educomp

"education thru motivation"

EDUCOMP BASIC

USER'S MANUAL

EDUCOMP CORPORATION
298 PARK ROAD
WEST HARTFORD, CONNECTICUT

TABLE OF CONTENTS

CHAPTER 1

AN OVERVIEW

1.1 *EDUCOMP BASIC* IS ITSELF A PROGRAM

The computer language called BASIC is, itself, a computer program. It is a very complex program written in a different kind of computer language ("machine" language), one that requires great detail when writing a program. Use of this great detail permits the construction of the language BASIC, which will let you write programs using a few English words and mathematical symbols. Because BASIC itself is a computer program, it has a limited number of ways of accepting instructions (statements and commands). Thus, the major effort in learning the language must be to master the rules for giving instructions and learning what the instructions do for you. (The error message "ILLEGAL SYNTAX" may be somewhat disconcerting in a computer-oriented endeavor but BASIC is a "language".)

Historically, the original development of BASIC was done at Dartmouth College by Drs. John Kemeny and Thomas Kurtz and it may properly be considered to have a bias toward an educational environment. (Dr. Kemeny was later made President of Dartmouth College.) BASIC is an acronym for Beginners All-purpose Symbolic Instruction Code. The objective in developing BASIC was to provide an easily learned computer language that would run on a computer in an interactive mode. The plan was to bring the computer to the student and provide instant service by means of terminals placed around the campus rather than require the student to go to the computer and face a lengthy wait to get his program run. Although BASIC has been considerably extended, and several versions exist, the few fundamental instructions are common to all versions and are easy to learn.

1.2 BASIC AS A PROGRAMMING LANGUAGE

BASIC is one of the simplest of all programming languages because of its small number of powerful but self-explanatory statements and commands and its easy application in solving problems. Its wide use in scientific, business, and educational installations attests to its value and straightforward application.

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

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

1.2.1 STATEMENTS AND COMMANDS

Instructions in BASIC are one of two types: (1) statements or (2) commands.

Statements are the instructions which make up the user program. They give the details of how the program is to perform the job to be done, i.e., solve your problem. Learning BASIC means learning what each statement does. As you learn how to combine the various kinds of statements in different ways you will find that writing a program becomes progressively easier and, for many people, a fascinating activity much like solving a puzzle. In BASIC, each statement starts with a line number. Each statement also includes a key word in English which indicates what the statement is to do.

Commands, in BASIC, are the instructions that tell BASIC how to act upon, or with, your program. Commands are external to the program which consists of statements. Because commands direct BASIC to work with the program, they are used in writing, editing, running, debugging and finally, storing your program. EDUCOMP BASIC is particularly rich in commands designed to make programming easier. An example of a BASIC command is RUN. RUN causes BASIC to start the sequence of events which, if the program is syntactically correct will result in execution of your program. Note that commands use no line numbers, and are executed immediately after they are typed.

1.2.2 RUNNING BASIC

When the RUN command is given, BASIC attempts to complete a two-phase operation: (1) compilation and (2) execution.

Compilation is the process by which BASIC translates the program coding you have written (in BASIC) into a form "understood" by the computer (machine language). The compiler checks the program to make certain that all the rules of the language are obeyed. When the statements are correct with respect to the syntax of BASIC, the conversion to machine language takes place and the second phase of running a program begins. EDUCOMP BASIC is an incremental compiler which partially compiles the user's program. This partial compilation is to be compared with a pure interpreter which is much slower in running programs.

Execution is the actual operation of the program to perform its assigned task (solve your problem). During execution the computer is performing the operations it was instructed to perform in the manner specified by the program. The computer can do only what it is instructed to do by the program!

1.3 THE ENVIRONMENT

The environment under which EDUCOMP BASIC functions is an extremely complex arrangement of interrelated programs. This particular manual is designed only to convey the constructs of the EDUCOMP BASIC language and it is assumed that the reader is (or will become) familiar with Digital Equipment Corporation's operating system OS/8.

We make the assumption here that the user is always in the language BASIC. Briefly, obtaining EDUBASIC requires (1) getting OS/8 on the air, i.e., starting the system, and (2) typing

```
.R BASIC
```

at the terminal in response to the dot "." printed by OS/8. EDUBASIC responds with

```
READY
```

to indicate that the system is in BASIC and 'READY' to perform for you.

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. Words appearing in capital letters are the key words and indicate what the statement is expected to do with the data found after the key word, if any. For example:

line number READ *list*

b. Square brackets indicate that the bracketed item is optional. For example:

line number [LET] *variable = expression*

c. Items in lower case type *(formula, variable, list,* etc.) are supplied by the user according to rules explained in the text. Items in capital letters (END, IF, READ, etc.) must appear exactly as shown because they form the BASIC language.

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

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

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

b. A <u>statement</u> is a line (or part of a line or multiple lines in some cases) within a user program containing a BASIC language instruction. Each line is terminated by typing the RETURN key.

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

d. <u>User programs</u> consist of a series of statements written by a person using the system in the BASIC language.

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

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

1.5 SPECIAL TERMINAL KEYS

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

The RETURN key causes two operations to be performed:

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

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

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

The CTRL key (or control key) is used in combination with certain letter keys to cause BASIC to perform special operations. These combinations are performed by the user holding down the CTRL key while typing the desired letter key, then releasing both keys. CTRL/U and CTRL/C are examples of these combinations. Some of the CTRL/key combinations are introduced below for use when working through this manual. All usable combinations are described in Chapter 11.

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

b. CTRL/P is used to interrupt the execution of a program and return to the interactive BASIC processor. When typed by the user, CTRL/P causes the system to echo ↑P when BASIC is in command mode and the system prints READY. When used to interrupt the execution of a program, CTRL/P is not printed, but the message STOP IN LINE xxx followed by READY is printed.

c. CTRL/C returns control of the terminal to the OS/8 monitor.

The LINE FEED key reprints the current line, free of rubouts. It does not enter the line into the program. The carriage return is needed to perform that function.

CHAPTER 2

FUNDAMENTALS OF EDUCOMP BASIC

2.1 SAMPLE BASIC PROGRAM

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

The user program (the lines numbered 10 through 999) may at this time mean little, although the remark in the first line (line 5) and the printed results (following the word RUN) clearly show that the program performs payroll calculations.

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 'key' word which specifies the type of operation to be performed.

2.2 LINE NUMBERS

Each line of a user program must be preceded by a line number. Line numbers have the following characteristics:

1. They indicate the order in which statements are normally evaluated;
2. The numbers serve as 'tags' to 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
3. Line numbers enhance program editing by permitting modification of any specified line without affecting any other portion of the program.

*Syntax refers to the rules which specify how the programming language elements are combined.

Line numbers are in the range 1 to 4094. 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 to complete the program.

```
LISTNH
5 REMARK -PAYROLL CHECKSTUBS  SIMULATION W/DATA
10 PRINT "COMPLETE PAYROLL DEMONSTRATION":PRINT:PRINT
15 G=0:READ E
17 IF E=-999 THEN 999
20 READW,H,D,Y,V
22 PRINT "EMPLOYEE NUMBER"; E
24 PRINT "HOURS WORKED=";H
26 PRINT "HOURLY WAGE=";W:PRINT
30 O=0:IF H<=40 THEN 60
35 LET T=H-40
40 LET O=T*(1.5*W)
50 LET G=O
55 LET H=H-T
60 LET R=W*H: LET G=G+R
80 LET Y=Y+G
90 IF Y-G>=9200 THEN 150
100 GO TO 130
110 LET F=(G-(Y-9200))*5.20000E-02:GOTO 160
130 LET F=G*5.20000E-02:GO TO 160
150 LET F=0
160 LET I= (G-(D*13.5))*.14
170 LET N=G-(I+F+V)
180 PRINT "EMP NO","REG WAGE","O T WAGE",,"GROSS"
190 PRINT E,R,O,,G:PRINT
200 PRINT"ITW","FICA","VOL DEDUCT","YTD EARNINGS","NETPAY"
210 PRINT I,F,V,Y,N
220 DATA 15722,3,40,1,5000,12,-999
230 GOTO 15
999 END

READY

RUNNH
COMPLETE PAYROLL DEMONSTRATION


EMPLOYEE NUMBER 15722
HOURS WORKED= 40
HOURLY WAGE= 3

EMP NO         REG WAGE       O T WAGE                      GROSS
 15722          120            0                             120

ITW            FICA           VOL DEDUCT     YTD EARNINGS   NETPAY
 14.91          6.24           12             5120           86.85

READY
```

Figure 2.1

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

2.3 STATEMENTS

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

2.3.1 Multiple Statements on a Single Line

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

```
     1Ø PRINT "EDUCOMP BASIC"
```

is a single statement line, while

```
     2Ø LET X=1:  PRINT X,Y,Z:  IF X=2 THEN 1Ø
```

is a multiple statement line containing three statements: a LET, a PRINT, and an IF-THEN 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 Continuation of a Single Statement onto Another Line

A single statement can be continued on the next line of the program. To type more than 72 characters (not including line numbers) on a line, simply continue typing after 72 characters. EDUBASIC will automatically perform a carriage return, line feed, tab, and allow you to ypte more characters. These two lines will be treated as one for GOTO's, IF-THEN, etc. The length of a multiple line statement is limited to 124 characters.

2.4 CHARACTER SET

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

A through Z

Ø through 9

and the following special symbols:

Symbol	Function
$	Used in specifying string variables.
"	Used to delimit string constants, i.e., text strings.
!	Begins comment part of a line (section 3.9).
:	Separates multiple statements on one line.
#	Denotes a device or filename, or is used as an output format effector.
,	Output format effector and list terminator
;	Output format effector.
()	Used to group arguments in an arithmetic expression.
+ - = < # * / ↑ >	Arithmetic operators.

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

```
1Ø LET C1  =  H * R
```

instead of:

```
1Ø LETC1=H*R
```

Both of the above statements mean the same thing to BASIC and are stored exactly the same within the computer when the program is executed. If a program is too large, spaces may be removed to decrease the size because each space is a character and takes up space in the computer's memory.

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 these, separated by arithmetic or relational operators. Expressions are created by the programmer and inserted into the standard BASIC statements in order to perform the various operations which comprise the user program.

The following are examples of expressions acceptable to EDUCOMP BASIC.

```
     4
     A7*(B↑2+1)
     X<Y
```

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

2.5.1 Numbers

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

```
     2
     -3.675
     1234.56
     -12345.6
     -.000078
```

The following forms are not acceptable numbers in BASIC:

$$\frac{14}{3}$$

$$\sqrt{7}$$

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

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

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

These formats are explained further in later sections.

A number representation using the letter E allows further flexibility. If numbers were limited to six digits, a computer would not be able to solve many problems involving large numbers. Consequently, rather than saying that BASIC can only accept numbers with a maximum of six digits, we say that BASIC has six digits of precision. Larger numbers 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
-123456ØØ.  can be written in BASIC as -1.23456E7
```

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

The combination E7, however, is not a constant, but a variable. The term 1E7 is used to indicate that 1 is multiplied by 10^7.

Numbers are specified according to the following rules:

a. line numbers are unsigned decimal integers in the range 1 to 4094.
b. numbers have the range $1E-616 \leq n \leq 1E616$.

2.5.2 Variables

A variable is a data item whose value can be changed by the programmer. A numeric variable is denoted by a single letter or by a letter followed by a single digit. Thus BASIC interprets E8 as a variable, along with A, X, N5, LØ, and 01. (Subscripted and character string variables are described in later sections.) All variables are set equal to zero (Ø) before program execution. Consequently it is only necessary to assign a value to a variable when an initial value other than zero is required, but it is good programming practice to initialize any variable.

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 formulae:

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

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 (1) having the highest precedence:

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

   ```
   (A+(B*(D↑2)))
   ```

   the innermost parenthetical quantity is calculated first.
2. 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

For example, -3↑2=-(3)↑2=-9

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

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

The formula (or expression) A+B*C↑D is evaluated as follows:

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

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

```
A+(B*C)↑D
```

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

```
(A+B)*C↑D
```

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

2.5.4 Relational Symbols

Relational symbols are used in IF-THEN statements (see section 3.4) where it is necessary to compare values. The relational symbols are as follows:

Mathematical Symbol	BASIC Symbol	Example	Meaning
$=$	=	A=B	A is equal to B
$<$	<	A<B	A is less than B
$\leq$	<=	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
$\neq$	<>or #	A<>B, A#B	A is not equal to B

CHAPTER 3

ELEMENTARY BASIC STATEMENTS

The simplest forms of the more elementary BASIC 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 later chapters.

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 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-X↑3)*(Z-A/B)
```

In line 10, the old value of X is increased by one and becomes the new value of X. In 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).

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

```
     1Ø X=12*(S+7)
```

than

```
     1Ø LET X=12*(S+7)
```

This convenience 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:

```
     1Ø LET X=Y=Z=5.7
```

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

3.2 PROGRAMMED INPUT AND OUTPUT

This section describes the techniques used in performing BASIC program input and output (I/O). The most elementary forms of the PRINT, INPUT, READ, and DATA statements are described 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 viable BASIC program of the simplest sort. If he should want to try his program, these simple I/O statements will provide a means of doing so and obtaining tangible output.

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

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

line number PRINT [*list*]

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

```
    1Ø PRINT
```

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

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

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

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

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

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

```
    STUDENT NUMBER =7744
```

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

line number READ *list*

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

A DATA statement looks as follows:

line number DATA *list*

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

```
10 READ A,B,C
20 READ D,E,F
30 READ A,B,C
40 DATA 1,2,3,4
50 DATA 5,6,7,8,9
```

results in the following assignments being made:

```
A=1
B=2      when line 10 is executed
C=3

D=4
E=5      when line 20 is executed
F=6

A=7
B=8      when line 30 is executed
C=9
```

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

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

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

```
?
```

to be printed. The user could type:

```
?15,24
```

followed by the RETURN key. BASIC would reply:

```
?15,24
?
```

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

3.3 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, equal to, 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
6Ø END
```

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(R2): PRINT A: GOTO 5Ø
```

3.4 CONDITIONAL BRANCH, IF-THEN STATEMENT

The IF-THEN is 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:

line number IF *condition* THEN *statement*
THEN *line number*

The specified condition is tested. If the relationship is found false, then control is transferred to the line 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.

