an INPUT statement is executed.

Error	Meaning	Examples
DATA FORMAT ERROR (ERR = 50)	Data input in an illegal form	3.4.5 or $2 or #16 or 2;3 or LORA input for a numeric variable; X" or "HELLO" "THERE" input for a string variable
ILLEGAL NUMBER (ERR = 52)	Overflow or underflow	3E+66 or --23
END OF FILE ON DEVICE (ERR = 11)	Input CTRL/Z	↑Z

The system assigns values to variables as they are input. Multiple variables can be assigned by separating them in the INPUT variable list by commas. Similarly, use commas or the RETURN key to separate values as they are input from the keyboard. For example:

```
10 INPUT X,Y,Z
20 PRINT X,Y,Z
RUNNH
? 3.14
? 14, 92
 3.14   14      92
READY
```

Do <u>not</u> use commas within a single number; the system ignores all characters input beyond a comma unless another variable is to be assigned. For example:

```
        Right                   Wrong

10 INPUT R              10 INPUT R
20 PRINT R              20 PRINT R
RUNNH                   RUNNH
? 25,902                ? 25902
 25                      25902
READY                   READY
```

Quotation marks (") should be used with string variables when an embedded comma is to be preserved. For example:

```
        Right                   Wrong

10 INPUT M$             10 INPUT M$
20 PRINT M$             20 PRINT M$
RUNNH                   RUNNH
 ? "MOUSE, MICKEY"      ? MOUSE, MICKEY
MOUSE, MICKEY           MOUSE
READY                   READY
```

Commas can be embedded without using quotation marks via the INPUT LINE statement, described below.

The format:

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

causes input to be read from the file or device indicated in the expression, by the internal file designation number given when the file was opened. (See Section 9.2 for a description of the OPEN statement.) If the value of the expression is non-zero and the specified file is the user terminal, open as an input device, then no ? character is printed at the terminal when input is requested.

For example:

```
75 OPEN "KB:" FOR INPUT AS FILE 2
80 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 printed on the terminal.

Another format of the INPUT statement allows the user to enter an entire line of data as a single character string entity, regardless of embedded spaces or punctuation. This is different from the normal mode of string input, where the comma, apostrophe, single quote and double quote characters have special significance. The format is:

line number INPUT LINE[# *expression* ,]<*string variable*>

For example:

```
25 INPUT LINE A$
```

pauses to allow the user to enter a line followed by the RETURN, LINE FEED or ESCAPE key (see also Section 5.3). Every character input, including quotation marks and commas, is present in A$, above. The end of the line being input is the carriage return/line feed sequence (or line feed/carriage return/null or ESCAPE, see Section 5.3) which is appended to the data typed by the user. To remove the CR and LF characters, use the string function LEFT, as described in Section 5.5. This can be done as follows:

```
G$ = LEFT (G$,LEN(G$)-2%)
```

END OF FILE ON DEVICE (ERR=11) occurs when CTRL/Z is input.

As another example:

```
20 OPEN "F2.DAT" 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) and to read a string of characters up to the next LINE FEED character. (See Table 9-4 for the size of buffers available for each device.)

10.4.1 MAT INPUT Statement

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

```
200 MAT INPUT A(20)
```

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

line number MAT INPUT[#<*expression*>,] *variable list*

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

```
45 DIM B(10,25)
50 OPEN "DT1:DATA1" FOR INPUT AS FILE 1
55 MAT INPUT #1, B(10,25)
```

The above lines cause the file DATA1 on DECtape 1 to be opened for input on channel 1 (of 12 possible channels) and a matrix of values for the elements of B to be read to fill B(10,25). The zero elements are not assigned a value. When input is from the user terminal, ? 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.

10.4.2 Input from Non-Terminal 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 9.2). As long as the value of the *expression* is non-zero, the specified file is read through one of the 12 internal I/O channels. If the *expression* is zero, or missing completely, input is from the user terminal. No ? character is printed on the terminal when input is requested from a device other than the user terminal. For example:

```
10 OPEN "PR:" FOR INPUT AS FILE 3
20 INPUT #3, A$,B$
```

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

Note that the data format is identical to the standard INPUT format. If the user wants to read numeric or string data from a file previously created (on disk or DECtape, for example) he should insert commas and carriage returns in the data when he places the data in the file. For example:

```
    .
    .
    .
100 OPEN "DT0:LEN" FOR OUTPUT AS FILE 1%
110 PRINT #1%, A "," B "," C
120 CLOSE 1%
130 OPEN "DT0:LEN" AS FILE 1%
140 INPUT #1%, A,B,C
150 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 position pointer within the file back to the beginning of the file, so that the entire file becomes available again for sequential referencing.

10.4.3 Opening the User Terminal as an I/O Channel

The internal file designator (following the # character in the INPUT or PRINT statements) is always in the range 1 to 12. File designator Ø is, by definition, always open as the user's terminal. Internal file designator Ø cannot be closed or opened. Use of file #Ø is indicated below (no OPEN #Ø statement is necessary or allowed).

```
10 INPUT #0, A$
```

is equivalent to:

```
10 INPUT A$
```

It is sometimes useful to be able to request keyboard input without having the "?" prompting character printed first. This can be accomplished by opening the user's terminal ("KB:") on some internal file designator other than Ø. The ? character is only generated for input requests on file #Ø, as shown in the following example:

```
LISTNH
10 OPEN "KB:" AS FILE 1
20 PRINT "WITH USE OF INTERNAL FILE DESIGNATOR"
30 PRINT "TYPE YOUR NAME, FOLLOWED BY RETURN KEY"
40 INPUT #1, A$; "THANK YOU"
50 PRINT: PRINT
60 PRINT "FOR COMPARISON, WITHOUT FILE DESIGNATOR"
70 PRINT "TYPE YOUR NAME, FOLLOWED BY RETURN KEY"
80 INPUT A$; "THANK YOU"
90 END

READY

RUNNH
WITH USE OF INTERNAL FILE DESIGNATOR
TYPE YOUR NAME, FOLLOWED BY RETURN KEY
J. P. JONES
THANK YOU

FOR COMPARISON, WITHOUT FILE DESIGNATOR
TYPE YOUR NAME, FOLLOWED BY RETURN KEY
? J. P. JONES
THANK YOU
READY
```

10.5 APPENDING DATA TO DISK FILES

Information can be added to existing files (appended) by using the MODE option, described in Section 9.2.4 in the OPEN statement. Once opened, a data file can be appended simply by using the PRINT# statement described in Section 10.3.1.

To write data to a new block following the current end of file in a disk file, specify the MODE 2% option in the OPEN statement. For example:

```
100 OPEN "DATA" AS FILE 1%, MODE 2%
```

As a result, the system opens the file DATA under the current account on the system disk. The next output operation creates a new block and appends it to those currently allocated to the file. Any fill characters in the previous last block of the file remain when the system appends the new last block. Do not use the OPEN FOR OUTPUT form of the OPEN statement, as it deletes the existing file.

10.6 PROGRAMMING EXAMPLE

Formatted ASCII disk files can be used to store data of variable length. For example, strings of different sizes can be stored immediately next to each other, making the most efficient use of storage space on the disk.

The example below illustrates how a programmer stores strings in a data file called DIARY. He opens the disk file with the MODE 2% option to append new data to the file when required. The program shown below is run the first time with the following user responses:

```
1000 REM  THIS PROGRAM SETS UP AN ASCII FILE IN APPEND MODE.
1010 OPEN "DIARY" AS FILE 3%, MODE 2%
1020 PRINT "NEXT LINE";
1030 INPUT LINE L$
1040 IF LEFT(L$,LEN(L$)-2) = "END" THEN 1070
1050 PRINT #3%, L$;
1060 GOTO 1020
1070 CLOSE 3%
1080 END
```

```
RUNNH
NEXT LINE?                                               JULY 5, 1686
NEXT LINE? DEAR DIARY,
NEXT LINE?      EARLIER TODAY, WHILST REPOSING 'NEATH YE OLDE APPLE TREE,
NEXT LINE? I WITNESSED A COMMONPLACE BUT PAINFUL EVENT WHICH METHINKS MAY
NEXT LINE? PROVE OF PUBLICK INTEREST.  TO WIT:  A RIPE AND GOOD-SIZED
NEXT LINE? APPLE LOOSENED ITSELF FROM THE TREE AND FELL EARTHWARD,
NEXT LINE? STRIKING THIS HUMBLE NARRATOR UPON THE HEAD AND CAUSING HIM TO
NEXT LINE? BE DAZED CONSIDERABLY.
NEXT LINE?      AS A RESULT, METHINKS THERE IS AN ATTRACTION BETWEEN BODIES.
NEXT LINE? I HERE USE THE WORD ATTRACTION IN GENERAL FOR ANY ENDEAVOR
NEXT LINE? WHATEVER, MADE BY BODIES TO APPROACH TO EACH OTHER, WHETHER THAT
NEXT LINE? ENDEAVOR ARISE FROM THE ACTION OF THE BODIES THEMSELVES, AS
NEXT LINE? TENDING TO EACH OTHER OR AGITATING EACH OTHER BY SPIRITS
NEXT LINE? EMITTED; OR WHETHER IT ARISED FROM THE ACTION OF THE ETHER OR OF
NEXT LINE? THE AIR, OR OF ANY MEDIUM WHATEVER, WHETHER CORPOREAL OR
NEXT LINE? INCORPOREAL, IN ANY MANNER IMPELLING BODIES PLACED THEREIN
NEXT LINE? TOWARDS EACH OTHER.  FURTHER, THIS ATTRACTION BETWEEN PARTICLES
NEXT LINE? IS DIRECTLY AS THEIR MASSES AND INVERSELY AS THE SPACE BETWEEN
NEXT LINE? THEM MEASURED FROM CENTRE TO CENTRE, THE QUANTITY OF THIS SPACE
NEXT LINE? BEING THENCE MULTIPLIED BY ITSELF.  I CALL THIS ATTRACTION THE
NEXT LINE? LAW OF GRAVITATION (THE THEORY OF RELATIVITY, HAVING A BETTER
NEXT LINE? SOUND, IS TOO OBSCURE).
NEXT LINE?      BUT I HASTEN TO OTHER THINGS.
NEXT LINE?
NEXT LINE?                                               IZZY NEWTON
NEXT LINE?
NEXT LINE? ------------------------------------------------------------------
NEXT LINE? END

READY
```

The file DIARY now contains the strings input above. More than two but less than three full blocks are used in this file at this point. The second execution of the above program adds information to the existing file, beginning at the next empty block (in this case, block #4).

```
RUNNH
NEXT LINE?                                                  JULY 6, 1686
NEXT LINE? DEAR DIARY,
NEXT LINE?      THE DYNAMICS OF FALLING APPLES IS HAZARDOUS WORK.  HENCE-
NEXT LINE? FORTH I SHALL CONCENTRATE ON INVENTIONS.  I PLAN TO INVENT
NEXT LINE? DIFFERENTIAL CALCULUS BEFORE SUPPER.
NEXT LINE?      SO LONG FOR NOW, DEAR DIARY.
NEXT LINE?
NEXT LINE?                                                  IZZY NEWTON
NEXT LINE? END

READY
```

The following program is run to print the contents of the file DIARY, from the beginning of the file to the last entry.

```
1000 REM  THIS PROGRAM READS THE FILE "DIARY."
1010 OPEN "DIARY" AS FILE 4%
1020 ON ERROR GOTO 1060
1030 INPUT LINE #4%, M$
1040 PRINT M$;
1050 GOTO 1030
1060 CLOSE 4%
1070 END
```

```
RUNNH
                                                JULY 5, 1686
DEAR DIARY,
        EARLIER TODAY, WHILST REPOSING 'NEATH YE OLDE APPLE TREE,
I WITNESSED A COMMONPLACE BUT PAINFUL EVENT WHICH METHINKS MAY
PROVE OF PUBLICK INTEREST.  TO WIT:  A RIPE AND GOOD-SIZED
APPLE LOOSENED ITSELF FROM THE TREE AND FELL EARTHWARD,
STRIKING THIS HUMBLE NARRATOR UPON THE HEAD AND CAUSING HIM TO
BE DAZED CONSIDERABLY.
        AS A RESULT, METHINKS THERE IS AN ATTRACTION BETWEEN BODIES.
I HERE USE THE WORD ATTRACTION IN GENERAL FOR ANY ENDEAVOR
WHATEVER, MADE BY BODIES TO APPROACH TO EACH OTHER, WHETHER THAT
ENDEAVOR ARISE FROM THE ACTION OF THE BODIES THEMSELVES, AS
TENDING TO EACH OTHER OR AGITATING EACH OTHER BY SPIRITS
EMITTED; OR WHETHER IT ARISED FROM THE ACTION OF THE ETHER OR OF
THE AIR, OR OF ANY MEDIUM WHATEVER, WHETHER CORPOREAL OR
INCORPOREAL, IN ANY MANNER IMPELLING BODIES PLACED THEREIN
TOWARDS EACH OTHER.  FURTHER, THIS ATTRACTION BETWEEN PARTICLES
IS DIRECTLY AS THEIR MASSES AND INVERSELY AS THE SPACE BETWEEN
THEM MEASURED FROM CENTRE TO CENTRE, THE QUANTITY OF THIS SPACE
BEING THENCE MULTIPLIED BY ITSELF.  I CALL THIS ATTRACTION THE
LAW OF GRAVITATION (THE THEORY OF RELATIVITY, HAVING A BETTER
SOUND, IS TOO OBSCURE).
        BUT I HASTEN TO OTHER THINGS.

                                                IZZY NEWTON

-----------------------------------------------------------------
                                                JULY 6, 1686
DEAR DIARY,
        THE DYNAMICS OF FALLING APPLES IS HAZARDOUS WORK.  HENCE-
FORTH I SHALL CONCENTRATE ON INVENTIONS.  I PLAN TO INVENT
DIFFERENTIAL CALCULUS BEFORE SUPPER.
        SO LONG FOR NOW, DEAR DIARY.

                                                IZZY NEWTON

READY
```

CHAPTER 11

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 a simple random-access file system called virtual core.

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 core memory, but within the RSTS-11 disk file system instead. Data stored in disk files external to the user program remain, even after the user leaves his terminal, and can be retrieved by name at a later session. Items within the file are individually addressable, 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.

With the virtual array facility, BASIC-PLUS programs can operate on data structures that are too large to be accommodated in core at one time. The disk file system is used for storage of data arrays, and only portions of these files are maintained in core at any given time.

With virtual data storage, the user can reference any element of one or more matrices within the file, no matter where in the file that element resides. This random access of data allows the user non-sequential referencing of the data for use in any BASIC statement. The virtual core matrices are read into memory automatically by the system.

The order in which array elements are referenced can have a significant effect on the program execution time. This section also describes the algorithms used in the virtual array processor, in order that users concerned with efficiency can optimize their use of this facility.

Each disk file appears to the user program as a contiguous sequence of 256-word records. Any position in a file can be specified internally with a two-component address; the first part being the relative record within the file, and the second being the position of the item within the block. One of the functions of the virtual array processor is to transform, or map, each virtual array reference into its corresponding file address.

Virtual arrays are stored as unformatted binary data. This means that no I/O conversions (internal form-to-ASCII) need to be performed in storing or retrieving elements in virtual storage. Thus, there is no loss of precision in these arrays, and no time wasted performing conversions.

11.1 VIRTUAL CORE DIM STATEMENT

In order for a matrix of data to exist in virtual core, it must be declared in a special form of the DIM statement. This special DIM statement is as follows:

line number DIM# < *integer constant* >,<*list*>

where the *integer constant* is between 1 and 12 and corresponds to the internal file designator on which the program has opened a disk file. The variable *list* appears as it would in a DIM statement for a core-resident matrix. Thus, a 1ØØ by 1ØØ matrix could be defined as:

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

Floating-point constants, integer constants and strings can be stored in virtual core matrices. More than one matrix can be specified in one virtual core field. For example:

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

allocates space for 1ØØØ floating-point numbers, 2ØØØ integer numbers and 25ØØ character strings (16 characters long). However, if a virtual array is defined in this fashion, future references should always dimension the arrays to the same size.

11.2 VIRTUAL CORE STRING STORAGE

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 memory are of variable length from Ø characters to any arbitrary length. Strings in virtual core matrices are of fixed length from Ø characters to a specified maximum length (all elements of a single string array have the same maximum length). This fixed length can be defined by the program and varies from 2 characters to 512 characters. The system forces the maximum length to be a power of 2; that is, one of the following lengths:

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

Each element in the virtual core string need not use the maximum length available, even though space is reserved for each element to be the maximum size. If the user indicates other than one of the values above, he receives the next higher size. Thus:

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

is equivalent to:

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

If no length is specified, a default length of 16 characters is assumed. The maximum length of virtual core strings is specified as an expression in the DIM statement, using the form:

line number DIM #<*integer constant*>,<*string (dimension(s))*>=<*integer constant*>

For example:

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

where:

A$ consists of 1Ø1 strings of 32 characters each, maximum;
B$ consists of 1Ø1 strings of 4 characters each, maximum;
C$ consists of 1Ø1 strings of 16 characters each, maximum.

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

11.3 OPENING AND CLOSING A VIRTUAL CORE FILE

In order for the user to reference his virtual core file, he must first associate a disk file (by name) with an internal channel designator from 1 to 12 (which is then used in the virtual DIM declaration). This is done with an OPEN, OPEN FOR INPUT, or OPEN FOR OUTPUT statement:

line number OPEN <*string*>$\left[\begin{matrix}\text{FOR INPUT}\\\text{FOR OUTPUT}\end{matrix}\right]$ AS FILE <*expression*>

where the *string* is the name of a disk file and the *expression* specifies an internal file designator (this is the same format described in Section 9.2); thus:

```
35 OPEN "ACCT" AS FILE 1%
```

associates the file named ACCT with internal channel 1. If ACCT already exists, then the existing file is used. If there is no file named ACCT, one would be created. If the user wishes to destroy an old file named ACCT and create a new file of the same name, he can use the statement:

```
35 OPEN "ACCT" FOR OUTPUT AS FILE 1%
```

which causes the file to be deleted if it already exists and a new file created (in which case the file is deleted if not used). If the user wants to be alerted that the file ACCT is not present, he could write:

```
35 OPEN "ACCT" FOR INPUT AS FILE 1%
```

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

NOTE

> Virtual core arrays do not permit internal buffers larger than 512 characters; therefore, the RECORDSIZE option is not used when opening a virtual core array file.

11.3.1 Pre-Extending a Virtual Array

Since the system overhead for extending a file by a single data element and by many elements is nearly the same, it is much more efficient to immediately extend a newly created file to its final length than to extend it many times in increments of a single data element. Whenever the final size of a file is known, the file should be extended to its full size in a single operation.

For example:

```
100 OPEN "DATA" FOR OUTPUT AS FILE 1%
200 DIM #1, A(10000)
300 A(10000) = 0
```

This extends the virtual core array A to its final length. Virtual core arrays, however, are not initially zeroed by the system. In the example given above, A(Ø) through A(9999) contain indeterminate values. Unless the user is careful these values could cause a program failure. The user is advised to first zero the virtual core array. This could be done as follows:

```
300 MAT A = ZER(10000)
```

which both zeros and extends the file to its maximum size. However, this uses the more time consuming method of extending the file. A more optimal approach would be:

```
300 A(10000%)=0: MAT A = ZER(10000)
```

which immediately extends the file to its maximum and then zeros it sequentially. These techniques have frequent practical application.

11.3.2 Closing a Virtual Core File

The CLOSE statement must be used to terminate I/O between the BASIC program and the virtual array. Once a virtual array has been closed, it can be reopened for reading or writing on any internal file designator.

All virtual arrays must be closed before the end of program execution. The CLOSE statement causes the output of the last data element to a virtual core file. Execution of a CHAIN statement automatically closes any open arrays, but does not cause the output of the last data elements to the array. The format of the CLOSE statement is as follows.

line number CLOSE <*expression*> [,<*expression*>...]

The *expression* indicated has the same value as the *expression* in the OPEN statement and indicates the internal channel number of the array to close. Any number of arrays can be closed with a single CLOSE statement; it more than one array is to be closed, the *expressions* are separated by commas. The CLOSE statement writes the current contents of the I/O buffer to a virtual core file before closing it and frees core storage space for the program to open other arrays or files (a maximum of 12 depending upon available space). For example:

```
255 CLOSE 2,4
345 CLOSE 10
```

Line 255 above closes the virtual arrays opened on internal I/O channels 2 and 4. Line 345 closes the array open on internal I/O channel 10.

11.4 VIRTUAL CORE PROGRAMMING CONVENTION

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

1. Reference a virtual core array without first opening the file.
2. Reference a non-disk file (for example, DECtape or the line printer) as a virtual core array.
3. Exceed virtual core, that is, define a matrix that is bigger than the amount of available disk storage on the system.

It is important to remember that a virtual core file must be closed before stopping the program (like any other file).

11.4.1 Array Storage

Any data element in a virtual array is completely contained within a single segment (256 words) of disk storage. This restriction has no effect on integers and floating-point items, where the size of data items is fixed (1-word integer, 2- or 4-word floating point numbers), but does limit the maximum length of a virtual string to 512 characters (512 bytes). The number of data elements stored in each disk segment is a function of the size of each element. For virtual strings, the number of elements is also related to the maximum string length specified in the DIM# statement. The size of a virtual string is defaulted to 16 characters, and can be specified as: 2, 4, 8, 16, 32, 64, 128, 256, or 512. Table 11-1 indicates the number of array elements stored in each segment of a virtual file.

Table 11-1
Virtual Array Storage Capabilities

Data Type	Number of Elements per Segment
Integer (%)	256
2-Word Floating Point	128
4-Word Floating Point	64
String ($) (where the maximum length = N)	512/N

Strings in virtual storage occupy pre-allocated space in the virtual file, and thus differ from strings in core storage, where space is allocated dynamically. A disk segment containing virtual strings can be considered to be a succession of fields, each of the maximum string length. When a virtual string is assigned a new value, it is stored left-justified in the appropriate field. If the new string value is shorter than the maximum length, the remainder of the field is filled with zeros. When the string is retrieved, its length is computed as the maximum string length minus the number of zero-filled bytes.

11.4.2 Translation of Array Subscripts into File Addresses

In order to translate an array subscript into a file address, RSTS-11 computes (a) the relative distance from the specified item to the first item in the array, and then adds (b) the relative distance from the first element of the array to the first item in the file. The first quantity (a) is computed from the array subscript and the number of elements per block, as shown in Table 11-1. The second number (b) is a constant for each array in a file, and is computed from the parameters specified in the DIM# statement.

Since the DIM# statement contains the only information used to define the structure of a file, it is possible for the user to specify differently accessing arrangements for the same file in one or more programs. For example, the user can reference the same data as either a series of 32-byte strings (A2$) or 16-byte strings (A1$), with the following statements:

```
10 DIM #1,A1$(1000) = 16       !16 CHARACTER STRINGS.
20 DIM #1,A2$(500) = 32        !32 CHARACTER STRINGS.
30 OPEN 'FIL1' AS FILE 1       !VIRTUAL ARRAY FILE.
```

The user should keep in mind that in BASIC-PLUS, as in most BASICs, array subscripts begin with 0, not 1. An array with dimension n, or (n,m) actually contains n+1, or [(n+1)*(m+1)] elements.

User programs may define two-dimensional virtual arrays as well as singly dimensioned ones. Two-dimensional arrays are stored on disk (and in core) linearly, row-by-row. Thus, in the case of an array (X1,2), the array appears logically as:

X(Ø,Ø)	X(Ø,1)	X(Ø,2)
X(1,Ø)	X(1,1)	X(1,2)

while physically it is stored as:

X(Ø,Ø)	lowest address
X(Ø,1)	
X(Ø,2)	
X(1,Ø)	
X(1,1)	
X(1,2)	highest address

If a virtual array is to be referenced sequentially, it is usually preferable to reference the rows, rather than the columns, in sequence. Consider the case in which it is necessary to compute the sum of each row and column in a two dimensional array. Program ONE does this far more efficiently than program TWO (see next page.)

```
10 REM PROGRAM 'ONE' TO COMPUTE SUMS EFFICIENTLY
20 REM "ARRAY" CONTAINS VIRTUAL ARRAY
30 REM R(I) IS SUM OF ROW I
40 REM C(J) IS SUM OF COLUMN J
50 DIM #1, A(10,50)                !10 ROWS, 50 COLUMNS
60 DIM R(10), C(50)
70 OPEN "ARRAY" AS FILE 1          !OPEN VIRTUAL FILE
80 MAT R = ZER                     !INITIALIZE SUM
90 MAT C = ZER
100 FOR I = 1 TO 10                !OPERATE ROW BY ROW
110 FOR J = 1 TO 50                !DO EACH COLUMN IN ROW
120 R(I) = R(I) + A(I,J)           !TOTAL ACROSS ROW
130 C(J) = C(J) + A(I,J)           !TOTAL DOWN COLUMN
140 NEXT J                         !NEXT COLUMN IN ROW
150 NEXT I                         !NEXT ROW
160 MAT PRINT R;                   !PRINT ROW TOTALS
170 MAT PRINT C;                   !PRINT COLUMN TOTALS
200 CLOSE 1
999 END
```

```
10 REM PROGRAM 'TWO' HAS INEFFICIENT USE OF VIRTUAL CORE
20 REM "ARRAY"                     CONTAINS VIRTUAL ARRAY
30 REM R(I)                        CONTAINS SUM OF ROW I
40 REM C(J)                        CONTAINS SUM OF COLUMN J
50 DIM #1, A(10,50)                !10 ROWS, 50 COLUMNS
60 DIM R(10), C(50)
70 OPEN "ARRAY" AS FILE 1
80 MAT R = ZER
90 MAT C = ZER
95 REM - REFERENCING ARRAY ELEMENTS DOWN THE
96 REM - COLUMNS CAUSES EXTRA DISK REFERENCES
100 FOR J = 1 TO 50                !OPERATE ONE COLUMN AT A TIME
110 FOR I = 1 TO 10                !AND EACH ROW IN COLUMN
120 R(I) = R(I) + A(I,J)           !TOTAL ACROSS ROW
130 C(J) = C(J) + A(I,J)           !TOTAL DOWN COLUMN
140 NEXT I                         !NEXT ROW IN COLUMN
150 NEXT J                         !NEXT COLUMN
160 MAT PRINT R;
170 MAT PRINT C;
200 CLOSE 1
999 END
```

In virtual core arrays it is permissible to have two (or more) arrays sharing the same file. That is, the following DIM# statement is perfectly legal.

```
100   DIM#1,A(1000),B%(999),C(1000)
```

The matrix B% begins immediately after the 1000th element of A and the matrix C begins immediately after B%(999). Therefore, the disk layout is as shown in Figure 11-1.

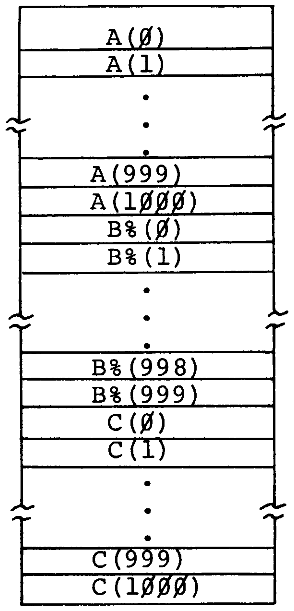


Figure 11-1 Virtual Array File Layout

There is, however, an exception to this rule. Elements in string arrays are allocated a fixed number of bytes in the disk file. This is either 2, 4, 8, 16, 32, 64, 128, 256 or 512 bytes of storage. A single string element must not cross a disk block boundary (where each disk block contains 512 bytes or 256 words). Consider the following case:

```
100  DIM A%(2),B$(1000)=4
```

The first three words of the disk block are allocated to A%. If the array B$ were to begin immediately after A%, one of the elements of B$ would cross a block boundary. Hence, B$ begins at the start of the second block in the file rather than immediately after A%.

The rule can be stated as follows: When more than one array is assigned to a single virtual array file, each array begins immediately following the last element of the preceding array unless such an allocation would cause an element of the array to be split across two disk blocks, in which case the array begins at the start of the next block of the file, and the remaining words of the current block are unused.

11.4.3 Access to Data in Virtual Arrays

Only a portion of a virtual array is in core at any given time. This data is transferred directly between the disk and an I/O buffer in the user core area, created when the OPEN statement is executed. This buffer must be 256 words (one segment) long, and may not be specified as several segments with the RECORDSIZE option in the OPEN statement. For each virtual array file, RSTS-11 notes (1) the segment of the file in the buffer, and (2) whether the data in the buffer has been modified since it was read into core.

After RSTS-11 translates a virtual array address into a file address, it checks whether the segment containing the referenced item is currently in the buffer. If the necessary segment is present the reference proceeds; but if not, another portion of the file is read into the buffer. If the current data in the buffer has been altered, it is necessary to rewrite this data on the disk prior to reading new data into the buffer.

The referencing algorithm, which minimizes the number of disk memory accesses generated when handling virtual arrays, is flow-charted in Figure 11-2.

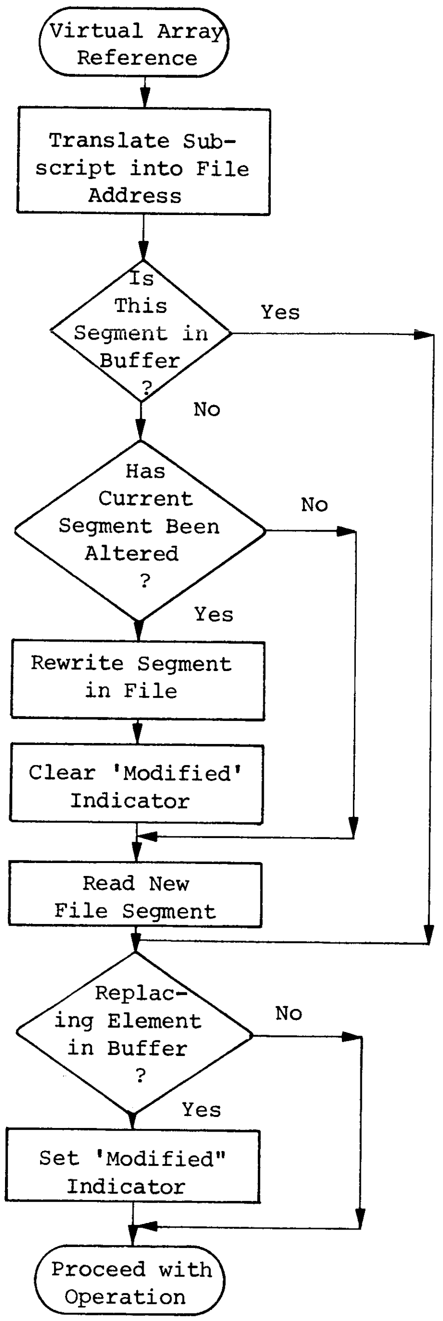

Virtual Array Reference
Translate Subscript into File Address
Is This Segment in Buffer ?
Yes
No
Has Current Segment Been Altered ?
No
Yes
Rewrite Segment in File
Clear 'Modified' Indicator
Read New File Segment
Replacing Element in Buffer ?
No
Yes
Set 'Modified" Indicator
Proceed with Operation

Figure 11-2   Virtual Array Accessing Algorithm

All references to virtual arrays are ultimately located via file addresses relative to the start of the file. No symbolic information concerning array names, dimensions, or data types is stored within the file. Thus, different programs may use different array names to refer to the data contained within a single virtual array file. The user must be cautious in such operations, since it is his responsibility to ensure that all programs referencing a given set of virtual arrays are referencing the same data. Consider the following example:

Program ONE contains

```
10 DIM#1, X(10),Y(10)
20 OPEN "FILE" AS FILE 1
      .
      .
      .
```

Program TWO contains

```
10 DIM#1, Z(10),X(10)
20 OPEN "FILE" AS FILE 1
```

Whenever program TWO references the array Z, it is using the data known to program ONE as array X. Both X and Z are the first arrays in their declarations, both contain floating-point data, and both are 11 elements (X(Ø),...,X(1Ø)) long. These two arrays, the, correspond in position, type, and dimension.

References to the array X (in ONE) and to the array X (in TWO) <u>do not</u> refer to the same data, even though both are using the same virtual file (FILE). The concept of using relative position, rather than name, to identify data items is familiar to users of the FORTRAN common facility.

Within a single BASIC-PLUS program it is possible to open a single virtual core array file twice on the same channel for the purpose of reallocating the data within the file. For example:

```
145 OPEN "DATA" FOR INPUT AS FILE 1
150 DIM#1, A$(10)=4
155 DIM#1, B$(4)=16
```

The program now has access to the file DATA through both the array A$ and the array B$. Each element of B$ contains four elements of A$ (B$(Ø) is equivalent to the elements A$(Ø) through A$(3), etc.).

Note that the two DIM# statements reference that file on a single channel number (#1 in this case).

Note also that the two statements:

```
75 DIM#1, A(10)
80 DIM#1, B(10)
```

are not equivalent to the statement:

```
90 DIM#1, A(10), B(10)
```

In the first case the arrays A and B are equivalent to each other and constitute the first array in the file open on channel 1. In the second case the arrays A and B are defined as both existing in the file open on channel 1.

CAUTION

The user is advised not to open a single file under two different channel numbers. For example:

```
55 OPEN "VALUES" AS FILE 1
55 OPEN "VALUES" AS FILE 2
     .
     .
     .
100 DIM#1, X$(20)
105 DIM#2, Y$(20)
```

causes two buffers to be created for the storage of input to/from channel 1 and to/from channel 2. Data output to channel 1 is not available to channel 2, etc.

11.4.4 Allocating Disk Storage to Virtual Files

The dimensions indicated in a DIM# statement set maximum allowable values for subscripts, and are not used to compute the initial size of the virtual file to be allocated on disk. Instead, the file is created with an initial length of Ø segments, and segments are appended to the file, to accommodate the highest referenced file address in the array. This permits a user to specify array dimensions larger than required at the time the program is written; such programs may eventually operate on larger arrays without modification, and without tying up disk storage unnecessarily.

Areas of unallocated disk storage are found only at the end of a file.

As segments are appended to a file, their contents are not initialized to zero. The data previously recorded in a segment (when it was part of another file) is available to the new owner of the segment. Users whose files contain confidential information should explicitly overwrite all data in such files, prior to file deletion, in order to protect data contained therein.

To override the dynamic virtual array allocation, reference the last element in the virtual array file. This causes all segments in the file, up to and including the last, to be allocated. As noted above, the contents of these segments as appended to the file is unknown. Using the MAT ZER command is advisable if the program depends on array values being initialized to a known (zero) quantity.

11.4.5 Simultaneous Access of a Virtual Core Array by Several Programs

As mentioned in Section 9.2, only the first program to open a file (array) is given write privileges. When a second program attempts to modify an array which is already open, the appropriate block is read from the disk but changed only in the second user's buffer - not on the disk. When the second program references this array and attempts to read another block from the disk, a PROTECTION VIOLATION error occurs. This is because the system attempts to update the disk with the new information in the current block before the required block is read into core. Since the second program has no write privileges, the disk cannot be updated. A CLOSE operation at this point also results in a PROTECTION VIOLATION error for the same reason. Once the job returns to BASIC-PLUS command level and a NEW, OLD or RUN command is executed, a CLOSE is performed on all channels. In this case, no write is attempted so the CLOSE is successful.

The best way to avoid simultaneous write accessing of a virtual core array is to determine whether the user program has write privileges. Do this with the STATUS variable (see Section 12.3.5) as shown below.