The deciding condition is a simple relational expression in which two mathematical expressions are separated by a relational operator. For example:

Relational Expression

```
A+2>B
```

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

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

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

When a line number follows the word THEN, execution is the same as if a GOTO statement followed the word THEN. 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"
```

The preceding line would 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.

(The above example, line 25, is the same as "IF A>B AND B>C THEN PRINT "A>B>C". This last form is not permitted in EDUCOMP BASIC.)

In the following example, the IF-THEN 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 <u>loop</u> 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 THEN 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 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 (=,<,<=,>,>=, # ,<>)

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

Examples of IF-THEN statements follow

```
    1Ø IF A>B THEN 1ØØ          ! SIMPLE COMPARISON
    2Ø IF A>B THEN A=-B         ! ASSIGNMENT BY A LET STATEMENT
```

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:

   ```
   1Ø IF A=1 THEN PRINT A;:PRINT "TRUE CASE": GOTO 2Ø
   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:

   ```
   2Ø IF A>B THEN IF B>C THEN PRINT "B>C": GOTO 3Ø
   25 PRINT "A<=B"
   ```

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

3.4.1 LOGICAL IF-THEN

It is sometimes useful to have available a somewhat different form of the IF-THEN statement. The following variation is called a logical IF-THEN;

IF *variable* THEN *statement*

If the value of the variable is zero, the statement is false and control is transferred to the next sequential line. If the value of the variable is anything other than zero, the statement is true and the specified expression is executed. For example,

```
110 INPUT A
115 IF A THEN PRINT "A<>Ø": GOTO 110
120 PRINT "A=Ø"
200 END

READY
```

```
RUNNH
? 5
A<>Ø
? -2
A<>Ø
? .246
A<>Ø
? Ø
A=Ø

READY
```

3.5 PROGRAM LOOPS

Loops were first mentioned in the section on the IF-THEN 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 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).

3.5.1 FOR and NEXT Statements

The FOR statement is of the form:

line number FOR *variable = expression* TO *expression* [STEP *expression*]

For example:

```
     1Ø FOR K=2 TO 2Ø 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 3.5.2). 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.

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

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

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

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

```
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.

```
10 FOR I = 2 TO 44 STEP 2
20 LET I = 44
30 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:

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

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

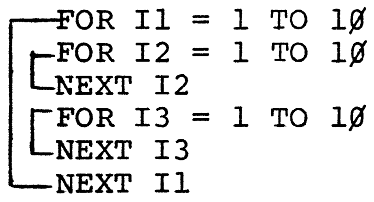
```
10 FOR I=20 TO 2 STEP -2
```

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.

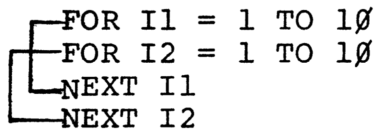
FOR loops can be nested but not overlapped. The depth of nesting depends upon the amount of user storage space available (in other words, upon the size of the user program and the amount of core each user has available). Nesting is a programming technique in which one or more loops are completely within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

Two Level Nesting

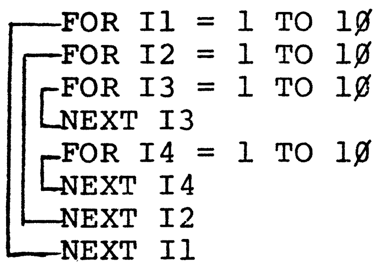


```
FOR I1 = 1 TO 1Ø
FOR I2 = 1 TO 1Ø
NEXT I2
FOR I3 = 1 TO 1Ø
NEXT I3
NEXT I1
```

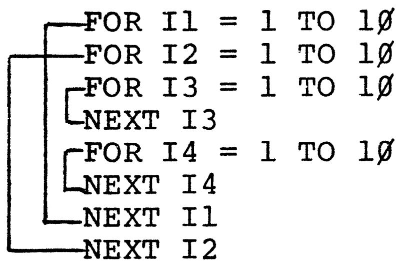


```
FOR I1 = 1 TO 1Ø
FOR I2 = 1 TO 1Ø
NEXT I1
NEXT I2
```

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
```



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

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:

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

causes:

```
    I= 6
```

to be printed when executed.

The FOR nor NEXT statement can be executed conditionally in an IF statement. The following statements are correct:

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

3.5.2 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.

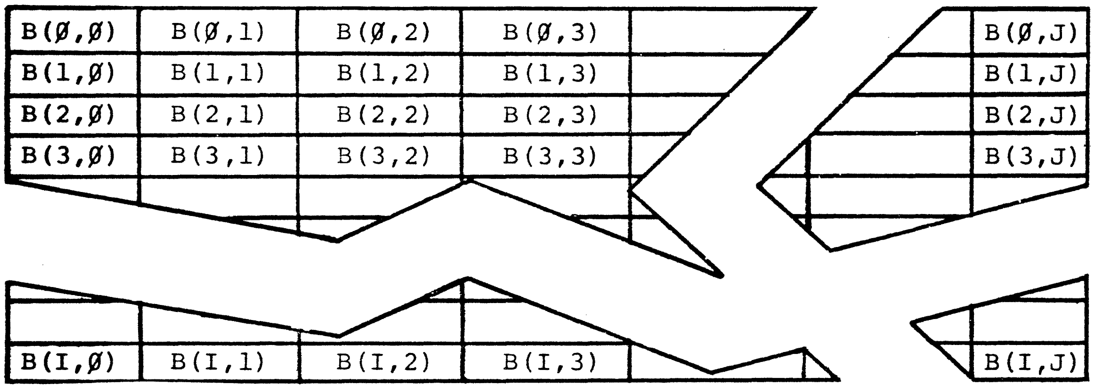
The name of a subscripted variable is any acceptable BASIC variable name followed by one or two integer expressions in parentheses. For example, a list might be described as A(I) where I goes from 1 to 5 as shown below (all matrices are created with a zero element, even though that element is never specified):

A(Ø), A(1), A(2), A(3), A(4), A(5)

This notation 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 possible to use the same variable name as both a subscripted and an unsubscripted variable. Both A and A(I) are valid variables and can be used in the same program. However, BASIC does not accept the same variable name as both a singly and a doubly subscripted variable name in the same program. If A(I) and A(I,J) are used in the same program, an error message 'ILLEGAL SUBSCRIPTING' results.

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 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 14(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
10 REM - MATRIX CHECK PROGRAM
20 DIM A(6,10)
30 FOR I=0 TO 6
40 LET A(I,0) = I
50 FOR J=0 TO 10
60 LET A(0,J) = J
70 PRINT A(I,J);
80 NEXT J: PRINT: NEXT I
90 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.

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.

3.6 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/180)
LOG(144)
```

are substituted.

The various mathematical functions available in EDUCOMP BASIC 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)
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 X
PI	has a constant value of 3.41593.
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.

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

3.6.1 Examples of Particular Intrinsic Functions

Sign Function, SGN(X)

The sign function prints the value

```
 1 if X is positive
-1 if X is negative
 0 if X is zero.
```

For example:

```
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*10↑ D+.5)/10↑D

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

```
LISTNH
10 REM- INT FUNCTION EXAMPLE
20 PRINT "NUMBER TO BE ROUNDED";
30 INPUT A
40 PRINT "NO. OF DECIMAL PLACES";
50 INPUT D
60 LET B=INT(A*10↑D+.5)/10↑D
70 PRINT "A ROUNDED =";B
80 GOTO 20
90 END

READY
```

```
RUNNH
NUMBER TO BE ROUNDED? 55.6534
NO. OF DECIMAL PLACES? 2
A ROUNDED = 55.65
NUMBER TO BE ROUNDED? 78.375
NO. OF DECIMAL PLACES? -2
A ROUNDED = 100
NUMBER TO BE ROUNDED? 67.89
NO. OF DECIMAL PLACES? -1
A ROUNDED = 70
NUMBER TO BE ROUNDED?
STOP AT LINE 30
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), but INT(-14.39) = -15.

NOTE

↑P in the above program terminates program execution.

Randon 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
 .770032        .728066         .438103         .076028         .51324
 .395189        .751974         .955142         .963083         .182217
 .425557        .913388         .650321         .681433         .235705
 .281333        .566656         .867935         .107712         .834855
 3.97218E-02    .724634         .990309         .420146         .608095
 .867253        .730664         .57871          .896285         .169325
READY
```

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

```
     4Ø PRINT INT(1Ø*RND(0)),
```

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

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

READY
```

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

```
     (B-A)*RND(Ø)+A
```

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

3.6.2 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 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 .808118
VALUE 2  IS .842323
VALUE 3  IS .780877
VALUE 4  IS .104348
VALUE 5  IS .598201

READY

RUNNH
VALUE 1  IS .572767
VALUE 2  IS .136269
VALUE 3  IS .662712
VALUE 4  IS .749856
VALUE 5  IS .534725

READY
```

The output from each run is different.

3.7 SUBROUTINES

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
20  INPUT A,B,C
30  GOSUB 100
40  LET A=ABS(INT(A))
50  LET B=ABS(INT(B))
60  LET C=ABS(INT(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 +  0 *X +  1
IMAGINARY SOLUTIONS... X = ( 0 , 1 ) AND ( 0 ,-1 )

STOP AT LINE 90
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.

3.7.1 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.

3.7.2 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.

3.7.3 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.

3.8 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 running a program with line numbers greater than that of the END statement results in the following error message being printed:

```
'END' NOT LAST
```

and execution is halted.

NOTE

A program will execute without an END statement; however, the following error message is printed: NO '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.

Execution of a STOP or END statement causes the message:

```
READY
```

to be printed by the teleprinter. This message signals that the execution of a program has been terminated or completed, and BASIC is able to accept further input.

3.9 <u>REMARKS AND COMMENTS</u>

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 the program.

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

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

The REMARK statement must be preceded by a line number. The word REMARK can be abbreviated to REM for typing convenience, and the message itself can contain any printing character on the keyboard. BASIC completely ignores anything on a line following the letters REM. (The line number of a REM statement can be used in a GOTO or GOSUB statement, see sections 3.4 and 3.8.1, as the destination of a jump in the program execution.) Typical REM statements are shown below:

```
1Ø REM - THIS PROGRAM COMPUTES THE AMOUNTS
11 REM - AND WRITES THE CHECKS
```

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

```
125 LET P1=(H-4Ø)*R          !SET EQUAL TO OVERTIME PAY
130 PRINT P + P1             !PRINT SUM OF OVERTIME AND REGULAR PAY
```

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

Messages in REMARK statements are generally called remarks, those after the exclamation mark, comments. Remarks and comments are printed when the user program is listed but do not affect program execution. It is good programming practice to include REMARKs and comments in all programs, unless space requirements are critical.

3.10 ON-GOTO STATEMENT

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

line number ON *expression* GOTO *list of line numbers*

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

```
5Ø ON X GOTO 1ØØ,2ØØ,3ØØ
```

transfers control to line number 1ØØ if the value of X is 1, to line number 2ØØ if X is 2, and to 3ØØ if X is 3. Any other values of X (other than 1,2, or 3 in this example) would cause a transfer to the next line.

3.11 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 3.7 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, control is transferred to the next line.

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

CHAPTER 4

CHARACTER STRINGS

4.1 CHARACTER STRINGS

The previous chapters describe the manipulation of numerical information; however, EDUCOMP 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 ASCII characters in Table 4-2.

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 PRINT "TYPE A NUMBER BETWEEN 1 AND 12";
12 INPUT N
15 IF N>1 THEN IF N<12 THEN IF N=INT(N) THEN 20
17 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? 2
THE SECOND MONTH IS FEBRUARY

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.

4.1.1 String Constants

Just as numbers can be used as constants or referenced by variable names, EDUCOMP BASIC permits character string constants. Character string constants are delimited by double quotes. For example:

```
1Ø5 LET Y$ = "FILE4"
 8Ø IF A$ = "YES" THEN 25Ø
```

where "FILE4" and "YES" are character string constants.

4.1.2 Character String Variables

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

```
A$, C$, Z$
```

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

```
V$(N), C$(M)
```

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

The same name can be used as a numeric variable, as a string variable and as a one dimensional array in the same program. For example:

```
A     A$     A(N)
```

can all be used in the same program, but

```
      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 "").

4.1.3 Subscripted String Variables

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

```
      1Ø DIM S$(5)
```

indicates the S$ is a string matrix with six elements, S$(Ø) through S$(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 array specifies the number of strings and not the number of characters in any one string. For example, if

```
     1Ø FOR I=1 TO 7
     2Ø LET B$(I)="PDP-8"
     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-8" and the others would be null strings (have no characters). As a general rule, all lists should be dimensioned to the maximum size being referenced in the program.

4.1.4 String Size

A character string can contain almost any number limited usually by the amount of memory storage available. In EDUCOMP BASIC the upper limit on string size is 2050 characters.

The DIM statement is used not only to define an array, but also to indicate the length (number of characters) of a string. In EDUBASIC, strings longer than fifteen (15) characters must be dimensioned before they are accessed. For example:

```
      1ØØ  A$ = "Ø123456789Ø123456789"
      2ØØ  END
```

The above example will generate an error message when executed, STRING OVERFLOW IN LINE 1ØØ. In the above example, line 9Ø should be added,

```
      9Ø  DIM A$ = 2Ø
```

Strings must be dimensioned for the maximum length which they will assume in the user's program. However, a string may contain fewer characters than the number specified in the DIM statement. For example,

```
      1ØØ  DIM A$ = 5Ø
      11Ø  A$ = "EDUCOMP"
      12Ø  END
```

The length of A$ will be seven after this program is executed. If no length is specified for string variables, a length of fifteen is assumed. The following line is an example of DIMensioning for string arrays:

```
      1ØØ  DIM A$ = 3ØØ, B$(1Ø), C$(12) = 24
```

The above statement would reserve space in memory for

1. A character string of length 3ØØ,
2. Eleven strings of length fifteen, and
3. Thirteen strings with twenty-four characters.

4.1.5 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 4-1 contains a list of the relational operators and their string interpretations.

Table 4-1

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.

In any string comparison, trailing blanks are part of the string. That is to say "YES" is not equivalent to "YES ". A null string (of length zero) is considered to be completely blank and is less than any string of length greater than zero.

Table 4-2

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	,		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. The characters CHR$(N) and CHR$(N+128) 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.

4.2 STRING INPUT

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

```
     1Ø READ A$, B, C, D
     2Ø DATA 17, 14, 13.4, CAT
```

causes the following assignments to be made:

```
     A$ = the character string "17"
     B  = 14
     C  = 13.4
```

reading D as CAT causes the message BAD INPUT IN LINE 1Ø to be printed. EDUBASIC then tries to read the next number for D. In this example, no number exists after CAT so another error message is printed OUT OF DATA IN LINE 1Ø.

Quotation marks are necessary around string items in DATA statements only if the string contains a comma or if leading blanks within the string are significant. Quotes 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. EDUBASIC will recognize imbedded and trailing blanks even though there are no quote marks around the string. The comma, carriage return, or second quote is the end of the string.

```
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"
50 END

READY

RUNNH
MR. JONESMISS SMITHMRS. BROWNMISSMR
MR. JONES     MISS SMITH    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.

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 spaces are ignored. If the item did appear in quotes, the string variable is equated to the entire string within the quotes.

A 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$
```

causes the program to accept a line of input from the terminal with 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 4.4 which describes string functions.

An INPUT LINE statement reads the entire line as typed by the user, excluding the line terminating character. The line terminator is a carriage return/line feed, generated by typing the RETURN key.

4.3 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
```

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

4.4 STRING FUNCTIONS

Like the intrinsic mathematical functions (e.g., SIN, LOG), EDUCOMP BASIC 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, and perform other usefull 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 4-3.

Table 4-3

String Functions[1]

Function Code	Meaning

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$). For example:
	`100 PRINT MID(A$,15,5)`
	`110 END`
	`RUNNH`
	`OPQRS`

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

+	Indicates a concatenation operation on two strings. For example "ABC"+"DEF" is equivalent to "ABCDEF". "12"+"34"+"56" is equivalent to "123456".
CHR$(N)	Generates a one-character string having the ASCII value of N (see Table 4-2). For example: CHR$(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.

[1] A$ in the 'MID' and 'LEN' examples is assumed to be "ABCDEFGHIJKLMNOPQRSTUVWXYZ".

CHAPTER 5

DATA STORAGE CAPABILITIES

5.1 FILE STORAGE

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

An EDUBASIC file consists of a sequence of data which is transmitted to (or from) a BASIC program from (or to) an external Input/Output device. The external device can be the user's terminal, the OS/8 system disk, a line printer, magnetic tape, or high-speed paper tape equipment. Each file has both an external name by which it is known within the system and an internal file designator (a number used to refer to the file within the program). An OPEN statement is used to associate an external name with an internal designator.

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

device:filename.extension

where the device can be one of the following:

SYS:	system device
DSK:	default device
DTAØ to DTA7:	DECtape units 0 to 7
PTR:	high-speed paper tape reader
PTP:	high-speed paper tape punch
LPT:	line printer
CDR:	card reader
TTY:	user's terminal
RKAØ	system halt of an RK8e disk
RKA1	other half of an RK8e disk
RKA2n-2: RKA2n-1:	RK8e units for n=2,3,4

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

.BA	BASIC source program, ASCII format
.BC	Compiled BASIC program, 'binary' format
.DA	Data file (sequential)
.BR	BASIC Random access data file (virtual file)

A user can have up to 4 files open (with internal designators 1 through 4) for access at any given time. Each open file consumes a buffer within core storage. The buffer sizes for various devices are all 256 words under OS/8. If a buffer cannot be created for a file, due to a lack of storage space in core, then the file cannot be opened. (The process of opening a file is described in section 5.2).

5.2 OPEN STATEMENT

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

line number OPEN *string* (Name) [FOR INPUT / OUTPUT] AS FILE *expression* ← no. between 1 and 4

The *string* field is a character string constant, variable or expression that contains the external file specification Name of the file to be opened. The AS FILE *expression* must have an integer value between 1 and 4, 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 FILE NOT FOUND error occurs. In the following examples the extensions .DA are assumed unless the extensions are provided.

```
50 OPEN "FILE" 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" 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.BR" AS FILE 4
```

The extension .BR is assumed if not specified.

EDUBASIC permits access to data files by two methods:

a. Formatted ASCII and
b. Virtual core arrays.

5.2.1 Formatted ASCII I/O

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 as follows:

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. (The virtual array dimension statements reference OPEN statements without the FOR INPUT or FOR OUTPUT phrase, as explained later.)

For example,

```
    1Ø OPEN "CDR:" FOR INPUT AS FILE N1
    2Ø INPUT #N1, A$
```

Line number 1Ø above causes the card reader to be opened as an input source with the internal file designator whose value is contained in the variable N1. Line number 2Ø causes input to be accepted from logical I/O channel N1; and the input is associated with the variable A$. (N1 must have a value between 1 and 4.)

5.2.2 File-Structured Vs. Non-File-Structured Devices

OS/8 distinguishes between file-structured (disk, DECtape and magtape) devices and non-file-structured (all other) 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 and a filename. On non-file-structured devices, the device name alone identifies a file (filename and extension, if specified, are ignored.) For example:

DTA1:	is insufficient information to specify a file

DTA1:FRED	is sufficient to specify the file FRED on DECtape unit 1
PTP:	uniquely specifies the high-speed punch
PTP:FILE	produces the error message FILE NOT FOUND IN LINE xxx

File specification syntax is such that the default device need not be specified. For example:

```
DSK:QUIZ
```

is equivalent to:

```
QUIZ
```

When a device is not specified, a file name alone always indicates a disk or DECtape as a default storage device. To store a file on DECtape (other than the default device) the device would be specifically indicated:

```
DTA4:FOO
```

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

```
10 LET I$ = "PTR:"
20 OPEN I$ FOR INPUT AS FILE 1
30 INPUT #1, A$
```

<u>If a file being opened for input does not exist, an error message is returned. If a file being opened for output does not exist, it is created. If a file for output already exists it is deleted and recreated</u>.

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

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

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

Thereafter, reference can be made to file SCRTCH.TM on device DSK: as follows (notice that the internal file designator is represented as a variable, although its value must still be between 1 and 4):

```
     1ØØ PRINT #N1, A$, B$
```

5.2.3 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 4. 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).

```
     1Ø INPUT #Ø, A$
```

is equivalent to:

```
     1Ø INPUT A$
```

It is sometimes useful to be able to request keyboard input without having the "?" prompting character printed first. This can be accomplished by opening the user's terminal ("TTY:") on some internal file designator other than Ø. The ? character is only generated for input requests on file #Ø, as shown in the following example:

```
   5 DIM A$=5Ø
   1Ø OPEN "TTY:" FOR INPUT AS FILE 1
   2Ø PRINT "WITH USE OF INTERNAL FILE DESIGNATOR"
   3Ø PRINT "TYPE YOUR NAME, FOLLOWED BY A RETURN KEY, AND A CTRL/Z"
   4Ø INPUT #1,A$
   5Ø PRINT: PRINT
   6Ø PRINT "FOR COMPARISON, WITHOUT FILE DESIGNATOR"
   7Ø PRINT "TYPE YOUR NAME FOLLOWED BY A RETURN KEY"
   8Ø INPUT A$
   9Ø END
```

```
READY

RUNNH
WITH USE OF INTERNAL FILE DESIGNATOR
TYPE YOUR NAME, FOLLOWED BY A RETURN KEY, AND A CTRL/Z
J. P. JONES


FOR COMPARISON, WITHOUT FILE DESIGNATOR
TYPE YOUR NAME FOLLOWED BY A RETURN KEY
? J. P. JONES

READY
```

If a file is being opened for both input and output or to be referenced as virtual arrays the form:

Line number OPEN *string* AS FILE *expression*

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

When a program used a statement such as:

```
50 OPEN "FOO" AS FILE 4
```

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

5.3 OUTPUT TO VARIOUS DEVICES

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

line number PRINT # *expression* , *list*

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

```
10 OPEN "DATA1" FOR OUTPUT AS FILE 1
20 PRINT #1, "START OF DATA FILE"
```

The above lines open a file called DATA1 on the system device with internal file designator #1 (of 4 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:

```
10 LET I$="LPT:"
20 OPEN I$ FOR OUTPUT AS FILE 1
30 FOR I = 1 TO 5: PRINT #1, I, SQR(I): NEXT I
40 END

READY
```

The results would appear on the line printer as follows:

```
1              1
2              1.41421
3              1.73205
4              2
5              2.23607
```

It is advisable to print only one character string per PRINT statement because terminators are not automatically introduced.

The carriage return serves as the delimiter. A MID function may be used to separate the fields as desired.

5.4 INPUT FROM VARIOUS DEVICES

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

line number INPUT # *expression , list*

causes input to be accepted from the previously opened file or device indicated in the expression (see section 5.1). As long as the value of the expression is non-zero, the specified file is read through one of the available user I/O buffers (internal file designators). If the expression is zero, or missing completely, input is from the user terminal. No ? character is printed on the terminal paper when input is requested from a device other than the terminal, opened on file #Ø. For example:

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

causes the string A$ to be read from the high-speed paper tape reader.

Note that spaces are ignored in numeric input data. Commas are inserted automatically when printing out to a data file. When inputting from a data file, a comma or carriage return is taken as a terminator.

Once a file is opened it can be closed (a CLOSE statement must be used) with a second OPEN statement. Closing and reopening the file moves the positioned pointer within the file back to the beginning of the file, so that the entire file becomes available again for sequential referencing. These operations serve much the same function as a RESTORE statement would to the pool of DATA statement.

5.5 VIRTUAL DATA STORAGE

Many applications require a capability to individually address and update records on a disk file in a random (non-sequential) manner. Other applications may require more core memory for data storage than is economically feasible. EDUBASIC fills both these requirements with its easy-to-use random access file system, called virtual core.

The EDUCOMP BASIC virtual core system provides a mechanism for the programmer to specify that a particular data array is not to be stored in the computer's core memory, but within the OS/8 file system, instead. Data stored in files external to the user program will survive, even after the user leaves his terminal, and can be retrieved by name at a later session. Items within the file are individually addressable, as are items within core arrays. In fact, it is the similar way in which data are treated in both core and random-access files which leads to the name virtual core.

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

5.5.1 Virtual Core DIM Statement

In order for an array of data to exist in virtual core, it must be declared in a special form of the DIM statement (places in program sequence somewhere <u>after</u> the corresponding OPEN statement). This special statement is as follows:

line number DIM # *expression* , *list*

where the expression is an integer constant between 1 and 4 and corresponds to the internal file designator on which the program has opened an internal file. The variable list appears as it would for a normal core resident array DIM statement. Thus, a 100 by 100 matrix could be defined as:

```
     1Ø DIM #2, A(1ØØ ,1ØØ)
```

Numbers and strings can both reside in virtual core arrays. More than one array can be specified in one virtual core file. For example:

```
     25 DIM #1, A(1ØØØ) , C$(25ØØ)
```

which allocates space for 1000 numbers and 2500 character strings (15 characters long each).

5.5.2 Virtual Core String Storage

One of the few differences in data handling between core and virtual arrays occurs in the storage of strings within string matrices in virtual core. Strings in virtual core are of fixed length (all elements having a particular name are of the same length.) This length can be defined by the programmer and varies from 1 character to 2000 characters. The system forces lengths to be a multiple of 3:

3, 6, 9, 12, 15, 18, 21, 24, 27, 30, 33,

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

```
     1Ø DIM #1, X$(1Ø)= 65
```

is the same as:

```
     1Ø DIM #1, X$(1Ø)= 66
```

If no length is specified, a default length of 15 characters is assumed. The length attribute of virtual core strings (as well as ordinary strings) is specified in the DIM statement, using the notation:

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

where

A$ consists of 101 strings of 33 characters each;
B$ consists of 101 strings of 6 characters each;
C$ consists of 101 strings of 15 characters each;

5.5.3 Opening a Virtual Core File

In order for the user to reference his virtual core file, he must first associate one of his files (known by name) with an internal file designator from 1 to 4 (which is then used in the virtual DIM declaration). This is normally done with the following OPEN statement:

line number OPEN *string* AS FILE *expression*

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

```
     35 OPEN "PAY" AS FILE 1
```

associates the file named "PAY" with internal file 1. If "PAY" already exists, then the existing file is used; if there is no file named "PAY" one would be created. The extension .BR is assumed.

Sophisticated users are urged to read Chapter 8 which describes the system implementation of the virtual core processor. A mastering of this information will produce programs which utilize the system resources in a highly efficient manner.

As an example of virtual core usage, consider the problems of implementing an information retrieval system for a small organization. There might be 1000 employees, each needing a 300-character record containing the name, home address, phone, work station, and phone extension of the employee. Rather than order the records in the file, it is decided to maintain a separate index file containing only badge numbers. The order of employee records in the master file is the same as the badge number sequence in the index file. Thus, to extract information on an employee with badge n, we find his badge number in the index file and use the index found to retrieve his data from the master file. Since the number of employees is small, numeric data can be used in the badge file; only alphanumeric data is stored in the master file.

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