```
    10 OPEN "ARRAY" AS FILE 1%
    20 IF (STATUS AND 1024%) THEN <WRITE LOCKED>
                                  ELSE <WRITE OKAY>
```

MODE 1% should <u>not</u> be used for updating an array by several programs simultaneously. This is because a user's buffer is modified when an array is open with the MODE 1% option -- the disk is not updated at this time. (Even when the first program unlocks the file, allowing other programs to access the array, the first program's modifications exist only in the first user's buffer.) The array is updated only when the first user accesses data from another block, as explained above.

11.5 PROGRAMMING EXAMPLE

As an example of virtual core usage, consider the problem of generating a large array of random numbers. Since a physical disk block is 256 words, the most efficient array would contain a multiple of 256 elements. The virtual core file, ARRAY1.DAT, in this example, contains 512∅ data elements in a 2 x 2560 array. The zero row and zero column are used, so this array is dimensioned V%(1%,2559%). Twenty physical blocks are used to store this array. The program shown below creates the virtual array V% by assigning a random value between ∅ and 1∅∅∅ to each element in the array.

```
    1000 OPEN "ARRAY1.DAT" AS FILE 3%
    1010 DIM #3%, V%(1%,2559%)
    1020 FOR I% = 0% TO 1%
    1030 V%(I%,J%) = RND(1) * 1000 FOR J% = 0% TO 2559%
    1040 NEXT I%
    1050 CLOSE 3%
    1060 END
```

Now that the file ARRAY1.DAT has been created, the virtual array can be accessed simply by specifying the elements by their subscripts. The program shown below prints every 256th value. Notice that the format of the array in the DIM statement, below, must be identical to the original format for predictable results. The file's internal channel number and the array's name can change, but the array must be formatted the same way every time it is accessed.

```
1000 OPEN "ARRAY1.DAT" AS FILE 3%
1010 DIM #3%, V%(1%,2559%)
1020 FOR I% = 0% TO 1%
1030 PRINT V%(I%,J%); FOR J%=0% TO 2559% STEP 256%
1040 NEXT I%
1050 PRINT
1060 CLOSE 3%
1070 END

READY

RUNNH
 204  909  954  839  65  131  537  784  371  798  565  173  122  910  39  8  318
  468  958  289

READY
```

Values of array elements can be changed simply by redefining them in assignment statements (e.g., LET, INPUT, READ). For example, the program below changes the value of specified data elements, once they are defined by subscripts.

```
1000 OPEN "ARRAY1.DAT" AS FILE 3%
1010 DIM #3%, V%(1%,2559%)
1020 ON ERROR GOTO 1080
1030 INPUT "ENTER THE I AND J LOCATION OF THE ELEMENT"; I%,J%
1040 N% = V%(I%,J%)
1050 INPUT "ENTER THE NEW VALUE"; V%(I%,J%)
1060 PRINT "OLD VALUE WAS: " N% ".  NEW VALUE IS: " V%(I%,J%)
1070 GOTO 1030
1080 CLOSE 3%
1090 END

READY
```

```
RUNNH
ENTER THE I AND J LOCATION OF THE ELEMENT? 0,0
ENTER THE NEW VALUE? 333
OLD VALUE WAS:  204 .  NEW VALUE IS:  333
ENTER THE I AND J LOCATION OF THE ELEMENT? 0,255
ENTER THE NEW VALUE? 999
OLD VALUE WAS:  78 .  NEW VALUE IS:  999
ENTER THE I AND J LOCATION OF THE ELEMENT? 1,500
ENTER THE NEW VALUE? 1
OLD VALUE WAS:  269 .  NEW VALUE IS:  1
ENTER THE I AND J LOCATION OF THE ELEMENT? ^Z

READY
```

Some thought should be given to access methods of virtual arrays. In the above examples, ARRAY1.DAT was allocated in the following manner:

```
Block 1              V(Ø,Ø) - V(Ø,255)
Block 2              V(Ø,256) - V(Ø,511)
Block 3              V(Ø,512) - V(Ø,767)
   .                      .
   .                      .
   .                      .
Block 1Ø             V(Ø,23Ø4)  - V(Ø,2559)
Block 11             V(1,Ø)     - V(1,255)
   .                      .
   .                      .
   .                      .
Block 2Ø             V(1,23Ø4)  - V(1,2559)
```

Notice that the second subscript varies from Ø to 2559 for each of the two values (Ø and 1) of the first subscript. Since the system transfers an entire physical record from the disk to core at one time, only one disk access is performed for each 256 consecutive data elements (e.g., V(Ø,256) - V(Ø,511)). It is therefore much more efficient to access data elements within a given block than to access data elements in different records.

The two programs shown below access, but do not print, each element in the virtual array. The first access method transfers a new block to core for each data element accessed, resulting in 5,120 disk accesses. The second method, however, transfers a new block to core only once per 256 data elements, resulting in only 20 disk accesses. The difference in execution time between both methods is very significant, as shown below.

Program #1
(Inefficient)

```
1000 OPEN "ARRAY1.DAT" AS FILE 3%
1010 DIM #3%, V%(1%,2559%)
1020 T = TIME(0)
1030 FOR J% = 0% TO 2559%
1040 D% = V%(I%,J%) FOR I% = 0% TO 1%
1050 NEXT J%
1060 PRINT "THE FIRST ACCESS TOOK" TIME(0)-T " SECONDS."
1070 CLOSE 3%
1080 END

READY

RUNNH
THE FIRST ACCESS TOOK 108  SECONDS.

READY
```

Program #2
(Efficient)

```
1000 OPEN "ARRAY1.DAT" AS FILE 3%
1010 DIM #3%, V%(1%,2559%)
1020 T = TIME(0)
1030 FOR I% = 0% TO 1%
1040 D% = V%(I%,J%) FOR J% = 0% TO 2550%
1050 NEXT I%
1060 PRINT "THE SECOND ACCESS TOOK" TIME(0)-T " SECONDS."
1070 CLOSE 3%
1080 END

READY

RUNNH
THE SECOND ACCESS TOOK 5  SECONDS.

READY
```

CHAPTER 12

RECORD I/O

There are three methods of performing I/O in BASIC/PLUS. Formatted ASCII I/O is simple and flexible, but requires conversion of numbers by the system from an internal form to an externally usable ASCII representation and does not permit random access to files. I/O to virtual core arrays permits high-speed random access to files but can be used only on disk files and does not allow true intermixing of string and numeric elements or use of the RECORDSIZE specification.

The third type of I/O, Record I/O, permits the user program to have complete control of I/O operations. Properly used, Record I/O is the most flexible and efficient technique of data transfer available under BASIC-PLUS. These advantages are obtained at the cost of the simplicity of the formatted ASCII and virtual array I/O. Less experienced users should first experiment with the simpler I/O techniques before attempting Record I/O.

12.1 OPENING A RECORD I/O FILE

To open a file for Record I/O requires an OPEN statement, described in Section 9.2. An additional field has been added to the OPEN statement: the MODE field. The format of the OPEN statement, including the MODE field, is as follows:

[*line number*] OPEN <*string*> $\left[\begin{matrix}\text{FOR INPUT}\\\text{FOR OUTPUT}\end{matrix}\right]$ AS FILE <*expr*>

[,*RECORDSIZE* <*expr*>] [,CLUSTERSIZE <*expr*>] [MODE <*expr*>]

[,FILESIZE <*expr*>]

The MODE option is used to establish device-dependent properties of the file. For disk files MODE indicates that the file is to be opened to update data (see Section 12.4.5). For non-file structured magtape operations, MODE establishes the density and parity settings for the magtape. For line printer operation, MODE is used in conjunction with the optional forms control to establish the current form length. For card reader operation, MODE sets a read mode to correspond to specific data card formats[1].

The RECORDSIZE and CLUSTERSIZE options can be specified for Record I/O files as described in Sections 9.2.1 and 9.2.2.

12.2 CLOSING A RECORD I/O FILE

Each Record I/O file must be closed once I/O operations on that file are completed. Files are closed with the CLOSE statement, as described in Section 9.3. The CLOSE statement is of the form:

line number CLOSE <*expr*>[,<*expr*>]

where the value of the expression(s) specifies one of the twelve I/O channels.

Remember, the CLOSE statement for formatted ASCII and virtual array files causes the the final record of the file to be written before closing the file. However, all I/O to Record I/O files is explicitly performed (with GET and PUT statements). For this reason, be sure the user program explicitly writes the last record onto a Record I/O file before executing a CLOSE.

12.3 THE GET AND PUT STATEMENTS

Input and output to Record I/O files is performed directly between the device channel and the I/O buffer created by the OPEN statement. All I/O is specified in terms of single records, using the GET and PUT statements. GET and PUT are of the form:

line number GET# <*expr1*> [,RECORD <*expr2*>]
line number PUT# <*expr1*> [,RECORD <*expr2*>] [,COUNT <*expr3*>]

[1]See the RSTS/E Programming Manual for details.

If the RECORD option (see Section 12.3.3) is not used, the GET statement reads the next sequential record from the file open on the channel designated by <*exprl*>. The record is placed in the I/O buffer which was associated with the channel by the OPEN statement. The size of the record depends upon the characteristics of the device on which the file resides, as described in Table 12-1. In Record I/O, the RECORD option refers to a 512-byte sector, not to a logical data record.

When the RECORD option is used in a GET or PUT statement, a specific record, or sector, is accessed. For example,

```
1ØØ GET #4%, RECORD 8%
```

reads the eighth sector of the file opened on channel 4 into the user I/O buffer. Notice that the preceding 7 sectors of the file need not be read. This feature, not available in formatted ASCII files, is called random access.

Table 12-1
Device Record Characteristics

Device	Input Record Characteristics
disk	Records (sometimes called blocks or segments) are always 512 characters long. When the RECORDSIZE option is specified in the OPEN statement, and a buffer longer than 512 characters is created, the system reads as many full records as possible. If several disk records are read with a single GET statement, the next sequential record is that record immediately following the last record read. In non-file structured operation, the minimum record size is dependent on the device clustersize.
DECtape	For file structured DECtape, records are always 510 characters long. For non-file structured DECtape, records are always 512 characters.
magtape	When performing file-structured I/O, magtape records are always 512 characters. With non-file structured I/O, magtape records can be of any length; only one record can be read per GET statement; and the record length can not exceed the buffer size as determined by the RECORDSIZE option.

Table 12-1 (Cont.)
Device Record Characteristics

Device	Input Record Characteristics
keyboard	The GET statement obtains one line from the keyboard, up to the first line delimiter (CTRL/Z, RETURN, LINE FEED, ESCAPE or FORM FEED).
card reader	A record consists of a single card. The RECORDSIZE option has no effect on card reader input.
paper tape	RSTS-11 reads a full buffer of input from the paper tape reader unless an end-of-tape is detected.

Similarly, if the COUNT and RECORD (see Sections 12.3.2 and 12.3.3) options are not used, the PUT statement writes the contents of the I/O buffer for the specified I/O channel onto the next sequential record of the file. The expression < *expr*1> specifies the internal channel number on which the file was opened. PUT writes a single record on the device, with the exception of disk files which permit several records to be written with one PUT statement (when the RECORDSIZE option in the OPEN statement is used to increase I/O buffer size).

12.3.1 The RECOUNT Variable

Non-file structured devices, as can be seen in the description of the GET statement, can read less than a full buffer of data. To permit the program to determine how much data was actually read, a system variable, RECOUNT, contains the number of characters read following every input operation.

RECOUNT is used primarily for non-file structured input; however, it may also be used with file structured devices. On file structured DECtape and magtape input, RECOUNT is set to the standard record length (51Ø characters for DECtape and 512 characters for magtape). On disk file input, RECOUNT is set to the RECORDSIZE. If the RECORDSIZE is not an integral multiple of 512, an error message is printed.

RECOUNT is set by every input operation on any channel (including channel Ø). It is, therefore, essential that the RECOUNT value be tested immediately following the GET statement.

12.3.2 The COUNT Option

The COUNT option used in a PUT statement with a non-file structured device specifies the number of characters to write in the current record. However, the COUNT expression cannot be greater than the size of the I/O buffer.

For example, where internal channel 1 is opened as magtape unit Ø (non-file structured magtape), the following statement could be used to write an 80-character record:

```
100 PUT #1%, COUNT 80%
```

When COUNT is not used, the PUT statement writes an entire buffer, regardless of whether the buffer contains data.

12.3.3 The RECORD Option

With disk files, the user has the capability of performing random access I/O to any record of the file. Records in a disk file are always 512 characters long and are logically numbered within the file from 1 to n, where n is the size of the file.

The RECORD expression provides the logical record number of the file to GET or PUT. For example, assuming a disk file opened on internal channel 1, the following statement writes the contents of the I/O buffer associated with channel 1 on records 1Ø through 99 of that disk file:

```
200 PUT #1%, RECORD I% FOR I%=10% TO 99%
```

More than one physical record or block can be read or written by assigning a large I/O buffer to the file with the RECORDSIZE option in the OPEN statement. (The size of the buffer does not affect the numbering of the records within the file.)

If the disk file on channel 1 were opened with a RECORDSIZE of 1Ø24 characters (which would cause two 512-character records to be written with each PUT) the PUT statement would be written as follows:

```
200 PUT #1%, RECORD I% FOR I%=10% TO 98% STEP 2%
```

After performing a random access GET or PUT on a disk file, the next GET or PUT statement on that channel accesses the next sequential record if no RECORD number is specified. For example:

```
290 OPEN "DATA" AS FILE 1%, RECORDSIZE 512%
300 GET #1%, RECORD 99%
310 PUT #1%
```

The PUT statement at line 31Ø writes record 1ØØ of the disk file.

12.3.4 BUFSIZ Function

In certain applications, it is important for a program to determine the buffer size of an open channel, especially if the OPEN statement specifies a logical device name. The user program can execute the integer function BUFSIZ to extract this information.

The BUFSIZ function returns an integer value telling the size of the buffer for a specified open channel. For example,