```
LISTNH
5 DIM R$=300
10 !PROGRAM TO LOOK UP NAMES IN MASTER FILE
20 OPEN "BADGE" AS FILE 1                   !BADGE FILE
30 OPEN "MASTER" AS FILE 2                  !MASTER FILE
40 DIM #1, B(1000)                          !1000 BADGE NUMBERS
50 DIM #2, A$(1000)=300                     !1000 RECORDS, EACH 300
55                                          !CHARACTERS LONG
60 PRINT "INPUT BADGE NUMBER";:INPUT E      !GET EMPLOYEE NUMBER FORM TTY
70 FOR I=1 TO 1000: IF B(I)=E THEN 100      !IS BADGE # IN FILE?
75 NEXT I
80 PRINT "NO SUCH EMPLOYEE": GOTO 60        !NO
100 !WE NOW HAVE INDEX INTO FILE,I          !YES
110 R$=A$(I)                                !BRING RECORD INTO CORE
120 PRINT "NAME IS";MID(R$,10,15)           !NAME IS FROM COLUMN 10 TO 25
130 GOTO 60                                 !NEXT...
200 END

READY
```

5.6 CLOSE STATEMENT

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

line number CLOSE *expression*

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

```
     255 CLOSE 2, 4
     345 CLOSE 3
```

Line 255 above closes the files opened on internal device designators 2 and 4. Line 345 closes the file open on internal device designator 3.

5.7 KILL STATEMENT

The KILL statement is of the form:

line number KILL *string*

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

```
     455 KILL "XYZ.DA"
```

When using the KILL statement, extensions must be used. Otherwise no error statement is given but the file is not deleted.

5.8 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:

```
    1ØØØ CHAIN "MAIN", 2ØØ
```

causes the program MAIN to be loaded and started at line 2ØØ.

Chaining to precompiled program files (.BC files) is considerably more efficient than chaining to BASIC source program files since .BA files require compilation upon each call.

Communication between chained programs is performed by means of the user's file area.

If no extension is given, EDUBASIC looks for a .BC file.
If no .BC file is found, EDUBASIC looks for a .BA file.
If no .BA file is found, an error message results.

When the CHAIN statement is executed, all open files for the current program are kept open, the new program segment is loaded, and execution continues. Virtual files should be closed and reopened across a chain.

The significance of not having to close and reopen a sequential data file is that the file pointer will not be reset (see section 5.4). In other words, a PRINT statement in the chained-to program will add information to the end of the file. This significance is not present when working with virtual files.

CHAPTER 6

EDUBASIC GENERALIZED INPUT AND OUTPUT OPERATIONS

6.1 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 numeric, subscripted, or character string variables. The list of values must correspond in type with the variables to which the value will be assigned, (although they are stored according to the type of the variable.)

The data pool consists of all DATA statements in a program. Values are read starting with the DATA statement having the lowest line number and continuing to the next 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.

Quotes are necessary in DATA statements only around string items which contain a comma or where leading blanks within the string are significant. 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 did not appear in quotes; leading spaces are ignored. If the item did appear in quotes, the string variable is equated to the entire string within the quotes.

6.2 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.

6.3 INPUT STATEMENT

The INPUT statement allows data to be entered to a running program from an external device, the user's keyboard, disk, DECtape, paper tape reader, etc. The full form for this statement is:

line number INPUT[# *expression* ,] *variable list*

In many cases the simpler form:

line number INPUT *variable list*

is used. This last form causes a ? to be printed at the terminal and the system then waits for the user to respond with the appropriate values. If sufficient values are not typed, the system prints another ?; if too many values are typed, excess values are ignored.

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. If the value of the expression is non-zero and the specified file is open to the user terminal as an input device, then no ? character is printed at the terminal when input is requested. For example:

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

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

Another format of the INPUT statement allows for the entering of an entire line of data as a single character string entity, regardless of punctuation. This statement is different from the normal mode of string input, where the comma and double quote characters have special significance. The format is:

line number INPUT LINE[# *expression* ,] *string variable*

For example, the statement

```
     25 INPUT LINE A$
```

would print a question mark and wait for the user to enter a line followed by the RETURN. As another example:

```
     2Ø OPEN "F2" FOR INPUT AS FILE 4
     25 INPUT LINE #4, B$
```

These lines cause the system to open a file F2 on the system disk on channel #4 (of 4 possible channels) to input a line of characters up to the next RETURN character.

6.4 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.

Elements in the list are separated by commas or semicolons. For example:

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

when executed 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

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

the comma character which causes the print head to move to the next available print zone (from 1 to 14 spaces away). If the fifth print zone on a line is filled, the print head moves to the first print zone on the next line.

The semicolon character functions as follows:

a. if a numeric variable 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 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.

b. at most six significant digits are printed.

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.

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

line number PRINT # *expression* , *list*

The expression is the number of a previously opened output file. For example:

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

causes four values to be punched onto paper tape by the high speed punch which is opened for output as file 3, of 4 possible files. As many as four possible virtual files may be open at once (for input or output).

6.4.1 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 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; and the list is a list of items to be printed. 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 identifies a one character string field. The string is specified in the list within the PRINT statement. For example:

```
     1Ø 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 by spaces enclosed between backslashes. The backslash character (\) is produced by typing SHIFT/L on the Teletype keyboard. Enclosing no spaces indicates a field two columns wide, etc. For example:

```
     2Ø 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. Any decimal point arrangement can be specified and rounding is performed as necessary (not truncation). For example:

```
     3Ø PRINT USING "###.##", 12.346
```

causes

```
     12.35
```

to be printed on the user's terminal, while

```
     4Ø PRINT USING "####", 12.345
     5Ø PRINT USING "####.", 12.345
     6Ø PRINT USING "##", 1ØØ
```

causes

```
     12
     12.
    *
```

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, an asterisk is printed for each alloted space.

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.

Exponential Format

When the exponential form of a number is desired, the numeric 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$="###↑↑↑↑ ######"
     1Ø A=1ØØØØ.
     2Ø PRINT USING F$,A,A
```

causes

```
     1ØE + Ø3  1ØØØØ
```

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.

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 does inhibit termination of the printed line.

```
     1Ø PRINT USING "##     ##  ##", 1;2,3
```

prints the following:

```
     1      2   3
```

As another example:

```
     1Ø PRINT USING "##.##", 2;5;
     2Ø PRINT "X"
```

prints

```
     2.5ØX
```

As another example:

```
     1Ø LET A=1.32111: B=2.45457
     15 LET F$ = "A=##.##B=##.##"
     2Ø OPEN "LPT:" FOR OUTPUT AS FILE 4
     25 PRINT #4, USING F$, A,B
```

would cause:

```
     A= ;/32B= 2.45
```

to be printed on the line printer.

6.4.2 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 Teletype 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:

```
     1Ø PRINT "X";TAB(1Ø);POS(Ø)
```

causes the following to be printed:

$$\text{Position } \overset{X}{1}\underbrace{\qquad\qquad\qquad}_{9\text{ spaces}}\overset{1Ø}{\text{position } 1Ø}$$

CHAPTER 7

EDUBASIC COMMANDS

7.1 INTRODUCTION

We have discussed the statements in EDUCOMP BASIC which are available to the programmer to solve the problem. However, equally important are the commands or immediately executed key words in BASIC which permit you to perform the tasks of creating your program, debugging it, running the program, and finally, saving the statements. All of these steps are greatly eased with the rich vocabulary of commands in EDUCOMP BASIC.

The user is assumed to be familiar with OS/8 and how to start up an OS/8 system. In response to the dot (.) given by the OS/8 command decoder, type

```
.R BASIC
```

EDUCOMP BASIC responds with

```
READY
```

7.2 CREATING A PROGRAM

In order to create a new user program, at any time a user can issue the NEW command as follows:

```
NEW
```

followed by the RETURN key. The system responds by printing:

```
NEW FILE NAME--
```

to which the user responds by typing the name of the new program (no more than six characters). When typing a new BASIC program, the file name extension .BA (for BASIC) is added to the name by the system.

Alternatively, the user can give the command NEW followed by the program name, to avoid having the system prompt the typing of the program name:

```
        NEW CALPPB
```

is equivalent to

```
        NEW
        NEW FILE NAME--CALPPB
```

When the NEW command is given, it:

a. Deletes any program currently in core, and

b. Causes BASIC to remember the new program name.

```
    NEW DTA1:CALPPB
```

is meaningless. All checking for duplicate files occurs when the SAVE command is given.

Following the creation of a new file with an acceptable file name, the user can begin to type his program, beginning each line with a line number.

If the user doesn't type NEW either he will get the program name given to the previous program or BASIC will create a file called NONE (if no previous name has been given) which can be referenced later as NONE. At any time, this name can be changed (see section 7.5). Only one file with the name NONE can exist at any one time.

7.3 CALLING AN EXISTING PROGRAM

When the user desires to recall the source file of an old BASIC program (previously saved on a storage device), he gives the OLD command as follows:

```
          OLD
```

to which the system replies:

```
          OLD FILE NAME--
```

The user then types the name of the old BASIC file containing the program. Alternatively, the user can indicate the old file name without prompting, as follows:

```
          OLD TAXES
```

which calls the old file TAXES from the disk. If the file is not available on the disk or if it is protected against that user, an appropriate message is printed.

There is a more general form for the OLD command which allows the user to specify the particular OS/8 device on which the OLD program exists.

OLD *device:file name.extension*

If the program ALUM is to be called from DECtape number 1, the command string is

```
          OLD  DTA1:ALUM
```

where the extension .BA is assumed.

OLD may also be used to read in a program (or data file) from a non-file structioned device (TTY:, PTR:, CDR:, etc.). In this case, only the device is specified since these devices have no directory and do not store more than one file at a time. As an example,

```
    OLD CDR:
```

reads a program from the card reader.

NOTE: When accepting input from non-file structional devices, CTRL/Z is used as an end-of-file character. This character may be typed at the console or may occur at the end of the file on the particular device used. OS/8 automatically inserts a CTRL/Z at the end of a paper tape reader file.

7.3.1 CALLING DATA FILES

A further generalization of the OLD command occurs in EDUBASIC for use with data files. Certainly a data file may be called into memory with the previously described versions of the OLD command.

However, many times it is very convenient to be able to append line numbers to the elements in a data file to ease editing the information. The full form of the OLD command is

OLD *device:file name.extension line number, increment*

As an example,

```
        OLD  RKA2:STUDNO.DA  100, 5
```

brings in the data file STUDNO from the second RK8e disk and numbers each element starting with line number 100 in increments of 5. New elements may be added, deleted, or modified easily and the file may be stored again without the line numbers by using the NSAVE command (section 7.5).

7.3.2 OVERLAYING A PROGRAM

Sometimes it is necessary to append a subroutine or series of statements to an already existing program. The OVERLAY command works exactly like OLD except that the program already in memory is not destroyed.

```
    OVERLAY  device;file name.extension  line number, increment
```

As an example, file BX on SYS contains

```
          4Ø  PRINT "TELL ME AGAIN"
          5Ø  GO TO 1Ø
```

The program (LOVE) in memory is

```
          1Ø  PRINT "IF YOU LOVE ME, TYPE A 7"
          2Ø  INPUT A
          3Ø  IF A #7 THEN PRINT "I DON'T LOVE YOU EITHER": GO TO 6Ø
          6Ø  END
```

If the command is now given,

```
          OVERLAY BX
```

BX now contains lines 1Ø through 6Ø.

```
     LISTNH
          1Ø  PRINT "IF YOU LOVE ME, TYPE A 7"
          2Ø  INPUT A
          3Ø  IF A #7 THEN PRINT "I DON'T LOVE YOU EITHER": GO TO 6Ø
          4Ø  PRINT "TELL ME AGAIN"
          5Ø  GO TO 1Ø
          6Ø  END
```

The OVERLAY command is very useful for adding a subroutine to a program.

Both the OLD and OVERLAY commands may be used only to call

ASCII files into memory (e.g., not compiled or .BC files). Any file called with OLD or OVERLAY may be edited by the user at the terminal.

7.4 EDITING PROGRAMS

During the course of typing a program at the terminal or after a program is seen to be incorrect, changes can be made in the text of a program. These changes are made in what is called the editing phase of BASIC, between the time when the system prints READY and the time when the user types RUN. (During this time, commands can be executed.)

The simplest type of correction is done during the typing of a line before the line is entered to the system with the RETURN key. For example:

```
10  PRHNT
```

If the user realizes he has typed PRH instead of PRI, he can type the RUBOUT key once for each character to be erased. The RUBOUT key causes the erased character to be echoed on the user terminal between back slashes as they are erased. For example:

```
ABC<RUBOUT><RUBOUT>DEF
```

Typing the above is printed on the terminal as follows:

```
ABC\CB\DEF
```

If the RETURN key is typed at the end of the above line, the system would receive it as follows:

```
ADEF
```

The letters B and C have been erased.

If the user decides that his easiest course is to delete the entire line, and he has not yet typed the RETURN key, then he can type CTRL/U (hold down CTRL and U keys), which performs this function. If the RETURN key has been typed, then the line may merely be retyped; the second version will replace the first in the computer memory.

7.4.1 THE EDIT COMMAND

One of the most useful commands in EDUBASIC is the EDIT command. This search command permits the user to modify a completed line or statement which is already contained within the memory of the computer. Thus, EDIT should be contrasted with the use of the RUBOUT key where the latter is used for changing a line already completed, i.e., RETURN has not been typed.

EDIT tells the computer to find a given line number, and to then search for a particular in that line. As an example,

```
READY
EDIT 12Ø
(character)
```

When you type the *character* to be searched for, this *character* is not printed, but the line requested is immediately printed out to the *character* which you have typed. If there are several occurrences of this *character*, the first one is printed and printing ceases. At this point you have several options:

a. Type a RUBOUT to delete the last character printed; type two RUBOUTs to delete the last two characters printed; and so on.

b. Type in new characters to take the place of any you rubbed out; or, of you have not typed any RUBOUTs, to add to the text already there.

c. Type CTRL/L; the computer will now search for the <u>next</u> occurrence of the same search character.

7.4.3 DELETE COMMAND

The DELETE command is used to remove one or more lines from the user program currently in core. For example:

```
          DELETE 1ØØ
```

causes line number 100 to be deleted. (The user should first be certain that no other line references line number 100 unless that line is to be replaced.)

```
          DELETE 1ØØ-2ØØ
```

causes all the program lines between and including line numbers 100 and 200 to be deleted. If 100 and/or 200 do not exist in the program, any lines within the range from 100 to 200 are deleted.

If several groups of lines are to be deleted, then the user can type:

```
          DELETE 1ØØ-2ØØ, 3ØØ-4ØØ, 1ØØØ-11ØØ, 162Ø
```

which deletes all lines between 100 and 200, 300 and 400, 1000 and 1100, and line number 1620.

Individual lines may be deleted with the following form:

```
          DELETE 1Ø, 33, 976
```

This command deletes only lines 10, 33 and 976.

If only one line is to be deleted it may be more convenient merely to type the line number and the RETURN KEY:

```
          1Ø
```

which is equivalent to:

```
          DELETE 1Ø
```

d. Type a BELL code (CTRL/G); now type a new search character (which is not printed). The computer will now print out the line until it meets this new character.

e. Type the ALT MODE key; the left half of the line, up to and including the last character printed, is erased. The line number, however, is not erased.

f. Type the RETURN key. All the line to the right of the last character printed is dropped. The left side of the line is saved and the RETURN indicates that the EDIT command is complete.

g. Type the LINE FEED key. The whole line, in its present condition (including any changes you have made) will be printed but not saved. To save the line you must type RETURN. LINE FEED may be typed as many times as you like.

Note that EDIT cannot be used to change a line number. The only way to move a line to a new position in the program is to retype it, complete with its new line number. The old line should then be deleted. The RESEQUENCE command (next section) is useful for creating the space to add new lines.

7.4.2 THE RESEQUENCE COMMAND

The RESEQUENCE command simply renumbers the line numbers in the user program. The general form of this command is

RESEQUENCE *line number, increment*

If only the word RESEQUENCE is typed and no *line number* and *increment* are specified, the program is renumbered starting with *line number* 100 in *increment*s of 10.

Note that only the program (or data file) currently in memory is resequenced and that if you wish to SAVE the new version or REPLACE the old version, these commands must be given (see section 7.5).

LIST Command	Meaning
LIST	List the entire user program as it currently exists.
NLIST	Same as LIST, but without line numbers.
LISTNH	Same as LIST, but without a program header.
NLISTNH	Same as NLIST, but without a program header.
LIST n	List line n, without a program header.
LIST m,n,p	List lines m,n,p without a program header.
NLIST m,n,p	List lines m,n,p without line numbers.
LIST nl-n2	List lines nl through n2, inclusive, without a program header.
LIST LPT:	Lists the user program on the line printer (if one exists on the system).

7.4.5 SEARCH

One of the most powerful editing features in EDUBASIC is the SEARCH command. The first form is

SEARCH *nl-n2/string A/*

This SEARCH command lists all lines in the range *nl* to *n2* inclusive that contain *string A* anywhere in the line. If no line numbers are specified, the entire text buffer is searched. Note that *string A* may be a variable name (A$) or a group of characters (ABCD) <u>without</u> quotation marks unless the quote marks are part of the string.

The second form of the SEARCH command is

SEARCH *nl-n2 /string A/string B/list*

This form of the SEARCH command replaces all occurrences of *string A* with *string B* in the range *nl-n2*. If the optional word *list* is specified at the end of the command, all line numbers in which replacement was performed are listed. If no line numbers are specified, the entire text buffer is searched.

7.4.4 LIST COMMAND

The LIST command is used to obtain a clean printed copy of all or part of the user's current program. This listing is especially useful during and after an editing session in which the original program is changed.

In order to obtain a printed copy of the entire program as it currently exists within the system, type:

```
          LIST
```

In order to list a single line, type:

```
          LIST 1ØØ
```

to type line 100. (LIST 100, 300 lists both lines 100 and 300.)

In order to list a section of the program, type:

```
          LIST 1ØØ-2ØØ
```

which will cause the listing of the entire program from line number 100 to line number 200 inclusive.

The above LIST commands list both statements and line numbers. If the user wishes a listing without line numbers, the command NLIST is available. NLIST may be used similarly to the three cases above, but the lines listed will have no line numbers.

In the first of the above cases, BASIC prints a program header containing the program title and data. If this header is not desired (as it might not be for normal editing), the command may be given as LISTNH to delete the header material. To summarize:

EXAMPLE: File in memory contains:

```
          1Ø   PRINT "SEARCH COMMAND USAGE"
          2Ø   INPUT B
          3Ø   IF B=5 THEN 2Ø
          4Ø   B=B+1
          5Ø   PRINT B
          6Ø   GO TO 2Ø
          7Ø   END
```

```
SEARCH 30-60/20/
```

```
          3Ø   IF B= 5 THEN 2Ø
          6Ø   GO TO 2Ø
```

```
SEARCH  /B/C/LIST
```

```
          2Ø   INPUT C
          3Ø   IF C=5 THEN 2Ø
          4Ø   C=C+1
          5Ø   PRINT C
```

```
SEARCH  PRINT/PRINT B;/
```

The file in memory now contains:

```
          1Ø   PRINT B; "SEARCH COMMAND USAGE"
          2Ø   INPUT C
          3Ø   IF C=5 THEN 20
          4Ø   C=C+1
          5Ø   PRINT C
          6Ø   GO TO 2Ø
          7Ø   END
```

(In order to permit the slash (/) to be part of the string, an alternate form of the SEARCH command allows replacement of the slash by any non-numeric character -- e.g., SEARCH A/A*A replaces all slashes with asterisks.)

7.5 MANIPULATING USER PROGRAMS

The commands in this section enable the user to compile, save, run, and rename his files. These are all operations performed on a program as a whole (either in core or as a file) and are used once a complete program has been prepared at the terminal.

7.5.1 RUN Command

The RUN command is used to cause the execution of any source

BASIC program. (Source programs are stored as the user typed them; compiled programs are files described in section 7.5.2.)

In order to run the program currently in core, the user simply types:

```
        RUN
```

This command causes the execution of the program in core. A program header is printed after the RUN command is given, consisting of the program name, date and language. If this information is not desired, the command

```
        RUNNH
```

should be given. RUNNH executes the current program without printing the header material.

7.5.2 EXECUTE Command

When it is desired to run a program not in memory, the EXECUTE command is used.

EXECUTE *device:file name.extension line number*

This command causes BASIC to search for *file name* on the *device,* load it, compile it (if necessary), and run it if it is found. If no extension is specified and both the .BA (source) and .BC (compiled) versions exist, BASIC will execute the compiled form because it requires less time. In order to retrieve and execute the source, it is necessary to specify the extension .BA after the *file name*. An alternate approach is to give the OLD command followed by the RUN command. This approach is not equivalent to the EXECUTE command because EXECUTE will save the file currently in core (before EXECUTE is typed), execute the program called for, and then restore the previous file into memory.

Compiled (.BC) files can only be executed with the EXECUTE Command.

If only the source version of a file exists on a device, the EXECUTE command serves as a combination of the OLD and RUN commands, except with the restoring of the previous file noted above. For example, if the program STOCK is stored on DECtape 1, it may be called into memory and executed with the following single command string:

```
          EXECUTE  DTA1:STOCK, 1ØØ
```

where execution starts at line number 100. (Perhaps lines 1 through 99 contained instructions not required for the running of the program.) As another example,

```
          EXECUTE  CDR:
```

reads a BASIC program from the card reader and runs it.

7.5.3 SAVE Command

The SAVE command is used to store BASIC source programs on the disk as follows:

```
          SAVE
```

The program currently in core is saved under its file name with the extension .BA. If a file of the same name exists, then SAVE returns the error message:

```
          DUPLICATE FILE NAME
```

Where the current name of the file is not the desired name, the format:

```
          SAVE GRADE
```

can be used, which saves the program currently in memory under the name GRADE.BA.

In cases where the desired storage device is not the default device, the format:

```
          SAVE  device:file name.extension n1,n2-n3,n4
```

is used where *device* indicates the device designation. The file is stored as FILE NAME.BA. For example:

```
          SAVE  DTA4:ACCPAY
```

saves the whole file ACCPAY.BA on DECtape 4. The numbers (*n1, n2-n3,n4*) are used if only part of the file in memory currently is to be saved. As an example:

```
          SAVE DTA4:ACCPAY  1Ø, 1ØØ-36Ø
```

saves only lines 10 and 100 through 360 of the file ACCPAY on DTA4.

The SAVE command is used only with source files and cannot be used with compiled files. When a program is saved, under some name, the program is still in core to be used or ignored as the user wishes.

To obtain a listing of his program on the line printer, the user can type:

```
          SAVE  LPT:
```

To punch a tape of his program, the user can type:

```
          SAVE  PTP:
```

7.5.4 SAVE Without Line Numbers

The NSAVE command saves the file currently in memory but without line numbers.

```
          NSAVE  device:file name  n1,n2-n3,n4
```

This particular command is very useful during the editing of a data file. The file may be called into memory with the OLD command and at the same time line numbers may be appended.

```
          OLD  DTA1:PARTFL  1ØØ,1Ø
```

After editing has occurred (adding, deleting, or changing the items in the file), the NSAVE command is used to save the file without line numbers.

```
          NSAVE  DTA2:PART2  1ØØ-95Ø
```

The above command string saves only lines 100 through 950 of the new data file (PART 2) on DTA 2.

7.5.5 UNSAVE Command

The UNSAVE command is used to remove a file from a storage device. The form:

```
          UNSAVE  device:file name.extension
```

removes the *file name* from the *device*.

Any number of files may be removed. Each name must be separated from the following name by commas. As an example

```
          UNSAVE  PART1, PART2, PART3
```

If no *extension* is given .BA is assumed. If no *file name* is given, BASIC responds with FILE NAME -- and waits for the user to input a file name.

7.5.6 RENAME Command

The RENAME command causes the name of the program currently in core to be changed to the specified name. For example:

```
          RENAME  COLGNO
```

The old name of the program in core is discarded and it is now known as COLGNO. If the SAVE command is given:

```
          SAVE
```

the file COLGNO.BA would be stored on the systems device.

7.5.7 REPLACE and NREPLACE

The REPLACE command is used when the program in memory has the same name as a file on the same device and the user wishes the program in memory to become the new file with that name. The command is simply of the form:

REPLACE *device:file name.extension n1,n2-n3*

where *n1,n2-n3* indicate that only these lines may be saved.

REPLACE is like SAVE, but destroys without notice the old copy of the same file, if it exists.

NREPLACE is the same as REPLACE except that the file is saved without line numbers.

7.5.8 COMPILE Command

Normally BASIC reads each line of a user's program as it is typed and, if acceptable, translates the line into a form more easily understood by the computer. When lines within the user's program are altered, all lines which are in the program need to be recompiled (i.e., translated). When the SAVE command is given, only the source version of the program (i.e., the text that is typed in response to the LIST command) is retained in the specified place. In response to the OLD command, BASIC reads the text from a file and compiles it in much the same manner as is done when the program is read from the user's keyboard.

Once a program is completely developed and debugged, it may be desirable to avoid the time-consuming practice of compiling the program every time it is fetched from the library. For this reason, the COMPILE command has been provided. This command permits the user to save an image of his compiled program, rather than (or in addition to) the source text of the program. This compiled program may be called and executed with a minimum of overhead by use of the EXECUTE command (see section 7.5.2).

Due to the transformation which takes place when a program is compiled, a file with the extension .BC can only be executed, it cannot be edited. Therefore, the user can issue the EXECUTE command with respect to these compiled files, but the file cannot be brought into core with the OLD command.

If the current file name (i.e., that which is typed in the heading of a listing) is INVCTL, then the command

```
          COMPILE
```

will save the compiled program in a file named INVCTL.BC. If another name is desired for the compiled file, it may be specified.

```
          COMPILE INVCL4
```

will generate a file named INVCL4.BC while the source file in the above example will be saved as INVCTL.BA.

7.6 LENGTH COMMAND

The LENGTH command returns the length of the user's current program in memory. For example:

```
          LENGTH
          710 CHARACTERS (2 BLOCKS)
```

The LENGTH command may also be used to give the length of lines in a program, by specifying the line numbers after the work LENGTH. As an example,

```
          LENGTH  1ØØ-2ØØ
          354 CHARACTERS (1 BLOCK)
```

The maximum size of a program to be run depends upon the number of variables in the program as well as the amount of text. This size varies between about 13 and 18 blocks. An 18 block file will not always execute, but may be edited.

7.7 CATALOG COMMAND

Giving the CATALOG command causes the user's file directory to be printed on the console. For example:

```
CATALOG
PPB        .BA        4         3/29/71
 ↑          ↑         ↑             ↑
name     extension  size    creation date
```

To obtain a CATALOG of files on a device other than the systems device, one can give the command

```
          CATALOG DEV:
```

For example:

```
          CATALOG DTA4:
```

lists the files on DECtape unit 4.

7.8 COMMANDS FOR INPUT/OUTPUT DEVICES

EDUBASIC has several commands specifically for I/O. However, it should be remembered that OS/8 handles all I/O (except the console) for EDUBASIC and many system commands should be given while under the monitor (e.g., ASSIGN).

7.8.1 TAPE COMMAND

The TAPE command is used to disable the terminal echo feature when reading a paper tape with the low-speed (terminal) reader. The command is given as follows:

TAPE {*initial line number, step*}

EDUBASIC will add line numbers to a file if no line numbers exist on the tape (especially data files). The tape is inserted in the low-speed reader and the reader control switch set to START.