```
Y% = BUFSIZ(N)
```

The statement returns to Y% the size of the buffer in number of bytes for channel N. If the channel is closed, the function returns Ø to Y%.

12.3.5 STATUS Variable

The variable STATUS contains information concerning the last channel on which a user program executed an OPEN statement. The variable is a 16-bit word, each bit of which the user program can test to determine status. Table 12-2 shows the information, the tests, and the meaning of each bit.

Table 12-2
RSTS Variable STATUS

Bit	Test	Meaning
0-7	(STATUS AND 255%)	The first 8 bits of the word contain the handler index. The following values apply for various devices. Ø Disk — 12 Card Reader 2 Keyboard — 14 Magtape 4 DECtape — 16 PK:device[1] 6 Line Printer — 18 DX:device[1] 8 HS Reader — 20 RJ:device[1] 10 HS Punch
8	(STATUS AND 256%)<>0%	The device is open in non-file structured mode or is characteristically non-file structured.
9	(STATUS AND 512%)<>0%	The job does not have read access to the device.
10	(STATUS AND 1024%)<>0%	The job does not have write access to the device.
11	(STATUS AND 2048%)<>0%	The device maintains its own horizontal position. Such devices are keyboard and line printer type.
12	STATUS AND 4096%	Not used.
13,15	IF STATUS <0% AND IF (STATUS AND 8192%)<>0%	Device is a character device which can be controlled by variable RECORD <expr> modifiers. Such devices are keyboard, line printers, and card readers.
	IF STATUS <0% AND IF (STATUS AND 8192%)=0%	Device is a random access blocked device such as disk and non-file structured DECtape.
	IF STATUS >=0% AND IF (STATUS AND 8192%)<>0%	Device is a character device which cannot use RECORD modifiers.
	IF STATUS >=0% AND IF (STATUS AND 8192%)=0%	Device is sequential blocked such as file structured DECtape and Magtape.
14	IF (STATUS AND 16384%) <> 0%	Device is an interactive type (keyboard).

[1]Pseudo keyboards always attempt to use handler index 16. If this slot is filled, however, it proceeds to handler index 18. If both these slots are filled, it attempts to fill handler index 20. Finally, if all three of these slots are filled, the system returns an error message.

The RJ device, if present, always fills handler index 20, but pseudo keyboards and all other non-standard devices can fill handler index 16, 18 and (if RJ is not present) even handler index 20.

12.4 WORKING WITH RECORD I/O FILES

Techniques for opening, closing, reading and writing Record I/O files have been described. But these techniques apply only to indivisible I/O buffers associated with internal channels; no mention has been made of manipulating data within these buffers. Techniques for moving data into or out of a buffer are provided by extensions to the BASIC language. The FIELD, LSET and RSET statements permit the program to access and modify the contents of an I/O buffer, character by character. These statements are discussed in the following sections.

12.4.1 Extending Disk Files

A disk file that is created by an OPEN FOR OUTPUT (or OPEN) statement has a length of Ø. As records are written, the file progressively grows in length; this growth is called extending the file.

A more exact description of file extending is as follows:

a) Is there room in the last cluster[1] of the file for the new record?

b) If so, then the file length is increased and previously unused space in that cluster is used.

c) If not, then a new cluster is appended to the file. There is then room in the newest last cluster for the new record so condition b) applies.

The amount of space actually allocated by the system to a file may be greater than the file length. For example, if the file clustersize is 4 and the first 6 records of that file have been written, the file is of length 6 but is actually allocated 8 records (2 clusters) of space.

A file is extended by attempting to write beyond the current end-of-file. Hence, a program must have write privileges in order to be able to extend a file. There is an exception to the rule that having write access to a file permits a program to extend the file. When a file is opened for update (see Section 12.4.5) several programs can have simultaneous write privileges on a single file.

[1]Note that the file CLUSTERSIZE is the least increment by which a disk file can be extended. (See Section 9.9.2.)

Nonetheless, if a program opens a file in this special update mode, that program may not extend the file. A file can only be extended when open in normal (non-update) mode.

It is possible to extend a file by a number of records at one time. For example:

```
100 OPEN "DATA" FOR OUTPUT AS FILE 1%
200 PUT #1%, RECORD 100%
```

creates a file DATA and (when line 200 is executed) extends it immediately to 1ØØ records. Since the system overhead for extending a file by a single record and by many records is nearly the same, it is much more efficient to immediately extend a newly created file to its final length than to extend it many times in increments of a single record. Whenever the final size of a file is known, the file should be extended to its full size in a single operation.

12.4.2 The FIELD Statement

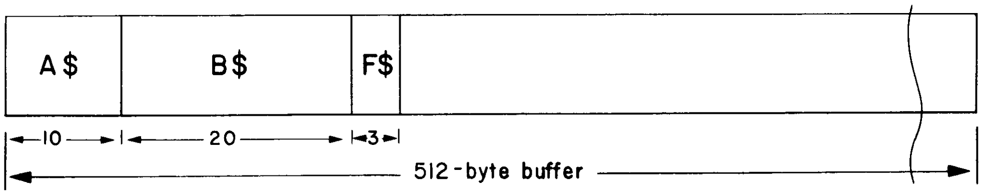
The FIELD statement is used to dynamically associate string names with all or part of an I/O buffer. The FIELD statement has the form:

line number FIELD #<*expr*>, <*expr1*> AS <*stringvar1*>

[,<*expr2*> AS <*stringvar2*>...]

where <*expr*> is an internal channel number associated with some file by an OPEN statement; <*expr1*> is the length, in characters, of the associated string variable; and <*stringvar1*> is a unique string variable name. The names are associated from left to right with successive characters in the I/O buffer assigned to the designated internal channel number. For example:

```
75 FIELD #2%, 10% AS A$, 20% AS B$, 3% AS F$
```



As shown in the previous diagram, statement 75 associates three strings, A$, B$, and F$ in the I/O buffer, with lengths of 1∅, 2∅, and 3 characters, respectively. The total number of characters represented in this statement is 33. The total number of characters must be less than or equal to the actual I/O buffer size (which is dependent on the device and the RECORDSIZE option, as described in Section 9.2.1).

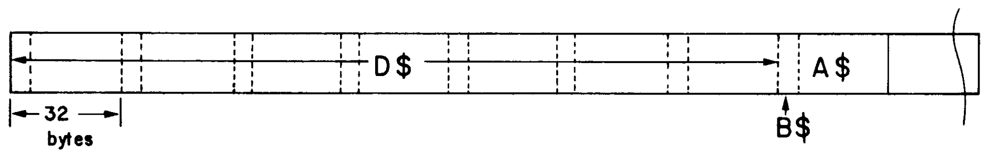
FIELD statements do not move data but rather permit direct access to sections of the I/O buffer via string variables. The effect upon a string variable is temporary and is nullified by any attempt to assign a value to the variable (other than the LSET and RSET, described in Section 12.4.3). For example:

```
100 OPEN "FILE" AS FILE 2%
110 FIELD #2%, 5% AS A$
120 LET A$ = "ABCDE"
```

Line 12∅ causes the string variable A$ to be removed from the I/O buffer. The string ABCDE is <u>not</u> stored in the I/O buffer by line 12∅.

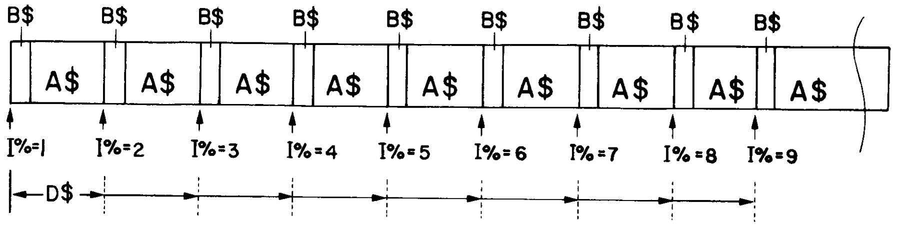
A FIELD statement is an executable statement, rather than a compiler directive (such as a DIM statement). To illustrate: suppose that each record of a disk file contains sixteen 32-character sub-records and that each sub-record consists of one 5-character field and one 27-character field. In order to extract the eighth sub-record from the I/O buffer, the following statement could be executed:

```
200 FIELD #1%, 224% AS D$, 5% AS B$, 27% AS A$
```



Line 2ØØ causes the string variables B$ and A$ to point to the desired sub-record. The string D$ is created to permit the first seven sub-records (7x32=224) to be skipped. An even more general statement could be used to obtain any of the sub-records in the I/O buffer, as follows:

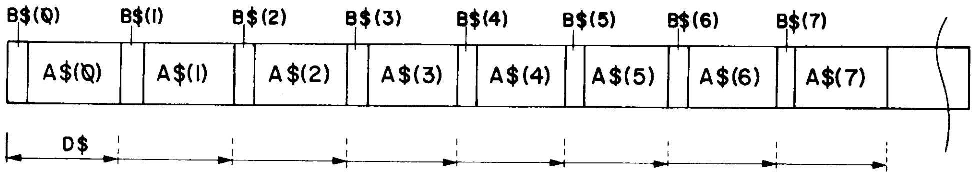
```
190 FOR I% = 0% TO 15%
200 FIELD #1%, (I%-1%)*32% AS D$, 5% AS B$, 27% AS A$
210 NEXT I%
```



When the statement above is executed, I% should contain the number of the sub-record that B$ and A$ are to contain, as an integer from 1 to 16. When I%=1, for example, the expression (I%-1%)*32% equals zero, so B$ points to the first sub-record in the buffer. When I%=2, however, the expression (I%-1%)*32% equals 32, so B$ now points to the first sub-record beyond the 32nd character of the buffer. Each single increment of I% moves B$ 32 characters further into the buffer.

Subscripted string variables can also be used in FIELD statements. For example, the following statements could be used to allocate the sub-records, described in the previous example, to two string arrays:

```
300 DIM A$(15), B$(15)
310 FOR I% = 0% TO 15%
320 FIELD #1%, I%*32% AS D$, 5% AS B$(I%), 27% AS A$(I%)
330 NEXT I%
```



With each iteration of the FIELD statement at line 32Ø the dummy string D$ increases by 32 characters, making the displacement from the start of the I/O buffer to the string B$(I%) equal to 32 times I% characters. Once this loop is executed, the position of each string in the arrays A$ and B$ is fixed, A$(Ø) and B$(Ø) pointing to the first sub-record and A$(15) and B$(15) to the last.

However, virtual array strings must not be defined as string variables in a FIELD statement. When strings are defined as virtual arrays they are required to be in a fixed place in both a disk file and the I/O buffer for that file. Attempting to specify a virtual array string variable in a FIELD statement has no effect on the virtual array string.

12.4.3 LSET and RSET Statements

Once the strings have been defined as part of the I/O buffer by a FIELD statement, values in these strings can be stored without moving them from the I/O buffer. The LSET and RSET statements store values in a string without redefining the string position. These statements are of the form:

line number LSET <*stringvar*> [,<*stringvar*>...] = <*string*>

line number RSET <*stringvar*> [,<*stringvar*>...] = <*string*>

where <*stringvar*> represents any legal existing string variable name (multiple string variable names can be separated by commas) and <*string*> represents any legal string expression.

LSET and RSET store the value of the string expression into the designated string or strings. The string previously stored in the variable is overwritten. The length of the string is not changed; if the new string is longer than the existing string, the new value is truncated. If the new string is shorter than the existing string, it is either padded with spaces on the right by LSET or padded with spaces on the left with RSET. LSET, then, causes the string to be left-justified in the field and RSET causes the string to be right-justified.

The normal use of LSET and RSET, as described in this section, is to store data in strings allocated within an I/O buffer by a FIELD statement. LSET and RSET can be used to assign a value to any string variable within a BASIC-PLUS program.

12.4.4 Differences Between the LET Statement and the LSET/RSET Statement

The LET statement cannot be used to place string values into an I/O buffer as it causes the string to be redefined elsewhere. Another restriction on LET occurs when that statement is used to equate two strings, as follows:

```
50 LET A$=B$
```

To avoid unnecessary character manipulation, this operation causes A$ and B$ to reference the same string. Normally, any operation which alters B$ causes that string to be moved, so no conflict arises. However, LSET and RSET do not move strings; they alter existing strings in a fixed position.

Therefore, if the value of B$ in line 5Ø above were altered by an LSET or RSET statement, the value of A$ also changes. For example:

```
400 B$ = "ABC"
410 A$ = B$
420 LSET B$ = "XYZ"
```

Both A$ and B$ contain "XYZ" following the execution of line 42Ø.

This phenomenon has another ramification; if the string B$ in this example had been defined by a field statement as being in some I/O buffer, the string A$ would also be in the I/O buffer (being identical to B$). Executing a GET statement to read another record into the I/O buffer would then change the value of A$ as well as B$. For this reason, LSET and RSET should be used only for Record I/O operations; using these statements for other purposes may cause illogical results.

When it is not desirable for the strings A$ and B$ to be physically identical, there is a means of causing the string B$ to be moved into the string A$. This operation is performed as follows:

```
300 LET A$ = B$ + ""
```

Line number 3ØØ appends a null string to B$, which has no effect on the string A$ but causes the two strings to occupy different storage areas.

12.4.5 Update Option for Disk Files[1]

In the description of disk files up to this point, the concept of simultaneous user access to a single file has been largely ignored. The system permits several users to read from a single file simultaneously. However, a problem arises if multiple users attempt to write onto a single file simultaneously. Two users could conceivably try to write the same record of the file, resulting in a loss of data. To avoid this conflict, the system permits only one user at a time to have write privileges on any given file. Thus, a user may fail to obtain write privileges even if the file is not protected against writing. If this occurs, the user must close the file and reopen it at a later time, after the other user has finished with the file and closed it.

In certain applications (for example, sales order-entry applications) it is often normal for multiple users to be updating a single master file. In these cases it is not satisfactory to be constantly closing and reopening the file to obtain write privileges, as this is a time-consuming operation. For this reason a special MODE option is available that permits multiple users to have write access to a file while guarding against simultaneous writing of a single block.

To indicate that a file is being opened for update, the MODE 1% specification is used when the file is opened. For example:

```
100   OPEN "MASTER" AS FILE 1%, MODE 1%
```

[1]Update by MODE 1% is an optional feature of RSTS/E and may not be available in all systems.

when used with a disk file indicates that the file is opened[1]. In this case the program is granted write privileges <u>unless</u> such access is specifically prohibited by the protection code of the file.

A file cannot be simultaneously open for update by one user and open in normal (non-update) mode by another user. Attempting to open a file for update if it is already open in normal mode, or attempting to open a file in normal mode if it is already open for update, results in a PROTECTION VIOLATION (ERR=1Ø) error.

Once a file has been opened for update, any read operation of a specific physical record puts it in a special locked state. No other user is permitted to read or write that physical record until it is released (or unlocked) by the program that locked it. Attempting to read or write a record that another user has locked results in a DISK BLOCK IS INTERLOCKED (ERR=19) error which can be trapped with an ON ERROR GOTO statement. There are five ways for a program to unlock the record:

1. The next write operation on the file unlocks the block.
2. Executing the UNLOCK statement. This statement has the form:

   line number UNLOCK *<expr>*

   where *<expr>* is the internal channel number of the file that is opened for update.
3. Any error encountered while accessing the file unlocks the block.
4. Reading another block unlocks the currently locked block. Any read operation locks the block just retrieved.
5. Executing a CLOSE statement on the file unlocks the block.

To illustrate MODE 1%, consider a simple inventory application where operators on several terminals can access one file to enter a part number and order quantities. Assume that the file is sequenced in such a fashion that each part number actually corresponds to the block number that contains information about the part, and that the first four characters of the 512-byte block contain the quantity

[1]The RECORDSIZE option may not be used on files that are opened with MODE 1%.

available as a (2-word) floating-point number. For this example, the remaining 5∅8 characters are ignored. A program to update the quantity available is as follows:

```
100 ON ERROR GOTO 1000                        !FIND OUT ABOUT ERRORS
200 OPEN "INVENT.ORY" AS FILE 1, MODE 1 !OPEN FILE IN UPDATE MODE
300 FIELD #1, 4 AS C$                         !C$ IS QTY IN FILE
400 INPUT "PART NUMBER";N;"QUANTITY";Q  !GET PART # AND QTY
500 GET #1, RECORD N                          !READ APPROPRIATE RECORD
600 X = CVT$F(C$)-Q                           !COMPUTE QTY REMAINING
700 IF X>=0 THEN 800                          !ENOUGH ON HAND?
710 UNLOCK #1                                 !PERMIT OTHER ACCESSES
720 PRINT "ONLY";CVT$F(C$);"ITEMS LEFT" !REPORT STOCK LEVEL
730 GOTO 400                                  !REVISE ORDER?
800 LSET C$ = CVTF$(X)                        !STORE NEW QTY ON HAND
850 PUT #1, RECORD N                          !REWRITE INTO FILE
900 GOTO 400                                  !NEXT TRANSACTION
1000 IF ERR <> 19 THEN ON ERROR GOTO 0  !IGNORE NON-INTERLOCK ERRORS
1100 PRINT "WAITING"                          !LET HIM KNOW WE'RE HERE
1200 SLEEP 5                                  !WAIT FOR CURRENT ACCESS
1300 RESUME 500                               !TRY AGAIN
1400 END
```

MODE 5% (MODE 1%+4%) is used to provide an additional protection feature. When a file is opened with this special update option, the user program can write a block only after it has read it. In other words, a PUT operation can be executed only after a GET operation has previously input the block into the I/O buffer. Attempting to write a block that was not previously read results in a PROTECTION VIOLATION error.

12.5 CVT CONVERSION FUNCTIONS

The FIELD, LSET, and RSET statements allow a program to store or retrieve string data directly from an I/O buffer. To permit floating-point and integer values in Record I/O files, four conversion functions are provided as described in Table 12-3. A fifth conversion function facilitates character string manipulation.

Table 12-3
CVT Conversion Functions

Function Form	Operation
A$ = CVT%$ (I%)	maps an integer into a 2-character string.
I% = CVT$% (A$)	maps the first two characters of a string into an integer. If the string has fewer than two characters, null characters are appended as required.
A$ = CVTF$ (X)	maps a floating-point number into a 4- or 8-character string (depending upon whether the 2-word or 4-word math package, respectively, is being used on the system).
X = CVT$F (A$)	maps the first four or eight characters (depending upon whether the 2-word or 4-word math package, respectively, is being used on the system) of a string into a floating-point number. If the string has fewer than the required number of characters, null characters are appended.
T$ = CVT$$ (S$,M%)	converts the source character string S$ to the string referenced by the variable T$. The conversion is performed according to the decimal value of the integer represented by M% as follows: 1% Trim the parity bit. 2% Discard all spaces and tabs. 4% Discard excess characters: CR, LF,FF, ESC, RUBOUT, and NULL. 8% Discard leading spaces and tabs. 16% Reduce spaces and tabs to one space. 32% Convert lower case to upper case. 64% Convert [to (and] to). 128% Discard trailing spaces and tabs. 256% Do not alter characters inside quotes.

Four of the functions do not affect the value of the data, but rather its storage format. Each character in a string requires one byte of storage (8 bits); hence, characters may assume (decimal) values from Ø through 255 and no others. A 16-bit quantity can be defined as either an integer or a 2-character string; 2-word floating-point numbers can equally be defined as 4-character strings.

The CVT functions which change storage format perform two important functions: first, they permit dense packing of data in records. For example, any integer value between -32768 and 32767 can be packed in a record in two characters using CVT%$; this would only be true for integers between -9 and 99 if the data were stored as ASCII characters. Second, converting the internal numeric representation to an ASCII string (as with the NUM$ function) is a more time-consuming process than that performed by the CVT functions. Thus, the CVT functions provide the means to speed the processing of a large amount of data within a file.

The CVT$$ function manipulates a character string and generates a new character string. This action is unlike other CVT functions because it does not change the internal format of the data, but rather alters the contents of the string. The output string is converted according to an integer value given by the user program and can be any value or sum of any values listed in Table 11-2.

The value 1% in the CVT$$ function removes the parity bit (most significant bit) from each character in the string. Under RSTS-11, characters are usually represented with no parity. All comparison of characters assume no parity. The value 2% removes all space characters (CHR$(32)) and horizontal tab characters (CHR$(9)) from the string while values 8%, 16%, and 128% remove only selective occurrences of space and horizontal tab characters. The terminating and excess characters removed by the value 4% in the CVT$$ function usually have no informational value in a string.

The value 32% converts all lower case characters in a string to upper case. This feature is valuable since some terminals transmit both forms of alphabetic characters. The lower case characters are between CHR$(97) and CHR$(122) and upper case characters are between CHR$(65) and CHR$(90).

The value 64% in the CVT$$ function enables BASIC-PLUS programs to accept the parenthesis and square bracket characters as delimiters of a project-programmer number. This action is desirable when handling account numbers from terminals not having the square bracket characters since most terminal devices have the parenthesis characters.

The value 256% in the CVT$$ function forbids any alteration of characters inside quotes, except parity bit trimming - set by M%=1%. Regardless of other values in the parameter M%, when 256% is included no operations are performed in the source string on characters within quotes.

Generally, the precedence of operations performed on the string is in increasing order of the individual values in the parameter M%. (The 256% value, however, is the exception; its precedence ranks between 1% and 2%.) This order implicitly determines which subsequent operations are performed on the string. For example, if the characters in the source string have their parity bit set and the parity trimming option is not selected, subsequent comparisons required by other options are possibly not successful because comparisons are made against ASCII characters with no parity. For example, a space (SP) character, which is CHR$(32) (octal 40) in no parity or odd parity form, does not compare with a space (SP) character which is CHR$ (160) (octal 240), its even parity form.

Keeping the parity bit in the input character of the string is important in text processing applications where the parity bit of each character is possibly a flag rather than a parity bit. As a result, such flagged characters are not changed or discarded if the parity trimming option is not selected.

The precedence of operations affects the result of values given in the CVT$$ function. If the values 2%, 8%, 16%, and 128% (154% or greater) are given in the CVT$$ function, the values 8%, 16%, and 128% have no effect on the output string since the first option performed (2%) removes all space and tab characters from the string and the remaining values dealing with space and tab characters have no effect. In like manner, the value 16% applies to all space and tab characters not discarded by the 2% and 8% options. Accordingly, to maintain at least a single space interval in a string, the user program must give the 16% value and omit the 2% and 8% values.

The use of the CVT$$ function in general eliminates the need for special code in BASIC-PLUS programs handling string input. For example, the following code at lines numbered 110 through 114 manipulates an input string.

```
95 DIM A6%(128%)
96 N1% = 1%
100 PRINT "TYPE THE INPUT STRING";: INPUT LINE A6$
105 T$ = FNC6$(A6$)
110 DEF FNC6$(A6$)
111 CHANGE A6$ TO A6%: J6% = 0%: FOR X6% = N1% TO A6%(0%)
112 IF A6%(J6%) <= 32% OR A6%(X6%) > 93% THEN 114 ELSE J6%=J6%+N1%
113 A6%(J6%) = A6%(X6%)
114 NEXT X6%: A6%(0%) = J6%: CHANGE A6% TO A6$:
    FNC6$ = A6$: FNEND
115 PRINT "T$ = "; T$
120 GOTO 100
999 END
```

Lines 11Ø through 114 can be replaced by a single CVT$$ function statement at line 1Ø5 as shown in the sample code below.

```
LISTNH
100 PRINT "TYPE THE INPUT STRING";: INPUT LINE A6$
105 T$ = CVT$$(A6$,7%)
115 PRINT "T$ = "; T$
120 GOTO 100
999 END
```

The value 7% in the CVT$$ function is the sum of 1%, 2%, and 4%. The CVT$$ function with a value 7% causes the same results as the code of the user-defined function FNC6$. The following sample dialog shows the effect of the value 7% at line 1Ø5.

```
TYPE THE INPUT STRING?      DEV:    FIL.F EXT  [100,100]
T$ = DEV:FILE.EXT[100,100]
TYPE THE INPUT STRING? ^C
```

The value 255% in the CVT$$ function at line 1Ø5 produces the results shown by the following sample dialog.

```
TYPE THE INPUT STRING?     DEV:  FILE.      EXT    [100,100]
T$ = DEV:FILE.EXT(100,100)
TYPE THE INPUT STRING? ^C
```

(The value 255% can be replaced by -1% to produce the same results.) The following sample dialog shows the effect of the value 189% (1%+4%+8%+16%+32%+128%).

```
TYPE THE INPUT STRING?     I      AM     A          GOOD    MAN.
T$ = I AM A GOOD MAN.
TYPE THE INPUT STRING? ^C
```

12.6 EXAMPLES OF RECORD I/O USAGE

In Figure 12-1, the device KB: is opened with the default size (128 characters) buffer length by the OPEN statement at line 1Ø.

```
LISTNH
10 OPEN "KB:" FOR OUTPUT AS FILE 1
20 FIELD #1, 10 AS A$, 10 AS B$, 10 AS C$
30 LSET A$="12345"
40 RSET B$="67890"
50 RSET C$="VWXYZ"
60 PUT #1, COUNT 30
70 END

READY
```

Figure 12-1
Record I/O Example #1

The FIELD statement at line 2Ø defines three 1Ø-character segments of the buffer as A$, B$ and C$. LSET at Line 3Ø positions "12345" in the leftmost 5 of the first 10 characters of the buffer via the pointer A$. Similarly the second and third 1Ø-character pieces of the buffer are set by lines 4Ø and 5Ø. When run, this program generates:

```
RUNNH
12345          67890     VWXYZ
READY
```

Note that no carriage return/line feed was output by the PUT statement. (The Monitor outputs a CR/LF sequence as the first part of the READY message.)

Figure 12-2 is a program to move data from a file named "SNOOPY.BAS" in the system library (note the $ in the filename) onto the line printer. Both the line printer and the disk file buffers are initialized to 512 characters. The FIELD statements at lines 4Ø and 5Ø set A$ and B$ to refer to these buffers. Data read at line 6Ø is transferred to the line printer buffer by the LSET statement (RSET would also be acceptable in this one case, since both A$ and B$ are the same length) at line 7Ø. Then, at line 8Ø, this data is output to the line printer. The loop terminates on end-of-file on attempting to read past the last block of the SNOOPY.BAS file via the ON ERROR GOTO mechanism.

```
LISTNH
10 OPEN "$SNOOPY.BAS" AS FILE 1
20 ON ERROR GOTO 100
30 OPEN "LP:" FOR OUTPUT AS FILE 2, RECORDSIZE 512
40 FIELD #1, 512 AS A$
50 FIELD #2, 512 AS B$
60 GET #1
70 LSET B$ = A$
80 PUT #2
90 GOTO 60
100 CLOSE 1,2
150 END

READY
```

Figure 12-2
Record I/O Example #2

FIELD statements can be used to perform blocking and deblocking of records where appropriate, as in Figure 12-3.

```
100 GET #2
110 FOR X=0 TO 420 STEP 80
120 FIELD #2, X AS A$, 80 AS B$
    •
    •
    •
    •
    •
180 NEXT X
190 PUT #2
```

Figure 12-3
FIELD Statement Example

Figure 12-4 illustrates the use of the CVT functions to store numerical data in compact form as strings of binary types. The tape punched by this program has each integer represented on two frames of tape. A similar program could be written to read this binary tape.

```
10 DIM A$(99)
20 OPEN "PP:" FOR OUTPUT AS FILE 1, RECORDSIZE 200
30 FIELD #1, 2*I AS Z$, 2 AS A$(I) FOR I=0 TO 99
40 LSET A$(I%) = CVT%$(I%) FOR I%=0% TO 99%
50 PUT #1
60 CLOSE 1
999 END
```

Figure 12-4
CVT Function Example

12.7 THE XLATE FUNCTION

The XLATE function is provided for use with Record I/O to translate a string from one storage code into another. For example, while reading a magtape file, it might be necessary to translate from EBCDIC code to ASCII code so that data could be processed by the PDP-11. The XLATE function is of the form:

```
XLATE (<string1>,<string2>)
```

For example:

```
X$ = XLATE(A$,B$)
```

The first argument, *<string1>*, is the source string, the second argument *<string2>*, is the table string; the string value returned by XLATE is called the target string. Characters are taken sequentially from the source string, and the value of each character (Ø to 255) is used as an index into the table string (that is, Ø means the first character of the table string, 1 means the second, etc.). The character value from the table string is appended to the target string unless the selected character in the table string has a value of Ø or the table string is shorter than the index value. This means that the target string is equal to or shorter than the source string.

For example, the following program removes all characters except "Ø" to "9" and changes the characters "8" and "9" into "A" and "B":

```
LISTNH
100 T$ = "01234567AB"
110 T$ = CHR$(I%)+T$ FOR I%=0% TO 47%
120 REM  - LINE 110 PUT 0'S CORRESPONDING TO CODES 0 TO 47
130 INPUT S$              !GET STRING TO TRANSLATE
140 PRINT XLATE(S$,T$)
150 END