Prior to giving the TAPE command, the user would set up conditions such that the system expects the program. TAPE does not scratch memory. For example, giving the following commands:

```
NEW ADDREC
TAPE
```

causes the system to await the new program file ADDREC which is to be entered to the system via the terminal tape reader. Giving the TAPE command disables the echo feature so that the program is not listed on the terminal as it is read. The same function would be served by the following commands:

```
OLD ADDREC
TAPE
```

7.8.3 PUNCH and NPUNCH

It is sometimes necessary to produce a paper tape using the low speed punch on the console teletype. The PUNCH (and NPUNCH) is used to create this tape.

```
          PUNCH  n1,n2-n3
          NPUNCH  n1,n2-n3
```

These commands punch a copy of the file currently in memory; the latter command produces no line numbers. The line numbers may be used to indicate which lines are to be punched.

The user types the word PUNCH, types a carriage return, and turns on the paper tape punch. Typing LISTNH, turning on the paper tape punch, and then typing a RETURN accomplishes approximately the same result, with the exception that leader is not punched and READY is punched after the program has been punched. PUNCH also punches the program name, extension, and date on the paper tape, which may be read by the user.

Note that when reading in a tape, the name, extension, and date punched by PUNCH should not read in. Place the tape in the reader after this information.

7.8.4 MARGIN COMMAND

The maximum line length can be changed using the margin command. The margin command is of the form:

```
          MARGIN number
```

The above statement changes the line length on all output devices from 72 to the specified *number*. MARGIN is in effect until another MARGIN command is given. Even leaving BASIC and then calling it in again or re-bootstrapping will not change the margin back to 72.

The maximum line number is in effect for all commands and all output devices. It is not in effect for character string output.

The following description will help the user to more fully understand the MARGIN command. The user may continue typing his line of text or command until the specified margin is reached. At this point an automatic Carriage RETURN/LINE FEED is performed and the user is allowed to keep typing. Only by striking the RETURN key does the user enter his command. No commands are affected by the margin command, i.e., operation of BASIC is the same with a line of 72 characters or a line of 5 characters. Even though a single statement may be printed as 10 lines with the new margin, those 10 lines are considered as 1 line to the computer.

7.9 SPECIAL CONTROL CHARACTERS

Some characters previously discussed are reviewed here. Additional control characters are available from OS/8.

7.9.1 RETURN KEY

Typing the RETURN key echoes as a carriage return/line feed operation on the terminal, as long as the terminal is not in TAPE mode. RETURN is used to indicate the end of a line typed. The RUBOUT key is of no use for corrections on the line just typed after the RETURN is typed. The line has been entered into the 'source' buffer.

7.9.2 LINE FEED KEY

The LINE FEED key causes the current line to be echoed, free of rubouts, up to the point at which the user typed LINE FEED. A RETURN must be typed to cause execution (command) or enter the line (BASIC statement). As an example,

```
    1ØØ  PRNT\TN\INT "MY NME\EM\AME IS LAH\HAL\HAL" (LINE FEED)
    1ØØ  PRINT "MY NAME IS HAL"
```

where the carriage position (next position to be printed) is after the last quotation mark. Additional characters may be added to the line or the RETURN key may be typed to accept the line as is.

7.9.3 RUBOUT KEY

The RUBOUT key is used as an eraser for the current line. If typed in TAPE mode, the RUBOUT key is ignored; otherwise, it causes the character most recently typed to be deleted. The erased characters are shown on the terminal paper between back slashes. For example,

```
        1Ø  LEF X=X*X
```

could be corrected by typing the RUBOUT key 7 times (to remove the F) and typing the remainder of the line correctly. The line would look as follows on the terminal paper:

```
        1Ø  LEF X=X*X\X*X=X F\T X=X*X
```

and would appear to the system as:

```
          1Ø  LET X=X*X
```

In cases where the mistake is toward the beginning of a line, it may be easier to simply retype the entire line. For example,

```
          1Ø  LEF X=X*X
          1Ø  LET X=X*X
```

Once the second line is entered into the system, the first line numbered 10 is deleted.

7.9.4 CTRL/C

CTRL/C returns control to the OS/8 keyboard monitor. BASIC may be recalled by typing R BASIC in response to the dot given by OS/8. START may also be typed and in most cases returns the user to EDUBASIC and into the program upon which he was working before CTRL/C was typed.

7.9.5 CTRL/P

By typing a CTRL/P (hold down the CTRL key and type the P key, release both), the user causes BASIC to return to command mode, where commands can be given or editing done. CTRL/P stops whatever BASIC was doing at the time and returns control of the system to the user.

7.9.6 CTRL/U

The CTRL/U combination deletes the current input line. This combination is useful when a long command has been typed and is no longer wanted. Rather than use the RUBOUT key repeatedly, CTRL/U cancels the entire line. This feature can be used when typing either commands or statements. The entire physical line is deleted.

7.9.7 CTRL/O

The CTRL/O combination suppresses output on the Teletype until the next time CTRL/O is typed (or CTRL/P is typed). When a program produces a large amount of output (usually in tabular form), the user may not wish to wait for the printing of the complete information. CTRL/O enables the user to monitor the output while not stopping it completely. Typing CTRL/O while output is occurring still allows the computer to output the data, but the Teletype does not print it. This speeds up the output process, since the Teletype is a rather slow device. The second time CTRL/O is typed, the output is again sent to the Teletype for as long as the user wishes.

CTRL/P, on the other hand, will completely stop the output. Think of CTRL/O as a switch, the first setting of which creates a condition and the second setting releases the condition.

7.9.8 TAB CHARACTER

The TAB character or CTRL/I combination allows the user to insert a tabular format into his typed material. When entering a program to the system, the TAB character allows formatting. The BASIC editor considers each line as being broken into tab stops eight spaces apart across the line. Typing the TAB character causes the printing head to move to the next of those stops on the line.

If using a model 33 Teletype, the TAB echoes as spaces. The model 35 Teletype has built-in hardware tabs.

7.9.9 CTRL/Z

The CTRL/Z combination is used to mark the end of a file; when inputting data from a file, a CTRL/Z character marks the end of the recorded data.

CHAPTER 8

DETAILS OF VIRTUAL ARRAYS

8.1 INTRODUCTION

The virtual array facility provides the means for an EDUCOMP BASIC program to operate on data structures that are too large to be accommodated in memory at one time. To accomplish this, BASIC uses the disk or DECtape file system for storage of data arrays, and only maintains portions of these files in memory at any given time.

An essential difference between real arrays and their virtual counterparts is the order in which array elements are referenced. In real arrays, the referencing algorithm has no effect on the time it takes to accomplish the references; while for virtual arrays, this order can have a significant effect on the program execution time. This chapter gives the user an in-depth look at the algorithms used in the virtual array processor, in order that users concerned with efficiency can optimize their use of this facility.

Each DECtape or 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. This virtual array processor is invisible to the user and BASIC performs all mapping functions automatically.

Virtual arrays are stored as unformatted binary data. This format means that no I/O conversions (internal form-to-ASCII) need 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.

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  OPEN "FILE" AS FILE 1
20  DIM #1,X(10),Y(10)
```

Program TWO contains

```
10  OPEN "FILE" AS FILE 1
20  DIM #1,Z(10),X(10)
```

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 numeric data, and both are 11 elements (X(0),...,X(10)) long. These two arrays, then, correspond in position, type, and dimension.

References to the array X (in ONE) and to the array X (in TWO) do not refer to the same data, even though both are using the same virtual file (FILE). The concept of using relative position, rather than name, to identify data items is familiar to users of the FORTRAN COMMON facility.

Within a single EDUBASIC program it is possible to redefine a single virtual array file on the same channel for the purpose of reallocating the data within the file. For example:

```
145  OPEN "DATA" FOR INPUT AS FILE 1
15Ø  DIM #1, A$(1Ø)=4
155  DIM #1, B$(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 file is open for input only and 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(1Ø)
8Ø  DIM #1, B(1Ø)
```

are not equivalent to the statement:

```
9Ø  DIM #1, A(1Ø),B(1Ø)
```

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:

```
 5Ø  OPEN "VALUES" AS FILE 1
 55  OPEN "VALUES" AS FILE 2
       .
       .
1ØØ  DIM #1, X$(2Ø)
1Ø5  DIM #2, Y$(2Ø)
```

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.

8.2 ARRAY STORAGE

Numbers (floating point) are stored in four words (8 characters) in virtual files so that an integer number of numbers may be contained in one segment (256 words). The only limit on the number of elements in a numeric virtual array is the size of the device.

Virtual array elements are limited to a length of 2046 characters (bytes). The number of data elements stored in each disk or DECtape 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 15 characters, and can be specified as a multiple of three: 3, 6, 9, 12, 15, 18, ... 2046.

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 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.

8.3 TRANSLATION OF ARRAY SUBSCRIPTS INTO FILE ADDRESSES

In order to translate an array subscript into a file address, EDUBASIC computer (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. 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 different accessing arrangements for the same file in one or more programs. For example, the user can reference the same data as either a series of 16-byte strings (A$) or 32-byte strings (B$), with the following statements:

```
10 OPEN 'FIL1' AS FILE 1   !VIRTUAL ARRAY FILE.
20 DIM #1,A$(1000) = 16    !16 CHARACTER STRINGS.
30 DIM #1,B$(500) = 32     !32 CHARACTER STRINGS.
```

The user should keep in mind that in EDUCOMP BASIC, 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 (except for string arrays) as well as singly dimensioned ones. Two-dimensional arrays are stored on disk or DECtape (and in core) linearly, row-by-row. Thus, in the case of an array X(1,2), the array appears logically as:

X(0,0)	X(0,1)	X(0,2)
X(1,0)	X(1,1)	X(1,2)

while physically it is stored as:

X(Ø,Ø)	lowest address
X(1,Ø)	
X(2,Ø)	
X(Ø,1)	
X(1,1)	
X(2,1)	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 virtual array. Program MAT1 below does this far more efficiently than program MAT2 below:

```
 1Ø REM PROGRAM 'MAT1' TO COMPUTE SUMS EFFICIENTLY
 2Ø REM 'AR' CONTAINS VIRTUAL ARRAY
 3Ø REM R(I) IS SUM OF ROW I
 4Ø REM C(J) IS SUM OF COLUMN J
 5Ø OPEN "AR" AS FILE 1                  !OPEN VIRTUAL FILE
 6Ø DIM #1,A(1Ø,5Ø)                      !1Ø ROWS, 5Ø COLUMNS
 7Ø DIM R(1Ø), C(5Ø)
 8Ø FOR R=1 TO 1Ø:R(R)=Ø:NEXT R          !INITIALIZE SUMS
 9Ø FOR C=1 TO 5Ø:C(C)=Ø:NEXT C
1ØØ FOR J = 1 TO 5Ø                      !OPERATE ONE COLUMN AT A TIME
11Ø FOR I = 1 TO 1Ø                      !AND EACH ROW IN COLUMN
12Ø R(I) = R(I) + A(I,J)                 !TOTAL ACROSS ROW
13Ø C(J) = C(J) + A(I,J)                 !TOTAL DOWN COLUMN
14Ø NEXT I                               !NEXT ROW IN COLUMN
15Ø NEXT J                               !NEXT COLUMN
16Ø FOR R=1 TO 1Ø:PRINT R(R);:NEXT R     !PRINT ROW TOTALS
17Ø FOR C=1 TO C:PRINT C(C);:NEXT C      !PRINT COLUMN TOTALS
999 END

READY
```

```
 10 REM PROGRAM 'MAT2' HAS INEFFICIENT USE OF VIRTUAL CORE
 20 REM 'AR' CONTAINS VIRTUAL ARRAY
 30 REM R(I) IS SUM OF ROW I
 40 REM C(J) IS SUM OF COLUMN J
 50 OPEN "AR" AS FILE 1                  !OPEN VIRTUAL FILE
 60 DIM #1,A(10,50)                      !10 ROWS, 50 COLUMNS
 70 DIM R(10), C(50)
 80 FOR R=1 TO 10:R(R)=0:NEXT R          !INITIALIZE SUMS
 90 FOR C=1 TO 50:C(C)=0:NEXT C
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 FOR R=1 TO 10:PRINT R(R);:NEXT R     !PRINT ROW TOTALS
170 FOR C=1 TO C:PRINT C(C);:NEXT C      !PRINT COLUMN TOTALS
999 END

READY
```

In virtual 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 follows:

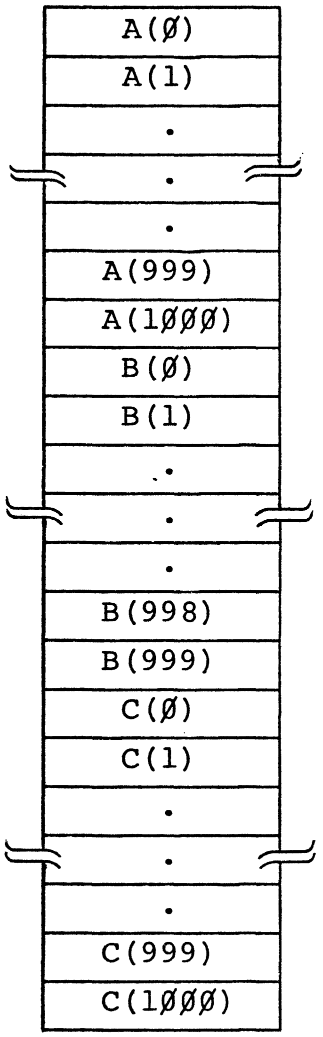


Figure 8-1 Virtual Array File Layout

8.4 ACCESS TO DATA IN VIRTUAL ARRAYS

Only a portion of a virtual array is in memory at any given time. This data is transferred directly between the disk and an I/O buffer in the user core area, created when the OPEN statement is executed. This buffer is 256 words (one segment) long. For each virtual array file, EDUBASIC notes (1) the segment of the file in the buffer, and (2) whether or not the data in the buffer has been modified since it was read into core.

After BASIC translates a virtual array address into a file address, it checks whether or not 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 flowcharted in Figure 8-2.

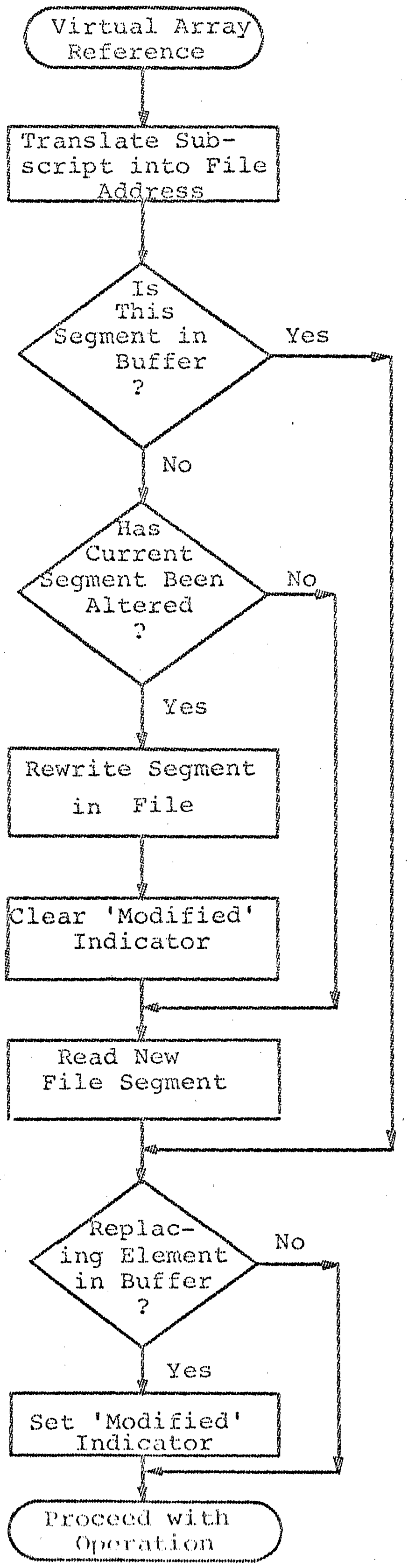

Virtual Array Reference
Translate Sub-script into File Address
Is This Segment in Buffer ?
Yes
No
Has Current Segment Been Altered ?
No
Yes
Rewrite Segment in File
Clear 'Modified' Indicator
Read New File Segment
Replacing Element in Buffer ?
No
Yes
Set 'Modified' Indicator
Proceed with Operation

Figure 8-2

APPENDIX A
LANGUAGE SUMMARY

A.1 SUMMARY OF VARIABLE TYPES

Type	Variable Name	Examples
Numeric (floating point)	single letter optionally followed by a single digit	A I X3
Character String	any letter name followed by a $ character	M$ R/$

Numeric Matrix	any numeric variable name followed by one or two dimension elements in parentheses	S(4) E(5,1)
		N2(8) V8(3,3)

Character String Matrix	any character string variable name followed by a one dimension element in parentheses	C$(1)

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	
String	+	concatenation	string constants and variables

A.3 EDUCOMP BASIC STATEMENT SUMMARY

The following summary of BASIC statements defines the general format for the statement and gives a brief explanation of its use.

CHAIN *dev:filnam.ex,line number*	Terminates execution of user program, loads and executes the specified program starting at the *line number* if included.
CLOSE *n*	Closes the logical file specified. If no file number is specified, closes all files which are open.
DATA *data list*	Used in conjunction with READ to input *data* into an executing program.
DIM *variable(n), variable (n,m)*	Reserves space for lists and tables according to subscripts specified after *variable* name.
END	Placed at the physical end of the program to terminate program execution.
FOR *variable = expression1* TO *expression2* STEP *expression 3*	Sets up a loop to be executed the specified number of times.
GOSUB *line number*	Used to transfer control to the first *line* of a subroutine.
GO TO *line number*	Used to unconditionally transfer control to other than the next sequential *line* in the program.
IF *expression rel.op. expression* THEN *line number*	Used to conditionally transfer control to the specified *line* of the program.
IF *expression rel.op. expression* THEN *statement*	Used to conditionally execute the *statement* after the THEN.

IF *variable* THEN *statement*	For the logical 'IF', when the *variable* is zero, the *statement* is not executed.
INPUT *list*	Used to input data from the terminal keyboard or papertape reader.
INPUT *#expression, list*	Inputs from a particular device.
INPUT LINE *string*	Inputs a record at a time. Accepts commas and quotes, and recognizes the RETURN as the delimiter.
INPUT LINE *#expression, string*	Inputs a record from a specified device.
KILL *file*	Unsaves the *file*. *File* may be of the form *dev:filnam.ex* or a scalar string variable. (Must have an extension.)
[LET] *variable = expression*	Used to assign a value to the specified *variable(s)*.
NEXT *variable*	Placed at the end of a FOR loop to return control to the FOR statement.
ON *expression* GOTO *list of line numbers*	The formula is evaluated and control transfers to the first, second, third, etc., *line number* depending on whether the truncated evaluation is 1,2,3, etc. If the magnitude of the index is greater than 2047, an error is generated. Otherwise, if the index is out of range, control passes to the next statement.
ON *expression* GOSUB *list of line numbers*	Same as the ON-GOTO statement except that a GOSUB is generated.
OPEN *file* FOR $\left\{\begin{matrix}\text{INPUT}\\\text{OUTPUT}\end{matrix}\right\}$ AS FILE *#n*	Opens a sequential *file* for input or output. *File* may be of the form *dev:filnam.ex* or a scalar string variable. Variables must be DIMensional in a separate statement.
PRINT *list*	Used to output data to the terminal. The *list* can contain expressions or text strings.

PRINT *text*	Used to print a message or a string of characters.
PRINT *#expression, list*	Outputs to a particular output device, as specified in an OPEN statement.
PRINT TAB *(x)*	Used to space to the specified column unless the column is already passed in which case TAB is ignored.
RANDOMIZE	Causes the random number generator to calculate different random numbers every time the program is run.
READ *variable list*	Used to assign the values listed in a DATA statement to the specified *variables*.
REM *comment*	Used to insert explanatory *comments* into a BASIC program.
RESTORE	Used to reset data block pointer so the same data can be used again.
RETURN	Used to return program control to the statement following the last GOSUB statement.
STOP	Used at the logical end of the program to terminate execution.

A.4 EDUCOMP BASIC COMMAND SUMMARY

COMMAND	EXPLANATION
CATALOG	Returns the user's file directory. Unless another device is specified following the term CAT or CATALOG, the 'DSK' is the assumed device.
COMPILE	Allows the user to store a compiled version of his BASIC program. The file is stored with the current name and the extension .BC. Or, a new file name can be indicated and the extension .BC will still be appended.
DELETE *n1,n2-n3,n4*	Removes line numbers *n1* and *n4,* as well as lines *n2* through *n3* inclusive, from the program currently in memory.
EDIT *line number*	After EDIT followed by a *line number* and RETURN is typed, EDUBASIC waits until the search character is typed (but not printed). The specified line is then listed until the first occurrence of the search character.
EXECUTE *dev:filnam.ex,line number*	Runs the specified program. Compiled or .BC programs are tried first.
LIST *dev:n1,n2-n3*	Prints out the current program on the device specified (console assumed). Prints out the specified program line(s) if given.
NLIST *dev:n1,n2-n3*	Same as LIST but without line numbers.
LISTNH *dev:n1,n2-n3*	Lists the lines associated with the specified numbers but does not print a header line.
MARGIN *line number*	Changes the maximum line length on all output devices.

<u>COMMAND</u>	<u>EXPLANATION</u>
NEW *filnam*	Does a SCRatch and sets the current program name to the one specified.
OLD *dev:filnam.ex line number,step*	Does a SCRatch and inputs the program from the specified file. *Line numbers* are added (if specified) to ASCII files not already containing them.
OVERLAY *dev:filnam.ex,increment*	Works like OLD but does <u>not</u> scratch.
PUNCH *n1,n2-n3*	Punches the current program on the fastest available papertape punch.
NPUNCH *n1,n2-n3*	Same as PUNCH but no line numbers are punched.
RENAME *filnam*	Changes the current program name to the one specified.
REPLACE *dev:filnam.ex n1,n2-n3*	Replaces the specified file with the current program. Parts of the program may be replaced by specifying particular line numbers.
NREPLACE *dev:filnam.ex n1,n2-n3*	Same as REPLACE but line numbers are not saved.
RESEQUENCE *line number, increment*	Renumber the lines in a program and changes appropriate GOTO, IF-THEN, etc. If *line number* is not specified, starts at 100 with *increments* of 10.
RUN	Executes the program in memory.
RUNNH	Executes the program in memory but does not print a header line.
SAVE *dev:filnam.ex n1,n2-n3*	Outputs the program in memory as the specified file.
NSAVE *dev:filnam.ex n1,n2-n3*	Like SAVE but does not save line numbers.
SCRatch	Erases the entire storage area.

COMMAND	EXPLANATION
SEARCH *n1-n2/stringA/*	Lists all lines in the range *n1* to *n2* that contain *string A* anywhere in the line.
SEARCH *n1-n2/stringA/stringB/LIST*	*String B* replaces all occurrences of *string A* and these lines are listed if *LIST* is specified.
TAPE *line number,increment*	Like OVERLAY, but the file comes from the fastest available papertape reader.

SPECIAL CONTROL CHARACTER SUMMARY

CONTROL CHARACTER	EXPLANATION
CTRL/C	Causes the system to return to the OS/8 monitor.
CTRL/P	Returns BASIC to the READY mode.
CTRL/O	Used as a switch to suppress/enable output of a program on the user terminal. Echoes as ↑O.
CTRL/U	Deletes the current typed line, echoes as ↑U and performs a carriage return/line feed.
CTRL/Z	Used as an end-of-file character.
LINE FEED Key	Used to list the current line.
RETURN Key	Enters a typed line to the system, results in a carriage return/line feed operation at the user terminal.
RUBOUT Key	Deletes the last character typed on that physical line. Erased characters are shown on the teleprinter between back slashes.
TAB or CTRL/I	Performs a tabulation to the next of nine tab stops (eight spaces apart) which form the terminal printing line.

A.6 SUMMARY OF FUNCTIONS

Under the Function column, the function is shown as:

Y=function

where the character '\$' is appended to Y if the value returned is a character string.

Function	Explanation
Y=ABS(X)	returns the absolute value of X.
Y=ATN(X)	returns the arctangent of X in radians.
Y=COS(X)	returns the cosine of X in radians.
Y=EXP(X)	returns the value of e↑X, where e=2.71828.
Y=INT(X)	returns the greatest integer which is less than or equal to X.
Y=LOG(X)	returns the natural logarithm of X, $\log_e X$.
Y=PI	has a constant value of 3.141593.
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 in radians.
Y=SQR(X)	returns the square root of X.
Y=TAN(X)	returns the tangent of X in radians.
Y=POS(X)	returns the current position of the print head for I/O channel X, Ø is the user's Teletype.
Y\$=TAB(X)	moves print head to position X in the current print record, or is disregarded if the current position is beyond X. (The first position is counted as Ø.)
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\$=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.

APPENDIX B

ERROR MESSAGES

The error messages appearing onthe following pages are designed to specifically to help the use pinpoint the 'bug' in his program quickly. An arrow (↑) is used in many statement to point to the offending syntax and in most error messages the line number of the statement in error is given.

B.1 COMPILER ERRORS

MESSAGE / **EXAMPLE** / **EXPLANATION**

CAN'T 'IF' VIRTUAL CORE STRING

```
1ØØ OPEN "CORE" AS FILE 1
11Ø DIM #1,A$=3ØØ
12Ø IF A$="ONE"THEN PRINT "ONE"
13Ø END
RUNH

 12Ø IF A$="ONE"THEN PRINT"ONE"
        ↑
```

User cannot have a virtual core string in an 'IF' statement.

CHARACTERS AFTER STATEMENT END

```
1ØØ INPUT LINE A$,B$
11Ø END
RUNH

 10Ø INPUT LINE A$,B$
                  ↑
```

Statement has unrecognized characters at the end of it.

COMPILER ERROR

```
1ØØ DIM A$(3,2)
11Ø END
RUNH

 1ØØ DIM A$(3,2)
             ↑
```

The user has used a legal statement in an illegal manner.

'END' NOT LAST AT LINE 11Ø

```
1ØØ PRINT "A"
11Ø END
12Ø GO TO 1ØØ
RUNH
```

The last statement must be an 'END' statement.

MESSAGE	EXAMPLE	EXPLANATION
EXTRA OPERATOR	2ØØ IF A==B THEN 3ØØ 3ØØ END RUNH 2ØØ IF A==B THEN 3ØØ ↑	The statement contains an extra operator.
EXTRA '('	1ØØ Y=((A+B)/5 11Ø END RUNH 1ØØ Y=((A+B)/5 ↑	Line has one more left parenthesis than right parenthesis.
EXTRA ')'	1ØØ Y=(A+B))/5 11Ø END RUNH 1ØØ Y=(A+B))/5 ↑	Line has one more right parenthesis than left parenthesis.
FILE TOO LARGE	1ØØ OPEN "DTA1:HALT" AS FILE 1 11Ø DIM #1,Q(1ØØØØØØ) 12Ø END RUNH 11Ø DIM #1,Q(1ØØØØØØ) ↑	A file is dimensioned too large for any device.
'FOR' WITHOUT 'NEXT' AT LINE 1ØØ	1ØØ FOR I=1 TO 5Ø 12Ø END RUNH	A variable used as the index in a 'FOR' statement does not appear in a corresponding 'NEXT' statement.

MESSAGE | **EXAMPLE** | **EXPLANATION**

ILLEGAL ASSIGNMENT

The assignment made is not acceptable to BASIC

ILLEGAL CONSTANT