READY

RUNNH
? 12XXX34WWW0987654321
12340BA7654321

READY
```

APPENDICES

The following pages contain a summary of the BASIC-PLUS language, the commands described in the _RSTS-11 System User's Guide,_ and error messages.

APPENDIX A
BASIC-PLUS LANGUAGE SUMMARY

A.1 SUMMARY OF VARIABLE TYPES

Type	Variable Name	Examples
Floating Point	single letter optionally followed by a single digit	A I X3
Integer	any floating point variable name followed by a % character	B% D7%
Character String	any floating point variable name followed by a $ character	M$ R1$

Floating Point Matrix	any floating point variable name followed by one or two dimension elements in parentheses	S(4) 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	-	unary minus	numeric variables and constants
	↑	exponentiation	
	*,/	multiplication, division	
	+,-	addition, subtraction	
Relational	=	equals	string or numeric variables and constants
	<	less than	
	<=	less than or equal to	
	>	greater than	
	>=	greater than or equal to	
	<>	not equal to	
	==	approximately equal to	
Logical	NOT	logical negation	relational expressions composed of string or numeric elements, integer variables or integer valued expressions
	AND	logical product	
	OR	logical sum	
	XOR	logical exclusive or	
	IMP	logical implication	
	EQV	logical equivalence	
String	+	concatenation	string constants and variables
Matrix	+,-	addition and subtraction of matrices of equal dimensions, one operator per statement	dimensioned variables. See Section 7.6.1 for further details.
	*	multiplication of conformable matrices	
	*	scalar multiplication of a matrix, see Section 7.5.1	

A.3 SUMMARY OF FUNCTIONS

Under the Function column, the functions is shown as:

Y=function

where the characters % and $ are appended to Y if the value returned is an integer or character string.

A floating value (X), where specified, can always be replaced by an integer value. An integer value (N%) can always be replaced by a floating value (an implied FIX is done) except in the CVT%$ and MAGTAPE functions (the symbol I% is used to indicate the necessity for an integer value).

Type	Function	Explanation
Mathematical	Y=ABS(X)	returns the absolute value of X.
	Y=ATN(X)	returns the arctangent of X where X is in radians.
	Y=COS(X)	returns the cosine of X where X is in radians.
	Y=EXP(X)	returns the value of e↑X, where e=2.71828.
	Y=FIX(X)	returns the truncated value of X, SGN(X)*INT(ABS(X))
	Y=INT(X)	returns the greatest integer in X which is less than or equal to X.
	Y=LOG(X)	returns the natural logarithm of X, $\log_e X$
	Y=LOG1Ø(X)	returns the common logarithm of X, $\log_{1Ø} X$
	Y=PI	has a constant value of 3.14159
	Y=RND	returns a random number between Ø and 1.
	Y=RND(X)	returns a random number between Ø and 1.
	Y=SGN(X)	returns the sign function of X, a value of 1 preceded by the sign of X
	Y=SIN(X)	returns the sine of X where X is in radians
	Y=SQR(X)	returns the square root of X
	Y=TAN(X)	returns the tangent of X where X is in radians
Print	Y%=POS(X%)	returns the current position of the print head for I/O channel X, Ø is the user's Teletype. (This value is imaginary for disk files.)
	Y$=TAB(X%)	moves print head to position X in the current print record, or is disregarded if the current position is beyond X. (The first position is counted as Ø.)
String	Y%=ASCII(A$)	returns the ASCII value of the first character in the string A$.
	Y$=CHR$(X%)	returns a character string having the ASCII value of X. Only one character is generated.
	Y$=CVT%$(I%)	maps integer into 2-character string, see Section 12.5.
	Y$=CVTF$(X)	maps floating-point number into 4- or 8-character string, see Section 12.5.
	Y%=CVT$%(A$)	maps first 2 characters of string A$ into an integer, see Section 12.5.
	Y=CVT$F(A$)	maps first 4 or 8 characters of string A$ into a floating-point number. See Section 12.5.

Type	Function	Explanation
String, cont'd.	Y$ = CVT$$(A$,I%)	converts string A$ to string Y$ according to value of I%. See Section 12.5.
	Y$ = STRING$ (N1%,N2%)	creates string Y$ of length N1 and characters whose ASCII decimal value is N2. See Section 5.5.
	Y$=LEFT(A$,N%)	returns a substring of the string A$ from the first character to the Nth character (the leftmost N characters)
	Y$=RIGHT(A$,N%)	returns a substring of the string A$ from the Nth to the last character; the rightmost characters of the strin starting with the Nth character.
	Y$=MID(A$,N1%,N2%)	returns a substring of the string A$ starting with the N1 and being N2 characters long (the characters between and including the N1 to N1+N2-1 characters).
	Y%=LEN(A$)	returns the number of characters in the string A$, including trailing blanks.
	Y%=INSTR(N1%,A$,B$)	indicates a search for the substring B$ within the string A$ beginning at character position N1. Returns a value 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).
	Y$=SPACE$(N%)	indicates a string of N spaces, used to insert spaces within a character string.
	Y$=NUM$(N%)	indicates a string of numeric characters representing the value of N as it would be output by a PRINT statement. For example: NUM$(1.ØØØØ) = (space)1(space) and NUM$(-1.ØØØØ) = -1(space).
	Y=VAL(A$)	computes the numeric value of the string of numeric characters A$. If A$ contains any character not acceptable as numeric input with the INPUT statement, an error results. For example: VAL("15")=15
	Y$=XLATE(A$,B$)	translate A$ to the new string Y$ by means of the table string B$, see Section 12.7.
System	Y$=DATE$(Ø%)	returns the current date in the following format: Ø2-Mar-71

Type	Function	Explanation
System, cont'd.	Y$=DATE$(N%)	returns a character string correspond- to a calendar date as follows: N=(day of year)+[(number of years since 1970)*1000] DATE$(1) = "Ø1-Jan-7Ø" DATE$(125) = "Ø5-May-7Ø"
	Y$=TIMES$(Ø%)	returns the current time of day as a character string as follows: TIME$(Ø)="Ø5:3Ø PM" or"17:3Ø "
	Y$=TIME$(N%)	returns a string corresponding to the time at N minutes before midnight, for example: TIME$(1) = "11:59 PM" or 23:59 " TIME$(144Ø)= "12:ØØ AM" or ØØ:ØØ " TIME$(721) = "11:59 AM" or 11:59 "
	Y=TIME(Ø%)	returns the clock time in seconds since midnight, as a floating point number.
	Y=TIME(1%)	returns the central processor time used by the current job in tenths of seconds.
	Y=TIME(2%)	returns the connect time (during which the user is logged into the system) for the current job in minutes.
	Y=TIME(3%)	returns to Y the decimal number of kilo-core ticks (kct's) used by this job. See Section 8.8.
	Y=TIME(4%)	returns to Y the decimal number of minutes of device time used by this job. See Section 8.8.
	Y%=STATUS	returns to Y% the status of the most recently OPENED statement executed in the program. See Section 12.3.5.
	Y%=BUFSIZ(N)	returns to Y% the buffer size of the device or file open on channel N. See Section 12.3.4.
	Y%=LINE	returns to Y% the line number of the statement being executed at the time of an interrupt. See Section 4.5.
	Y%=ERR	returns value associated with the last encountered error if an ON ERROR GOTO statement appears in the program. See Section 8.4.
	Y%=ERL	returns the line number at which the last error occurred if an ON ERROR GOTO statement appears in the program. See Section 8.4.3.
	Y%=SWAP%(N%)	causes a byte swap operation on the two bytes in the integer variable N%.

Type	Function	Explanation
	Y$=RAD$(N%)	converts an integer value to a 3-character string and is used to convert from Radix-5Ø format back to ASCII. See Appendix D.
Matrix	MAT Y=TRN(X)	returns the transpose of the matrix X, see Section 7.6.2.
	MAT Y=INV(X)	returns the inverse of the matrix X, see Section 7.6.2.
	Y=DET	following an INV(X) function evaluation, the variable DET is equivalent to the determinant of X.
	Y%=NUM	following input of a matrix, NUM contains the number of rows input, or in the case of a one dimensional matrix, the number of elements entered.
	Y%=NUM2	following input of a matrix, NUM2 contains the number of elements entered in that row.
Input/Output	Y%=RECOUNT	returns the number of characters read following every input operation. Used primarily with non-file structured devices. See Section 12.3.1.

A.4 SUMMARY OF BASIC-PLUS STATEMENTS

The following summary of statements available in the BASIC-PLUS language defines the general format for the statement as a line in a BASIC program. If more detailed information is needed, the reader is referred to the section(s) in the manual dealing with that particular statement.

In these definitions, elements in angle brackets are necessary elements of the statement. Elements in square brackets are necessary elements of which the statement may contain one. Elements in braces are optional elements of the statement.

Where the term line number ({*line number*}) is shown in braces, this statement can be used in immediate mode.

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.
constant	Any acceptable integer constant (need not contain a % character).
argument(s) or *arg*	Dummy variable names.
statement	Any legal BASIC-PLUS statement.
string	Any legal string constant or variable as described in Section 5.1.
protection	Any legal protection code as described in Section 9.1.
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
   {line number}  REM <message>                                     3.1
   {line number}{<statement>}!<message>
             1Ø  REM  THIS IS A COMMENT
             15  PRINT   !PERFORM A CR/LF

LET
   {line number}{LET}<var>{,<var>,<var>...} = <exp>                 3.2
             55  LET A=4Ø:  B=22
             6Ø  B,C,A=4.2      !MULTIPLE ASSIGNMENT

DIM
    line number   DIM<var(dimension(s))>                            3.6.2
             1Ø   DIM A(2Ø), B$(5,1Ø), C%(45)                         7.1

    line number   DIM #<constant>,<var(dimension(s))>=<constant>   11.1
             75   DIM #4, A$(1ØØ)=32,B(5Ø,5Ø)


RANDOMIZE                                                           3.7.2
    line number  RANDOM{IZE}
             55  RANDOMIZE
             7Ø  RANDOM
IF-THEN, IF-GOTO            ┌THEN<statement>     ┐
    line number  IF <cond> │ THEN<line number>  │                 3.5
                            └GOTO<line number>   ┘
             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                                                        8.5
                            ┌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┐             8.6
    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
             Ø5  FOR B= 1 UNTIL B>1Ø

NEXT                                                                3.6.1
    line number  NEXT <var>
             25  NEXT I
             6Ø  NEXT N
```

Statement Formats and Examples — Manual Section

DEF, single line — 3.7.3, 5.5.1, 6.4
```
    line number  DEF FN<var>(arg) =<exp(arg)>
             2Ø  DEF FNA(X,Y,Z)=SQR(X↑2+Y↑2+Z↑2)
```

DEF, multiple line — 8.1
```
    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 — 8.2
```
    line number  ON <exp> GOTO <list of line numbers>
             75  ON X GOTO 95, 15Ø, 45, 2ØØ
```

GOSUB — 3.8.1
```
    line number  GOSUB <line number>
             9Ø  GOSUB 2ØØ
```

ON-GOSUB — 8.3
```
    line number  ON <exp> GOSUB <list of line numbers>
             85  ON FNA(M) GOSUB 2ØØ, 25Ø, 4ØØ, 375
```

RETURN — 3.8.2
```
    line number  RETURN
            375  RETURN
```

CHANGE — 5.2
```
  {line number}  CHANGE [<array name>]  TO  [<string var> ]
                        [<string var>]      [<array name>]
             25  CHANGE A$ TO X
             7Ø  CHANGE M TO R$
             75  CHANGE B TO B$
```

OPEN — 9.2, 9.2.1, 9.2.2
```
  {line number}  OPEN<string>{FOR[INPUT ]}AS FILE <exp>
                                 [OUTPUT]
         {,RECORDSIZE<exp>}{,CLUSTERSIZE <exp>}{,MODE <exp>}
             1Ø  OPEN "PP:" FOR OUTPUT AS FILE B1
             2Ø  OPEN "FOO" AS FILE 3
             3Ø  OPEN "DT4:DATA.TR" FOR INPUT AS FILE 1Ø
```

CLOSE — 9.3
```
  {line number}  CLOSE <list of exp>
            1ØØ  CLOSE 2
            255  CLOSE 1Ø, 4, N1
```

READ — 3.3.1, 5.3, 6.3, 10.1
```
    line number  READ <list of variables>
             25  READ A, B$, C%, F1, R2, B(25)
```

```
                                                                    Manual
Statement Formats and Examples                                      Section

DATA                                                                  3.3.1
         line number   DATA <list of values>                            5.3
                 3ØØ   DATA 4.3, "STRING",85,49,75.Ø4,1Ø                6.3

RESTORE                                                               3.3.1
         line number   RESTORE                                         10.2
                 125   RESTORE

PRINT                                                                 3.3.2
                                                                        5.4
       {line number}   PRINT{{#<exp>,}<list>}                           6.3
                  25   PRINT  !GENERATES  CR/LF                        10.3
                  75   PRINT "BEGINNING OF OUTPUT";I,A*I             10.3.1
                  45   PRINT #4,"OUTPUT TO DEVICE"FNM(A)↑2;B;A       10.3.2

PRINT USING
       {line number}   PRINT {#<exp>,}USING <string>, <list>         10.3.3
                  54   PRINT USING "##.##",A
                  55   PRINT #3, USING"\\###.##    \\##↑↑↑↑","A=",A,"B=",B
                  56   PRINT #7, USING B$,A,B,C

INPUT                                                                 3.3.3
       {line number}   INPUT {#<exp>,}<list>                            5.3
                  25   INPUT "TYPE YOUR NAME ",A$                       6.3
                  55   INPUT #8, A, N, B$                              10.4
                                                                     10.4.1


INPUT LINE                                                              5.3
       {line number}   INPUT LINE {#<exp>,} <string>
                  4Ø   INPUT LINE  R$
                  75   INPUT LINE  #1, E$

NAME-AS
       {line number}   NAME <string> AS <string>                        9.5
                 455   NAME "NONAME" AS "FILE1<48>"
                 27Ø   NAME "DT4:MATRIX" AS "MATA1<48>"

KILL                                                                    8.4
       {line number}   KILL <string>
                  45   KILL "NONAME"

ON ERROR GOTO
         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                                                                8.4.1
         line number   RESUME {<line number>}
                1ØØØ   RESUME    !OR RESUME Ø ARE EQUIVALENT
                 655   RESUME  2ØØ

CHAIN
         line number    CHAIN <string> {<exp>}                          9.6
                 375    CHAIN "PROG2"
                 5ØØ    CHAIN "PROG3" 75
                 6ØØ    CHAIN "PROG3" A
```

Statement Formats and Examples   Manual Section

STOP   3.9

```
    line number   STOP
             75   STOP
```

END   3.9

```
    line number   END
            545   END
```

Matrix Statements

MAT READ   7.2

```
    line number   MAT READ <list of matrices>
             55   DIM A(2Ø), B$(32), C%(15,1Ø)
             9Ø   MAT READ A, B$(25), C%
```

MAT PRINT   7.3

```
  {line number}   MAT PRINT{#<exp>,} <matrix name>
             1Ø   DIM A (2Ø), B(15,2Ø)
             9Ø   MAT PRINT A;           !PRINT 1Ø*1Ø MATRIX, PACKED
             95   MAT PRINT B(1Ø,5),     !PRINT 1Ø*5 MATRIX, FIVE
                                         !ELEMENTS PER LINE
             97   MAT PRINT #2, A;       !PRINT ON OUTPUT CHANNEL 2
```

MAT INPUT   7.4

```
  {line number}   MAT INPUT{#<exp>,} <list of matrices>
             1Ø   DIM B$(4Ø), F1%(35)
             2Ø   OPEN "DT3:FOO" FOR INPUT AS FILE 3
             3Ø   MAT INPUT #3, B4, F1%
```

MAT Initialization   7.5

```
                                    ┌ZER┐
  {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 B = ZER(N,M)       !CLEARS AN N BY M MATRIX
```

Statement Modifiers (can be used in immediate mode)

IF   8.7.1

```
  <statement> IF  <condition>
             1Ø   PRINT X IF X<>Ø
```

UNLESS   8.7.2

```
  <statement> UNLESS <condition>
             45   PRINT A UNLESS A=Ø
```

FOR   8.7.3

```
  <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   8.7.4

```
  <statement> WHILE <condition>
             1Ø   LET A(I) = FNX(I) WHILE I<45.5
```

Statement Formats and Examples

```
                                                                   Manual
                                                                   Section

UNTIL                                                               8.7.5
    <statement> UNTIL <condition>
             115  IF B Ø THEN A(I)=B UNTIL I>5
```

System statements

```
    <line number> SLEEP  <expression>                                 8.8
             1ØØ  SLEEP 2Ø   !DISMISS JOB FOR 2Ø SEC.

    <line number> WAIT  <expression>                                  8.8
             525  WAIT A%+5   !WAIT A%+5 SEC. FOR INPUT
```

Record I/O Statements

```
    <line number> LSET<string var>{,<string var>}=<string>          12.4.3
              9Ø  LSET B$="XYZ"

    <line number> RSET<string var>{,<string var>}=<string>          12.4.3
             25Ø  RSET C$="6789Ø"

    <line number> FIELD#<expr>,<expr>AS<string var>{,<expr>AS<string var>}
              75  FIELD#2%,1Ø% AS A$, 2Ø% AS B$                     12.4.2

    <line number> GET#<expr>{,RECORD<expr>}                          12.3
             1ØØ  GET#1%,RECORD 99%

    <line number> PUT#<expr>{,RECORD<expr>}{,COUNT<expr>}            12.3
             5ØØ  PUT#1%,COUNT 8Ø%

    <line number> UNLOCK#<expr>                                    10.5.1
             7ØØ  UNLOCK #3%
```

APPENDIX B

BASIC-PLUS COMMAND SUMMARY

Command	Explanation	Section in RSTS-11 System User's Guide
APPEND	Used to include contents of a previously saved source program in current program.	2.4.3
ASSIGN	Used to reserve an I/O device for the use of the individual issuing the command. The specified device can then be given commands only from the job which issued the ASSIGN. Also establishes a logical name for a device, establishes an account for the @ character, and establishes a default protection code.	2.6.3
ATTACH	Attaches a detached job to the current terminal.	4.1
BYE	Indicates to RSTS that a user wishes to leave the terminal. Closes and saves any files remaining open for that user.	2.1.3
CAT CATALOG	Returns the user's file directory. Unless another device is specified following the term CAT or CATALOG, the disk is the assumed device.	2.5.2
CCONT	For privileged users. Same as CONT command, but detaches job from terminal.	2.2.9
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.	2.3.3
CONT	Allows the user to continue execution of the program currently in core following the execution of a STOP statement.	2.2.8
DEASSIGN	Used to release the specified device for use by others. If no particular device is specified, all devices assigned to that terminal are released. An automatic DEASSIGN is performed when the BYE command is given. Also releases any logical name for a device.	2.6.4 2.7
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 dash (-) indicating the first and last line of the section of code to be removed. Several single lines or line sections can be indicated by separating the line numbers, or line number pairs, with a comma.	2.2.5

Command	Explanation	Section in RSTS-11 System User's Guide
HELLO	Indicates to RSTS that a user wishes to log onto the system. Allows the user to input project-programmer number and password. Also attaches a detached job to the current terminal or changes accounts without having to log off the system.	2.1.2
KEY	Used to re-enable the echo feature on the user terminal following the issue of a TAPE command. Enter with LINE FEED or ESCAPE key.	2.6.2
LENGTH	Returns the length of the user's current program in core, in 1K increments.	2.5.1
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 dash (-) will list the lines between and including the lines indicated. Several single lines or line sections can be indicated by separating the line numbers, or line number pairs, with a comma.	2.2.4
LISTNH	Same as LIST, but does not print header containing the program name and current data.	2.2.4
LOGIN	Same as HELLO on Version 5 (RSTS/E) systems.	4.1
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.	2.2.1
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. A device specification without a filename will cause a program to be read from an input-only device (such as high-speed reader, card reader).	2.4.2
REASSIGN	Transfers control of a device to another job.	2.6.5
RENAME	Causes the name of the program currently in core to be changed to the name specified after the word RENAME.	2.2.6
REPLACE	Same as SAVE, but allows the user to substitute a new program for an old program with the same name, erasing the old program.	2.4.7

Command	Explanation	Section in RSTS-11 System User's Guide
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 (if necessary), 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).	2.3.1
RUNNH	Causes execution of the program currently in memory but header information containing the program name and current data is not printed. If a filename is used, the command is executed as if no filename were given.	2.3.1
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).	2.4.1
SCALE	Sets the scale factor to a designated value or prints the value(s) currently in effect if no value is designated.	2.8
TAPE	Used to disable the echo feature on the user terminal while reading paper tape via the low-speed reader.	2.6.1
UNSAVE	The word UNSAVE is followed by the file name and, optionally, the extension of the file to be removed. The UNSAVE command cannot remove files without an extension. If no extension is specified, the source (.BAS) file is deleted. If no device is specified, the disk is assumed.	2.4.6

Command	Explanation	Section in RSTS-11 System User's Guide
Special Control Character Summary		
CTRL/C	Causes the system to return to BASIC command mode to allow for issuing of further commands or editing. Echoes on terminal as ↑C.	3.5
CTRL/O	Used as a switch to suppress/enable output of a program on the user terminal. Echoes as ↑O.	3.7
CTRL/Q	When generated by a device on which a CTRL/S has interrupted output, causes computer to resume output at the next character.	3.10
CTRL/S	When generated by a device for which STALL characteristics are set, interrupts computer output on the device until either CTRL/Q or another character is generated.	3.10
CTRL/U	Deletes the current typed line, echoes as ↑U and performs a carriage return/line feed.	3.6
CTRL/Z	Used as an end-of-file character.	3.9
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.	3.2
LINE FEED Key	Used to continue the current logical line on an additional physical line. Performs a line feed/carriage return operation.	3.3
RETURN Key	Enters a typed line to the system, results in a carriage return/line feed operation at the user terminal.	3.1
RUBOUT Key	Deletes the last character typed on that physical line. Erased characters are shown on the teleprinter between back slashes.	3.4
TAB or CTRL/I	Performs a tabulation to the next of nine tab stops (eight spaces apart) which form the terminal printing line.	3.8

APPENDIX C

ERROR MESSAGE SUMMARY

Wherever possible, RSTS follows an error message with the phrase

```
        AT LINE xxxx
```

where xxxx is the line number of the statement which caused the error. For example:

```
        1Ø  TALK
        ILLEGAL VERB AT LINE 1Ø
        READY
```

The additional message is not printed when no line number can be associated with the error.

```
        TALK

        WHAT?

        READY
```

An (SPR) in the description of any error message in this Appendix indicates an error which should never be seen by a user. If such a message is received, the user should document how he obtained the error and file a Software Performance Report with DEC, including the pertinent output.