```
1ØØ Y=88888888888
11Ø END
RUNH

 1ØØ Y=88888888888
                  ↑
```

A number inside the program cannot be longer than 10 digits. A number that is input can be any length.

ILLEGAL INTEGER

```
1ØØ DIM A$="3Ø"
11Ø END
RUNH

 1ØØ DIM A$="3Ø"
            ↑
```

A string's length must be an integer number.

ILLEGAL STRING VARIABLE

```
1ØØ A1$="PDP"
11Ø END
RUNH

 1ØØ A1$="PDP"
         ↑
```

A string variable must be a single letter followed by a '$'.

ILLEGAL SUBSCRIPTING

```
1ØØ PRINT A(5
11Ø END
RUNH

 1ØØ PRINT A(5
            ↑
```

A subscripted variable has been dimensioned or used incorrectly.

ILLEGAL SYNTAX

```
1ØØ A$=7
11Ø END
RUNH
```

A string variable has been used where a numeric variable should have been used.

MESSAGE | EXAMPLE | EXPLANATION

ILLEGAL USE OF FUNCTION

```
1ØØ Y = LOG 1Ø (X)
11Ø END
RUNH

 1ØØ Y = LOG 1Ø (X)
                ↑
```

A function must be followed by an open parenthesis, an argument and then a closed parenthesis.

ILLEGAL VARIABLE

```
1ØØ PRINT FI
11Ø END
RUNH

 1ØØ PRINT FI
            ↑
```

A variable must be one letter or one letter followed by a number.

INCONSISTENT SUBSCRIPTING

```
 1ØØ DIM A$(1Ø)
 11Ø PRINT A$(1,1)
2ØØØ END
RUNH

 11Ø PRINT A$(1,1)
                 ↑
```

A string variable has been dimensioned as a one-dimensional variable and utilized as a two-dimensional variable.

MISPLACED ',' OR ';'

```
1ØØ A=5
11Ø END
RUNH

 1ØØ A=5,
        ↑
```

A comma or semicolon doesn't belong where it was placed.

MISSING '='

```
1ØØ Y-5
11Ø END
RUNH

 1ØØ Y-5
      ↑
```

Statement requires an equal sign.

MISSING ','

The syntax requires a comma in the designated position.

MESSAGE — EXAMPLE — EXPLANATION

MISSING OPERATOR

```
1Ø PRINT "TEST1""TEST2"
2Ø END
RUNH

 1Ø PRINT"TEST1""TEST2"
                ↑
```

The statement is missing an arithmetic or relational operator, or a punctuation mark.

MISSING '('

```
1ØØ DIM A$(4)=2Ø
11Ø A$=5
12Ø END
RUNH

 11Ø A$=5
        ↑
```

Subscripted variable has been used as a non-subscripted variable.

MISSING QUOTE

```
1ØØ PRINT "ABC
11Ø END
RUNH

 1ØØ PRINT "ABC
              ↑
```

A string variable assignment statement must have the assigned value surrounded by quotes.

MISSING VARIABLE

```
1Ø PRINT TAB();X
2Ø END
RUNH

 1Ø PRINT TAB();X
               ↑
```

The statement is missing a numeric variable or constant.

MIXED MODE EXPRESSION

```
1ØØ Y="ABCD"
11Ø END
RUNH

 1ØØ Y="ABCD"
        ↑
```

A numeric variable was used where a string variable should be used.

MESSAGE	EXAMPLE	EXPLANATION
'NEXT' WITHOUT 'FOR' AT LINE 11Ø	1ØØ FOR I=1 TO 5Ø 11Ø NEXT A 12Ø END RUNH	A variable used as the index in a 'NEXT' statement does not appear in a corresponding 'FOR' statement.
NO 'END' STATEMENT	1ØØ PRINT "ABC" RUNH	Program must have an 'END' statement.
NON-BASIC STATEMENT	1ØØ DEF FNA(X)=X↑2 11Ø END RUNH 1ØØ DEF FNA(X)=X↑2 ↑	BASIC does not understand the statement.
PROGRAM TOO LONG		Program is too long for BASIC to compile.
'READ' WITHOUT 'DATA' ON LINE 1ØØ	1ØØ READ A 2ØØ END RUNH	The program contains one or more 'READ' statements and no 'DATA' statements.
TOO MANY ARRAYS AT LINE 11Ø	1ØØ DIM A(2ØØØ) 11Ø END RUNH	Not enough space in core for all of the subscripted variables in the program.
TOO MANY LITERALS		The program has too many literals for BASIC to handle.
TOO MUCH DATA AT LINE 31		Program has too much data in its DATA statements for BASIC to handle.

MESSAGE	EXAMPLE	EXPLANATION
UNDEFINED LINE NUMBER AT LINE 1ØØ	1ØØ GO TO 15Ø 11Ø END RUNH	Any statement which references a non-existent line (GOTO, GOSUB, ON-GOTO, ON-GOSUB, IF-THEN).
VARIABLE DIMENSIONED TWICE	1ØØ DIM A$=15 11Ø DIM A$=2Ø 12Ø END RUNH 11Ø DIM A$=2Ø ↑	A variable must appear in only one dimension statement.

B.2 RUNTIME ERRORS

MESSAGE	EXAMPLE	EXPLANATION
ARRAY OF WRONG SIZE IN LINE 1ØØ	9Ø OPEN "A" AS FILE 1 1ØØ DIM #1,A(1ØØØ) 2ØØØ END RUNH	A virtual file cannot be dimensioned larger than at the time it was created unless the original is deleted.
BAD FILE FOR CHAIN IN LINE 1Ø	1Ø CHAIN "CDR3.DA" 2Ø END RUNH	Only a BASIC program can be chained.
BAD INPUT IN LINE 1ØØ	1ØØ INPUT A 11Ø END RUNH ? 1ØP	Numeric variables may have only numbers as input.
CAN'T OPEN OUTPUT FILE IN LINE 1ØØ	1ØØ OPEN "CDR:" FOR OUTPUT AS FILE 1 111Ø END RUNH	A sequential access file cannot be opened because the device specified is full or there is a mistake in the 'OPEN' statement.
CHANNEL NOT OPEN FOR INPUT IN LINE 1ØØ	1ØØ INPUT #1,A 11Ø END RUNH	No file or device has been opened under the specified channel number.
CHANNEL NOT OPEN FOR OUTPUT IN LINE 1ØØ	1ØØ PRINT #1,A 11Ø END RUNH	No file or device has been opened under the specified channel number.

MESSAGE	EXAMPLE	EXPLANATION
CHANNEL OUT OF RANGE IN LINE 1ØØ	1ØØ OPEN "EDU" AS FILE 5 11Ø END RUNH	Files can only be opened under numbers 1-4.
DEVICE ERROR		A device that the system has been configured for has been used in an illegal manner.
DEVICE FULL IN LINE 2Ø	1Ø OPEN "AFILE" AS FILE 1 2Ø DIM #1,A(5ØØØØ) 2ØØ END RUNH	Device specified doesn't have enough contiguous blocks to contain file.
DEVICE NOT AVAILABLE IN LINE 1ØØ	1ØØ OPEN "AAA:RISK" AS FILE 1 11Ø END RUNH	Any device for which the system is not configured cannot be accessed.
DIVISION BY ZERO IN LINE 7Ø	7Ø PRINT T/Y 2ØØØ END RUNH	Division by zero is an undefined operation.
END OF FILE IN LINE 2Ø	5 DIM A$=12Ø 1Ø OPEN "CDR" FOR INPUT AS FILE 1 2Ø INPUT LINE #1, A$ 30 GOTO 2Ø 1ØØ END RUNH	An input device or file has no more elements remaining.
ERROR CLOSING FILE IN LINE 1ØØ	1ØØ OPEN "K" FOR OUTPUT AS FILE 1 11Ø PRINT #1,A 115 GOTO 11Ø 12Ø END RUNH	A sequential access file cannot be closed because the specified device is full.

MESSAGE	EXAMPLE	EXPLANATION
ERROR READING FILE	1ØØ OPEN"READ"FOR INPUT AS FILE 1 11Ø END RUNH	Input from device or file contains an unrecognizable error.
FILE ALREADY OPEN IN LINE 1ØØ	1ØØ OPEN"LOAN"FOR OUTPUT AS FILE 1 1Ø5 A=1Ø 11Ø PRINT #1,A 12Ø GOTO 1ØØ 4ØØ END RUNH	A file or device which the program attempts to open has previously been opened.
FILE NOT FOUND IN LINE 1ØØ	1ØØ CHAIN "INFORT",2ØØ 11Ø END RUNH	File specified does not exist.
FUNCTION ARG TOO BIG IN LINE 1ØØ	1ØØ Y=2↑1ØØØØ 11Ø END RUNH	BASIC cannot manipulate the function with the specified arguments.
LINE NOT FOUND IN LINE 2ØØ	2ØØ CHAIN "PLOT",12 2ØØØ END RUNH	There is no such line in the program chained.
MID ERROR IN LINE 2Ø	1Ø A$="ABCD" 2Ø B$=MID(A$,8Ø,5) 3Ø END RUNH	A MID function must have a positive integer for its length specification, it must have at least one character to 'MID', and it must not 'MID' past the dimension of the string.
NEGATIVE OR ZERO LOG IN LINE 1ØØ	1ØØ Y=LOG(Ø) 11Ø END RUNH	The LOG function requires a positive argument.

MESSAGE	EXAMPLE	EXPLANATION
NO CLOSING QUOTE IN LINE 1ØØ	1ØØ INPUT A$ 11Ø END RUNH ? "ABC CR.	A string variable must have a closing quote if it has an opening quote.
NOT A BINARY FILE IN LINE 1ØØ	1ØØ DIM #4,A$(1Ø)=2Ø 11Ø END RUNH	A virtual file must be opened before it is dimensioned.
OUT OF DATA IN LINE 1ØØ	1ØØ READ A, B, C, D 11Ø DATA 4,7,2 12Ø END RUNH	A READ statement has no more data available to read.
OUT OF STORAGE IN LINE 5Ø		Program has run out of storage performing an operation.
RETURN WITHOUT GOSUB IN LINE 1ØØ	1ØØ RETURN 11Ø END RUNH	A RETURN statement must only be accessed after a GOSUB command has previously been executed.
SQR OF NEGATIVE ARG IN LINE 1ØØ	1ØØ Y=SQR(-9) 11Ø END RUNH	The SQR function requires non-negative argument.
STEP OF Ø IN LINE 1ØØ	1ØØ FOR I=1 TO 1Ø STEP X 2ØØ NEXT I 4ØØØ END RUNH	The step of a FOR-NEXT loop must be a non-zero number.

MESSAGE	EXAMPLE	EXPLANATION
STOP AT LINE 15Ø	15Ø STOP 2ØØ END RUNH	Execution has been halted by BASIC at the line indicated.
STRING OVERFLOW IN LINE 11Ø	1ØØ DIM A$=4 11Ø A$="123456" 12Ø END RUNH	A string must not be set equal to a length greater than its demension statement.
SUBSCRIPT OUT OF BOUNDS IN LINE 11Ø	1ØØ DIM A(5) 11Ø PRINT A(6) 12Ø END RUNH	A subscript of a subscripted variable has exceeded its DIMension.
TAN OF PI/2 IN LINE 45	45 PRINT TAN(PI/2) 2ØØØ END RUNH	The tangent of PI/2 does not exist.
UNDEFINED ERROR IN LINE 25		An error has occurred which BASIC does not know how to handle.
ZERO TO ZERO POWER IN LINE 1Ø	1Ø PRINT X↑X 2ØØØ END RUNH	Zero to the zero power does not exist.

B.3 COMMAND ERRORS

MESSAGE	EXAMPLE	EXPLANATION
DEVICE ERROR	SAV PTR:	If you try to use a device the system is configured for in an illegal manner, this error statement will result.
DEVICE NOT AVAILABLE	SAV PTT:	System is not configured for the specified device.
ERROR DELETING FILE *n*	UNSAVE COMP	File *n* not found.
ERROR READING FILE	OLD PAYROL	An error has occurred in calling a previously saved program into core.
filnam.ex ALREADY SAVED	SAVE	The specified file already exists on the specified device.
filnam.ex NOT FOUND	OLD COMP	File specified is not found on specified device.
ILLEGAL FILE NAME	OLD A-5	File name must be less than six characters, consisting of alphanumeric characters, and starting with a letter.
LINE NOT FOUND	EDIT 11Ø	When using the EDIT command, the line specified does not exist.

MESSAGE	EXAMPLE	EXPLANATION
LINE NUMBERS MISSING ON *filnam.ex*	OLD A.DA	File called in is missing some or all line numbers.
LINE TOO LONG		Line greater than 124 characters.
NOT A FILE DEVICE	UNSAVE PTP:AA	The specified device can't be used to save files.
NUMBER OUT OF RANGE	1ØØØØ END	Line numbers must be in the range 1-4094.
PROGRAM TOO LONG TO RESEQUENCE	RESEQUENCE	File in core contains too many characters for BASIC to resequence.
SEQUENCE NUMBER OVERFLOW	1ØØ PRINT "A" 11Ø PRINT "B" 12Ø PRINT "C" 13Ø PRINT "D" 14Ø END RESEQUENCE 1ØØØ,1ØØØ	Specified command needs more arguments than were given. When trying to resequence, a line number became greater than 4094.
TEXT BUFFER IS FULL	OLD LCARDS.DA 1Ø,1	File is too big to fit into core or too large to be able to use the SEARCH command.
TOO FEW ARGS	SEARCH 1ØØ-2ØØ/A	Specified command needs more arguments than were given.
TOO MANY LINES	OLD CARD.DA 1Ø,1Ø	User tried to type in or call in a file with more lines than is acceptable to BASIC.
WHAT??	MISTNH	BASIC does not recognize the command.

INDEX