C.1 USER RECOVERABLE ERRORS

A (C) in the description of the error message indicates that program execution continues, following printing of the error message, if an ON ERROR GOTO statement is not present. Normally, execution terminates on an error condition, the error message is printed, and the system prints READY. The ERR column gives the value of the ERR variable (see Section 8.4).

ERR	Message Printed	Meaning
1	BAD DIRECTORY FOR DEVICE	The directory of the device referenced is in an unreadable format.

ERR	Message Printed	Meaning
2	ILLEGAL FILE NAME	The filename specified is not acceptable. It contains unacceptable characters or the filename specification format has been violated.
3	ACCOUNT OR DEVICE IN USE	Removal or dismounting of the account or device cannot be done since one or more users are currently using it.
4	NO ROOM FOR USER ON DEVICE	Storage space allowed for the current user on the device specified has been used or the device as a whole is too full to accept further data.
5	CAN'T FIND FILE OR ACCOUNT	The file or account number specified was not found on the device specified.
6	NOT A VALID DEVICE	Attempt to use an illegal or nonexistent device specification.
7	I/O CHANNEL ALREADY OPEN	An attempt was made to open one of the twelve I/O channels which had already been opened by the program. (SPR)
8	DEVICE NOT AVAILABLE	The device requested is currently reserved by another user.
9	I/O CHANNEL NOT OPEN	Attempt to perform I/O on one of the twelve channels which has not been previously opened in the program.
10	PROTECTION VIOLATION	The user was prohibited from performing the requested operation because the kind of operation was illegal (such as input from a line printer) or because the user did not have the privileges necessary (such as deleting a protected file).
11	END OF FILE ON DEVICE	Attempt to perform input beyond the end of a data file; or a BASIC source file is called into memory and is found to contain no END statement.
12	FATAL SYSTEM I/O FAILURE	An I/O error has occurred on the system level. The user has no guarantee that the last operation has been performed. (SPR)
13	USER DATA ERROR ON DEVICE	One or more characters may have been transmitted incorrectly due to a parity error, bad punch combination on a card, or similar error.
14	DEVICE HUNG OR WRITE LOCKED	User should check hardware condition of device requested. Possible causes of this error include a line printer out of paper or high-speed reader being off-line.

ERR	Message Printed	Meaning
15	KEYBOARD WAIT EXHAUSTED	Time requested by Wait statement has been exhausted with no input received from the specified keyboard.
16	NAME OR ACCOUNT NOW EXISTS	An attempt was made to rename a file with the name of a file which already exists, or an attempt was made by the system manager to insert an account number which is already within the system.
17	TOO MANY OPEN FILES ON UNIT	Only one open DECtape output file is permitted per DECtape drive. Only one open file per magtape drive is permitted.
18	ILLEGAL SYS() USAGE	Illegal use of the SYS system function.
19	DISK BLOCK IS INTERLOCKED	The requested disk block segment is already in use (locked) by some other user.
20	PACK IDS DON'T MATCH	The identification code for the specified disk pack does not match the identification code already on the pack.
21	DISK PACK IS NOT MOUNTED	No disk pack is mounted on the specified disk drive.
22	DISK PACK IS LOCKED OUT	The disk pack specified is mounted but temporarily disabled.
23	ILLEGAL CLUSTER SIZE	The specified cluster size is unacceptable.
24	DISK PACK IS PRIVATE	The current user does not have access to the specified private disk pack.
25	DISK PACK NEEDS 'CLEANING'	Non-fatal disk mounting error; use the CLEAN operation in UTILTY.
26	FATAL DISK PACK MOUNT ERROR	Fatal disk mounting error. Disk cannot be successfully mounted.
27	I/O TO DETACHED KEYBOARD	I/O was attempted to a hung up dataset or to the previous, but now detached, console keyboard for the job.
28	PROGRAMMABLE ↑C TRAP	ON ERROR-GOTO subroutine was entered through a program trapped CTRL/C. See a description of the SYS system function.
29	CORRUPTED FILE STRUCTURE	Fatal error in CLEAN operation.

ERR	Message Printed	Meaning
30	DEVICE NOT FILE STRUCTURED	An attempt is made to access a device, other than a disk, DECtape, or magtape device, as a file-structured device. This error occurs, for example, when the user attempts to gain a directory listing of a non-directory device.
31	ILLEGAL BYTE COUNT FOR I/O	The buffer size specified in the RECORDSIZE option of the OPEN statement or in the COUNT option of the PUT statement is not a multiple of the block size of the device being used for I/O, or is illegal for the device.
32	NO ROOM AVAILABLE FOR FCB	When the user accesses a file under programmed control in RSTS-11, a system control structure called an FCB requires one small buffer and one small buffer is not available for the FCB.
33	UNIBUS TIMEOUT FATAL TRAP	This hardware error occurs when an attempt is made to address nonexistent memory or an odd address using the PEEK function. An occurrence of this error message in any other case is cause for an SPR.
34	RESERVED INSTRUCTION TRAP	An attempt is made to execute an illegal or reserved instruction or an FPP instruction when floating point hardware is not available.
35	MEMORY MANAGEMENT VIOLATION	This hardware error occurs when an illegal Monitor address is specified using the PEEK function. Generation of the error message in situations other than using PEEK is cause for an SPR.
36	SP (R6) STACK OVERFLOW	An attempt to extend the hardware stack beyond its legal size is encountered. (SPR)
37	DISK ERROR DURING SWAP	A hardware error occurs when a user's job is swapped into or out of memory. The contents of the user's job area are lost but the job remains logged into the system and is reinitialized to run the NONAME program.
38	MEMORY PARITY FAILURE	A parity error was detected in the memory occupied by this job.
39	MAGTAPE SELECT ERROR	When access to a magtape drive was attempted, the selected unit was found to be off line.
40	MAGTAPE RECORD LENGTH ERROR	When performing input from magtape, the record on magtape was found to be longer than the buffer designated to handle the record.

ERR	Message Printed	Meaning
41	NO RUN-TIME SYSTEM	Reserved.
42	VIRTUAL BUFFER TOO LARGE	Virtual core buffers must be 512 bytes long.
43	VIRTUAL ARRAY NOT ON DISK	A non-disk device is open on the channel upon which the virtual array is referenced.
44	MATRIX OR ARRAY TOO BIG	In-core array size is too large.
45	VIRTUAL ARRAY NOT YET OPEN	An attempt was made to use a virtual array before opening the corresponding disk file.
46	ILLEGAL I/O CHANNEL	Attempt was made to open a file on an I/O channel outside the range of the integer numbers 1 to 12.
47	LINE TOO LONG	Attempt to input a line longer than 255 characters (which includes any line terminator). Buffer overflows.
48	FLOATING POINT ERROR	Attempt to use a computed floating point number outside the range $\|1E-38\| \le n \le \|1E38\|$ excluding zero. If no transfer to an error handling routine is made, zero is returned as the floating point value. (C)
49	ARGUMENT TOO LARGE IN EXP	Acceptable arguments are within the approximate range $-89 \le arg \le +88$. The value returned is zero. (C)
50	DATA FORMAT ERROR	A READ or INPUT statement detected data in an illegal format. For example, 1..2 is an improperly formed number, and 1.3 is an improperly formed integer, and X" is an illegal string. (C)
51	INTEGER ERROR	Attempt to use a computed integer outside the range $-32767 \le n \le 32767$. If no transfer to an error handling routine is made, zero is returned as the integer value. (C)
52	ILLEGAL NUMBER	Integer or floating point overflow or underflow.
53	ILLEGAL ARGUMENT IN LOG	Negative or zero argument to log function. Value returned is the argument as passed to the function. (C)
54	IMAGINARY SQUARE ROOTS	Attempt to take square root of a number less than zero. The value returned is the square root of the absolute value of the argument. (C)
55	SUBSCRIPT OUT OF RANGE	Attempt to reference an array element beyond the number of elements created for the array when it was dimensioned.

ERR	Message Printed	Meaning
56	CAN'T INVERT MATRIX	Attempt to invert a singular or nearly singular matrix.
57	OUT OF DATA	The DATA list was exhausted and a READ requested additional data.
58	ON STATEMENT OUT OF RANGE	The index value in an ON-GOTO or ON-GOSUB statement is less than one or greater than the number of line numbers in the list.
59	NOT ENOUGH DATA IN RECORD	An INPUT statement did not find enough data in one line to satisfy all the specified variables.
60	INTEGER OVERFLOW, FOR LOOP	The integer index in a FOR loop attempted to go beyond 32766 or below -32766.
61	DIVISION BY Ø	Attempt by the user program to divide some quantity by zero. If no transfer is made to an error handler routine, a Ø is returned as the result. (C)

C.2 NON-RECOVERABLE ERRORS

Message Printed	Meaning
ARGUMENTS DON'T MATCH	Arguments in a function call do not match, in number or in type, the arguments defined for the function.
BAD LINE NUMBER PAIR	Line numbers specified in a LIST or DELETE command were formatted incorrectly.
BAD NUMBER IN PRINT-USING	Format specified in the PRINT-USING string cannot be used to print one or more values.
CAN'T COMPILE STATEMENT	
CAN'T CONTINUE	Program was stopped or ended at a spot from which execution cannot be resumed.
CATASTROPHIC ERROR	The user program data structures are destroyed. This normally indicates a BASIC-PLUS malfunction and, if reproducible, should be reported to DEC on a Software Performance Report form (SPR).
DATA TYPE ERROR	Incorrect usage of floating-point, integer, or character string format variable or constant where some other data type was necessary.

Message Printed	Meaning
DEF WITHOUT FNEND	A second DEF statement was encountered in the processing of a user function without an FNEND statement terminating the first user function definition.
END OF STATEMENT NOT SEEN	Statement contains too many elements to be processed correctly.
EXECUTE ONLY FILE	Attempt was made to add, delete or list a statement in a compiled (.BAC) format file.
EXPRESSION TOO COMPLICATED	This error usually occurs when parentheses have been nested too deeply. The depth allowable is dependent on the individual expression.
FIELD OVERFLOWS BUFFER	Attempt to use FIELD to allocate more space than exists in the specified buffer.
FILE EXISTS-RENAME/REPLACE	A file of the name specified in a SAVE command already exists. In order to save the current program under the name specified, use REPLACE, or RENAME followed by SAVE.
FNEND WITHOUT DEF	An FNEND statement was encountered in the user program without a previous DEF statement being seen.
FNEND WITHOUT FUNCTION CALL	A FNEND statement was encountered in the user program without a previous function call having been executed.
FOR WITHOUT NEXT	A FOR statement was encountered in the user program without a corresponding NEXT statement to terminate the loop.
ILLEGAL CONDITIONAL CLAUSE	Incorrectly formatted conditional expression.
ILLEGAL DEF NESTING	The range of one function definition crosses the range of another function definition.
ILLEGAL DUMMY VARIABLE	One of the variables in the dummy variable list of a user-defined function is not a legal variable name.
ILLEGAL EXPRESSION	Double operators, missing operators, mismatched parentheses, or some similar error has been found in an expression.

Message Printed	Meaning
ILLEGAL FIELD VARIABLE	The FIELD variable specified is unacceptable.
ILLEGAL FN REDEFINITION	Attempt was made to redefine a user function.
ILLEGAL FUNCTION NAME	Attempt was made to define a function with a function name not subscribing to the established format.

Message Printed	Meaning
ILLEGAL IF STATEMENT	Incorrectly formatted IF statement.
ILLEGAL IN IMMEDIATE MODE	User issued a statement for execution in immediate mode which can only be performed as part of a program.
ILLEGAL LINE NUMBER(S)	Line number reference outside the range $1 \leq n \leq 32767$.
ILLEGAL MAGTAPE() USAGE	Improper use of the MAGTAPE function.
ILLEGAL MODE MIXING	String and numeric operations cannot be mixed.
ILLEGAL STATEMENT	Attempt was made to execute a statement that did not compile without errors.
ILLEGAL SYMBOL	An unrecognizable character was encountered. For example, a line consisting of a # character.
ILLEGAL VERB	The BASIC verb portion of the statement cannot be recognized.
INCONSISTENT FUNCTION USAGE	A function is being redefined in a manner inconsistent in the number or type of arguments with one or more calls to that function existing in the program.
INCONSISTENT SUBSCRIPT USE	A subscripted variable is being used with a different number of dimensions from the number with which it was originally defined.
K OF CORE USED	Message printed by LENGTH command, preceded by the appropriate number describing the user program currently in core to the nearest 1K.
LITERAL STRING NEEDED	A variable name was used where a numeric or character string was necessary.
MATRIX DIMENSION ERROR	Attempt was made to dimension a matrix to more than two dimensions, or an error was made in the syntax of a DIM statement.
MATRIX OR ARRAY WITHOUT DIM	A matrix or array element was referenced beyond the range of an implicitly dimensioned matrix.
MAXIMUM CORE EXCEEDED	User program grew to be too large to run or compile in the area of core assigned to each user at the given installation.
MISSING SPECIAL FEATURE	User program employs a BASIC-PLUS feature not present on the given installation.

Message Printed	Meaning
MODIFIER ERROR	Attempt to use one of the statement modifiers (FOR, WHILE, UNTIL, IF, or UNLESS) incorrectly.
NEXT WITHOUT FOR	A NEXT statement was encountered in the user program without a previous FOR statement having been seen.
NO LOGINS	Message printed if the system is full and cannot accept additional users or if further logins are disabled by the system manager.
NOT A RANDOM ACCESS DEVICE	Attempt to perform random access I/O to a non-random access device.
NOT ENOUGH AVAILABLE CORE	The already compiled user program is too large to run in the area of core assigned to each user at the given installation.
NUMBER IS NEEDED	A character string or variable name was used where a number was necessary.
1 OR 2 DIMENSIONS ONLY	Attempt was made to dimension a matrix to more than two dimensions.
ON STATEMENT NEEDS GOTO	A statement beginning with ON does not contain a GOTO or GOSUB clause.
PLEASE SAY HELLO	User not logged into the system has typed something other than a legal, logged-out command to the system.
PLEASE USE THE RUN COMMAND	A transfer of control (as in a GOTO, GOSUB or IF-GOTO statement) cannot be performed from immediate mode.
PRINT-USING BUFFER OVERFLOW	Format specified contains a field too large to be manipulated by the PRINT-USING statement.
PRINT-USING FORMAT ERROR	An error was made in the construction of the string used to supply the output format in a PRINT-USING statement.
PROGRAM LOST-SORRY	A fatal system error has occurred which caused the user program to be lost.
REDIMENSIONED ARRAY	Usage of an array or matrix within the user program has caused BASIC-PLUS to redimension the array implicitly.
RESUME AND NO ERROR	A RESUME statement was encountered where no error had occurred to cause a transfer into an error handling routine via the ON ERROR-GOTO statement.

Message Printed	Meaning
RETURN WITHOUT GOSUB	RETURN statement encountered in the user program without a previous GOSUB statement having been executed.
SCALE FACTOR INTERLOCK	An attempt was made to execute a program or source statement with the current scale factor. The program executes but the system uses the scale factor of the program in memory. Use REPLACE and OLD or recompile the program to execute with the current scale factor.
STATEMENT NOT FOUND	Reference is made within the program to a line number which is not within the program.
STOP	STOP statement was executed. The user can usually continue program execution by typing CONT and the RETURN key.
STRING IS NEEDED	A number or variable name was used where a character string was necessary.
SYNTAX ERROR	BASIC-PLUS statement was incorrectly formatted.
TEXT TRUNCATED	No BASIC-PLUS statement can be more than 255 characters long.
TOO FEW ARGUMENTS	The function has been called with a number of arguments not equal to the number defined for the function.
TOO MANY ARGUMENTS	A user-defined function may have up to five arguments.
UNDEFINED FUNCTION CALLED	BASIC-PLUS interpreted some statement component as a function call for which there is no defined function (system or user).
WHAT?	Command or immediate mode statement entered to BASIC-PLUS could not be processed. Illegal verb or improper format error most likely.
WRONG MATH PACKAGE	Program was compiled with an incompatible version of RSTS. Program source must be recompiled.

C.3 SYSTEM IDENTIFICATION MESSAGE

ERR code Ø is associated with the system installation name for use by the system programs.

APPENDIX D

BASIC-PLUS CHARACTER SET

D.1 BASIC-PLUS CHARACTER SET

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

A through Z
Ø through 9
Space
Tab

and the following special symbols and keys:

Key	Use and Section in BASIC-PLUS Language Manual
$	Used in specifying string variables (Section 5.1), or as the System Library file designator (RSTS-11 System User's Guide).
%	Used in specifying integer variables (Section 6.1). Also denotes account [1,4] (Section 9.1.1).
' "	Used to delimit string constants, i.e., text strings (Section 5.1).
!	Begins comment part of a line (Section 3.1). Also denotes account [1,3] (Section 9.1.1).
:	Separates multiple statements on one line (Section 2.3.1).
\	Separates multiple statements on one line as the colon (:) also does.
#	Denotes a device or file channel number, or is used as an output format effector (Chapter 7 and Section 10.3.3). Also denotes account number using current project number with a programmer number of Ø (Section 9.1.1).
,	Output format effector and list terminator (Section 3.3).
;	Output format effector (Section 3.3).
&	Denotes account [1,5] (Section 9.1.1).
@	Denotes the assignable account (Section 9.1.1).
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), or to delimit project-programmer number.
[]	Used to group project-programmer number. Equivalent to ().

Key	Use and Section in BASIC-PLUS Language Manual
< >	Used to delimit file protection codes.
+ - */↑	Arithmetic operators (Section 2.5.3).
=	Replacement operator (Section 3.2). Logical equivalence operator (Section 2.5.4).
<	Logical "less than" operator (Section 2.5.4).
>	Logical "greater than" operator (Section 2.5.4).
==	Numeric "approximately equal to" operator (Section 2.5.4). Logical "exactly equal to" string operator (Section 5.1.5).

D.2 ASCII CHARACTER CODES

Decimal Value	ASCII Char- acter	RSTS Usage	Decimal Value	ASCII Char- acter	RSTS Usage	Decimal Value	ASCII Char- acter	RSTS Usage
Ø	NUL	FILL character	43	+		86	V	
1	SOH		44	'	COMMA	87	W	
2	STX		45	-		88	X	
3	ETX	CTRL/C	46	.		89	Y	
4	EOT		47	/		9Ø	Z	
5	ENQ		48	Ø		91	[	
6	ACK		49	1		92	\	Backslash
7	BEL	BELL	5Ø	2		93	]	
8	BS		51	3		94	^	or ↑
9	HT	HORIZONTAL TAB	52	4		95	_	or ←
1Ø	LF	LINE FEED	53	5		96	`	Grave accent
11	VT	VERTICAL TAB	54	6		97	a	
12	FF	FORM FEED	55	7		98	b	
13	CR	CARRIAGE RETURN	56	8		99	c	
14	SO		57	9		1ØØ	d	
15	SI	CTRL/O	58	:		1Ø1	e	
16	DLE		59	;		1Ø2	f	
17	DC1		6Ø	<		1Ø3	g	
18	DC2		61	=		1Ø4	h	
19	DC3		62	>		1Ø5	i	
2Ø	DC4		63	?		1Ø6	j	
21	NAK	CTRL/U	64	@		1Ø7	k	
22	SYN		65	A		1Ø8	l	
23	ETB		66	B		1Ø9	m	
24	CAN		67	C		11Ø	n	
25	EM		68	D		111	o	
26	SUB	CTRL/Z	69	E		112	p	
27	ESC	ESCAPE[1]	7Ø	F		113	q	
28	FS		71	G		114	r	
29	GS		72	H		115	s	
3Ø	RS		73	I		116	t	
31	US		74	J		117	u	
32	SP	SPACE	75	K		118	v	
33	!		76	L		119	w	
34	"		77	M		12Ø	x	
35	#		78	N		121	y	
36	$		79	O		122	z	
37	%		8Ø	P		123	{	
38	&		81	Q		124	|	Vertical Line
39	'	APOSTROPHE	82	R		125	}	
4Ø	(		83	S		126	~	Tilde
41	)		84	T		127	DEL	RUBOUT
42	*		85	U				

[1]ALTMODE (ASCII 125) or PREFIX (ASCII 126) keys which appear on some terminals are translated internally into ESCAPE.

Radix-5Ø Character/Position Table

Single Char. or First Char.		Second Character		Third Character	
A	ØØ31ØØ	A	ØØØØ5Ø	A	ØØØØØ1
B	ØØ62ØØ	B	ØØØ12Ø	B	ØØØØØ2
C	Ø113ØØ	C	ØØØ17Ø	C	ØØØØØ3
D	Ø144ØØ	D	ØØØ24Ø	D	ØØØØØ4
E	Ø175ØØ	E	ØØØ31Ø	E	ØØØØØ5
F	Ø226ØØ	F	ØØØ36Ø	F	ØØØØØ6
G	Ø257ØØ	G	ØØØ43Ø	G	ØØØØØ7
H	Ø31ØØØ	H	ØØØ5ØØ	H	ØØØØ1Ø
I	Ø341ØØ	I	ØØØ55Ø	I	ØØØØ11
J	Ø372ØØ	J	ØØØ62Ø	J	ØØØØ12
K	Ø423ØØ	K	ØØØ67Ø	K	ØØØØ13
L	Ø454ØØ	L	ØØØ74Ø	L	ØØØØ14
M	Ø5Ø5ØØ	M	ØØ1Ø1Ø	M	ØØØØ15
N	Ø536ØØ	N	ØØ1Ø6Ø	N	ØØØØ16
O	Ø567ØØ	O	ØØ113Ø	O	ØØØØ17
P	Ø62ØØØ	P	ØØ12ØØ	P	ØØØØ2Ø
Q	Ø651ØØ	Q	ØØ125Ø	Q	ØØØØ21
R	Ø7Ø2ØØ	R	ØØ132Ø	R	ØØØØ22
S	Ø733ØØ	S	ØØ137Ø	S	ØØØØ23
T	Ø764ØØ	T	ØØ144Ø	T	ØØØØ24
U	1Ø15ØØ	U	ØØ151Ø	U	ØØØØ25
V	1Ø46ØØ	V	ØØ156Ø	V	ØØØØ26
W	1Ø77ØØ	W	ØØ163Ø	W	ØØØØ27
X	113ØØØ	X	ØØ17ØØ	X	ØØØØ3Ø
Y	1161ØØ	Y	ØØ175Ø	Y	ØØØØ31
Z	1212ØØ	Z	ØØ2Ø2Ø	Z	ØØØØ32
$	1243ØØ	$	ØØ2Ø7Ø	$	ØØØØ33
.	1274ØØ	.	ØØ214Ø	.	ØØØØ34
unused	1325ØØ	unused	ØØ221Ø	unused	ØØØØ35
Ø	1356ØØ	Ø	ØØ226Ø	Ø	ØØØØ36
1	14Ø7ØØ	1	ØØ233Ø	1	ØØØØ37
2	144ØØØ	2	ØØ24ØØ	2	ØØØØ4Ø
3	1471ØØ	3	ØØ245Ø	3	ØØØØ41
4	1522ØØ	4	ØØ252Ø	4	ØØØØ42
5	1553ØØ	5	ØØ257Ø	5	ØØØØ43
6	16Ø4ØØ	6	ØØ264Ø	6	ØØØØ44
7	1635ØØ	7	ØØ271Ø	7	ØØØØ45
8	1666ØØ	8	ØØ276Ø	8	ØØØØ46
9	1717ØØ	9	ØØ3Ø3Ø	9	ØØØØ47

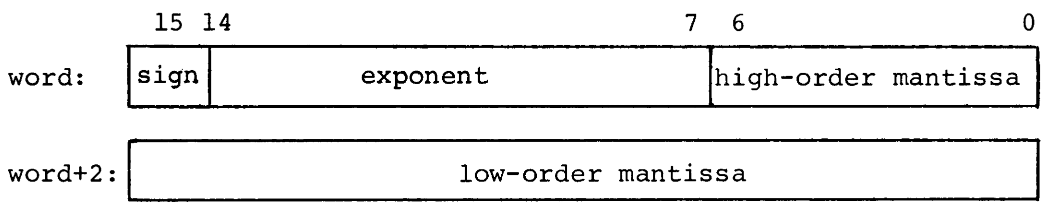
APPENDIX E

RSTS FLOATING-POINT AND INTEGER FORMATS

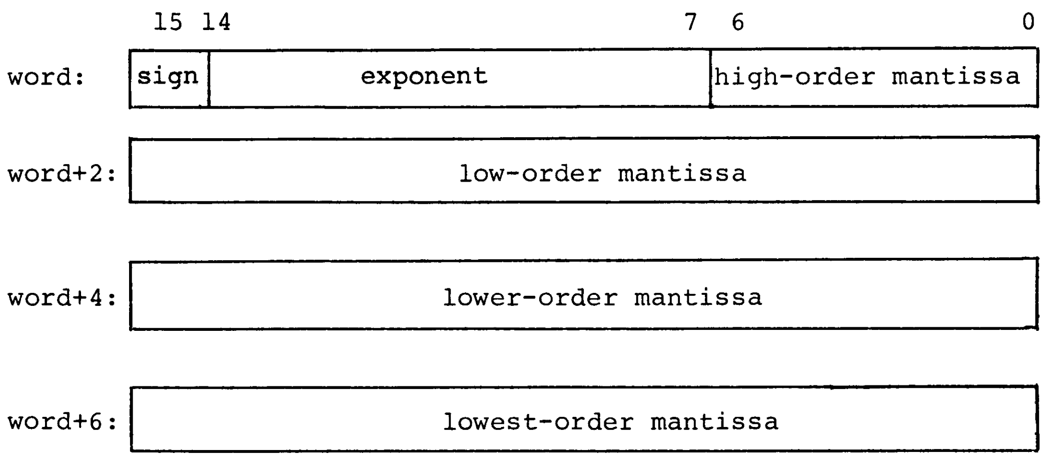
E.1 FLOATING-POINT FORMATS

RSTS systems use two standard floating-point packages: the single precision, two-word package or the double precision, four-word package. The determination of which package will be used is made by the system manager at the time the RSTS Monitor is built.

The single precision format provides economical storage, while the double precision format is used for high accuracy. The single precision format provides up to 24 bits or approximately seven decimal digits of accuracy. The magnitude range lies between Ø.29 x 1Ø$^{-38}$ and 1.7 x 1Ø^{38}. Double precision calculations have a precision of 56 bits or approximately sixteen decimal digits, with magnitudes in the same range as for single precision format.



SINGLE PRECISION FORMAT (2 WORD)



DOUBLE PRECISION FORMAT (4 WORD)

The exponent is stored in excess 128 ($2ØØ_8$) notation. Exponents from -127 to +127 are represented by the binary equivalent of 1 through 255 (1 through 377_8). Fractions are represented in sign magnitude notation with the binary radix point to the left. Numbers are assumed to be normalized and, therefore, the most significant bit is not stored because it is redundant (this is called "hidden bit normalization"); it is always a 1 unless the exponent is Ø in which case it is assumed to be Ø. The value Ø is therefore represented by two or four words of zeroes. For example: +1 would be represented by:

```
word:      Ø4Ø2ØØ
word+2:    ØØØØØØ
```

in the 2-word format, or:

```
word:      Ø4Ø2ØØ
word+2:    ØØØØØØ
word+4:    ØØØØØØ
word+6:    ØØØØØØ
```

in the 4-word format. -5 would be:

```
word:      14Ø64Ø
word+2:    ØØØØØØ
```

in the 2-word format, or:

```
word:      14Ø64Ø
word+2:    ØØØØØØ
word+4:    ØØØØØØ
word+6:    ØØØØØØ
```

in the 4-word format.

While it is generally possible to run programs written on one RSTS system on another RSTS system, certain restrictions apply if the math packages are not the same. These are:

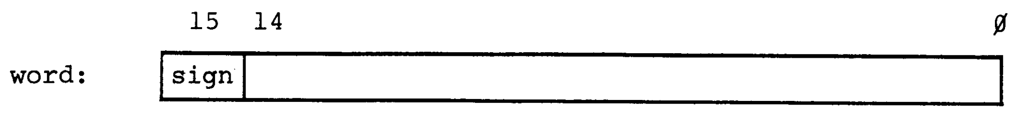
a. Programs depending on 4-word accuracy cannot be run with the 2-word package.

b. .BAC compiled programs can not be interchanged. The program source file must be recompiled.

c. Floating-point virtual core array file formats are not compatible between math packages.

d. Programs using the RECORD I/O functions CVT$F and CVTF$ are not compatible between math packages.

E.2 INTEGER FORMAT



Integers are stored in a two's complement representation. Integer values must be in the range -32768 to +32767. For example:

+22 = ØØØØ26$_8$
-7 = 177771$_8$

As a rule, an integer value is assumed by RSTS only where a constant or variable name is followed by a % character. Otherwise, constants and variables are assumed to be floating-point values.

BIBLIOGRAPHY

1. Programming Time-Shared Computers in BASIC Language

   Eugene H. Barnett
   Wiley-Interscience Books. 1972

2. An Introduction to Computer Programming BASIC Language

   James S. Coan
   Hayden Book Company, Inc. 1970

3. Introduction to Programming: A BASIC Approach

   Van C. Hare, Jr.
   Harcourt Brace Jovanovich, Inc. 1970

4. BASIC Programming, Second Edition

   John G. Kemeny and Thomas E. Kurtz
   John Wiley and Sons, Inc. 1971

5. Introduction to Computing Through the BASIC Language

   R. L. Nolan
   Holt, Rinehart & Winston, Inc. 1974

6. Simplified BASIC Programming

   Gerald A. Silver
   McGraw-Hill Co. 1974

7. Programming in BASIC, with Applications

   Bernard M. Singer
   McGraw-Hill Co. 1973

8. Teach Yourself BASIC, Volume 1 and Volume 2 (self teaching workbook)

   Robert L. Albrecht
   Technica Education Corporation. 1970

9. Fundamentals of Digital Computers (elementary and historical)

   Donald D. Spencer
   Howard W. Sams and Co., Inc. 1969

10. Computer Programming in BASIC

    Joseph P. Pavlovich and Thomas E. Tahan
    Holden-Day Co. 1971

INDEX

BASIC-PLUS
Language Manual
DEC-11-ORBPB-A-D

READER'S COMMENTS

NOTE: This form is for document comments only. Problems with software should be reported on a Software Problem Report (SPR) form

Did you find errors in this manual? If so, specify by page.

Did you find this manual understandable, usable, and well-organized? Please make suggestions for improvement.

Is there sufficient documentation on associated system programs required for use of the software described in this manual? If not, what material is missing and where should it be placed?

Please indicate the type of user/reader that you most nearly represent.

- ☐ Assembly language programmer
- ☐ Higher-level language programmer
- ☐ Occasional programmer (experienced)
- ☐ User with little programming experience
- ☐ Student programmer
- ☐ Non-programmer interested in computer concepts and capabilities

Name ______________________ Date ______________

Organization ______________________________

Street ______________________________

City ______________ State ______________ Zip Code or Country ______________

If you require a written reply, please check here. ☐

Please cut along this line.

-- **Fold Here** --

-- **Do Not Tear - Fold Here and Staple** --

FIRST CLASS
PERMIT NO. 33
MAYNARD, MASS.

BUSINESS REPLY MAIL
NO POSTAGE STAMP NECESSARY IF MAILED IN THE UNITED STATES

Postage will be paid by:

digital

Software Communications
P. O. Box F
Maynard, Massachusetts 01754

digital

digital equipment corporation

Printed in U.S.A.