DEC-11-AJPB-D

PDP-11 BASIC

PROGRAMMING MANUAL

Single-User, Paper Tape Software

For additional copies, order No. DEC-11-AJPB-D from Digital Equipment Corporation, Direct Mail, Bldg. 6A-3, Maynard, Mass. 01754 Price $2.50

First Printing, September 1970
Second Printing, December 1970

Your attention is invited to the last two pages of this document. The Reader's Comments page, when filled in and returned, is beneficial to both you and DEC; all comments received are considered when documenting subsequent manuals, and when assistance is requested, a knowledgable DEC representative will contact you. The How To Obtain Software Information page offers you a means of keeping up-to-date with DEC's software.

Supporting and referenced documents:

PDP-11 Handbook (order No. AJO)

PDP-11 Paper Tape Software Programming Handbook (order No. DEC-11-GGPA-D)

These and other DEC documents may be ordered from DEC, Direct Mail Bldg. 6A-3, Maynard, Massachusetts 01754.

PREFACE

This manual contains a comprehensive description of PDP-11 BASIC[1]. As implemented on the PDP-11, BASIC has a few limitations offset by a number of special features which provide added power and flexibility. Among the latter is a particularly strong debugging capability.

For those who plan to learn BASIC from this manual, numerous examples are provided to fully illustrate the use and operation of each BASIC statement. The knowledgeable BASIC user can turn to Appendix A for a summary of the differences from and extensions to Dartmouth BASIC. Appendix B summarizes PDP-11 BASIC's command structure, and Appendix D gives loading instructions.

A knowledge of computers is not prerequisite to the efficient use of BASIC. However, a knowledge of binary notation is required to load the BASIC program into the PDP-11 (see Chapter 7).

PDP-11 BASIC can be used in a minimal PDP-11 system: 4K words of core memory and a Teletype.[2]

For a more elementary description of the BASIC language, see BASIC Programming, by Kemeny & Kurtz, published by John Wiley & Sons, Inc., New York.

---

[1] BASIC is a trademark registered by the Trustees of Dartmouth College, New Hampshire.

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

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION TO THE MANUAL AND BASIC OPERATIONS

1.1 LOADING DATA

PDP-11 BASIC is an on-line, conversational program for use from the Teletype (terminal) keyboard, in a paper tape environment. It can be run in the minimal 4K configuration. Any additional core storage can be utilized for user program storage. If BASIC is not in core, directions for loading and responding to its initial dialogue can be found in Chapter 7.

After the initial dialogue has been completed, BASIC prints

```
READY
```

to indicate that it is in command mode. At this point the user can begin to type in his program.

After performing user services, such as printing data on the terminal printer or punching out or reading in user programs on paper tape, BASIC returns automatically to command mode.

1.2 THE BASIC LANGUAGE

BASIC statements are explained, beginning in Chapter 2, along with the rules of BASIC syntax. Throughout the text, simple examples have been used in an effort to illustrate clearly the operation of each statement. A number of useful sample programs are furnished in Chapter 9 and have been annotated to explain program logic and the working of certain BASIC statements.

1.2.1 Documentation Conventions

Certain conventions are used throughout the manual in clarifying examples of BASIC syntax.

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

```
LET <variable>    =  <expression>
```

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

```
IF <formula> [THEN <statement>     ]
             [THEN <line number>   ]
             [GOTO <line number>   ]
```

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

```
PRINT {list}
```

Items in lower case print, such as list under item c, above, are supplied by the user, according to rules explained in the text. Items in capitals, such as THEN and GOTO under b, above, must appear exactly as shown.

d. User input (from the keyboard) to a running program is underlined whereas program printout is not underlined.

1.2.2 Deferred and Immediate Modes

BASIC programs are most frequently written in deferred mode, that is, with a number preceding each line to indicate its sequence in the program. In this mode, statement execution is deferred until a specific command (e.g., RUN or GOTO) is typed. In deferred mode the program is stored in core memory where it can be edited, debugged, run, rerun, and, if desired, saved on paper tape for future use.

One of the special features of PDP-11 BASIC is that certain statements can be typed without the usual preceding line number. Without a line number, the statement is executed when the RETURN key is typed at the end of the line. The result of the computation is stored for subsequent use, but the statement by which the result was obtained is lost.

This mode of operation is referred to as immediate, and is useful when BASIC is to serve as a quick calculator. Immediate mode is especially useful in debugging deferred (stored) programs. It can be used, for example, to duplicate and test certain computational statements, and to print the value of variables at various stages of program execution.

1.3 KEYBOARD ERROR CORRECTION

Since the keyboard is used for input, typing errors may occur. To expunge unwanted characters from a line being typed, press the RUBOUT key once for each previous character to be deleted. To replace the entire line being typed, type CTRL/U (i.e., hold down the CTRL key while typing the U key; the slash in CTRL/U is shown merely to tie the operations together) then begin again. Input lines are terminated with the RETURN key. If the RETURN key is typed before a typing error is discovered, the line must be edited as described in Section 1.4 below.

1.4 PROGRAM EDITING

Programs being created from the keyboard, or those which have been written previously and read in from paper tape, can be edited in line-oriented fashion.

To change or correct a line, simply retype its line number and then the desired statement. For example, the line

```
40 PRINT A
```

can be replaced by typing

```
40 GOTO 200
```

The PRINT statement would no longer exist; its place would be occupied by the GOTO statement.

To delete a line, type the line number followed by the RETURN key

```
10  <RETURN key>
```

or use the DELETE command. For example,

```
DELETE 10
```

will delete line number 10.

```
DELETE 5,40
```

will delete lines 5 through 40.

```
DELETE 1,8191
```

will delete the entire user program.

1.5 ERROR MESSAGES

As user programs are typed in and executed, an error message may appear on the teleprinter as the result of a typing error, badly formatted or missing statement, etc. The message will appear as follows:

```
ERROR  xxx  AT LINE  yyy
```

where xxx represents an error code and yyy represents the line number of the line containing the error. All error codes are explained in Chapter 6.

1.6 SPECIAL FUNCTIONS

BASIC contains a number of special functions which perform specific mathematical operations. It is not necessary, for example, to create an original program to find the cosine of a given angle. The BASIC mathematical function COS followed by a parenthesized argument (angle) will compute this value automatically for the given angle. BASIC's mathematical functions are described in Chapter 4.

1.7 SPECIAL FEATURES OF PDP-11 BASIC

PDP-11 BASIC has a number of added features, as compared with standard Dartmouth BASIC, which make it simpler to use and which are especially helpful in debugging user programs. These features are noted throughout the text and listed along with BASIC's limitations in Appendix A.

CHAPTER 2

BASIC VOCABULARY AND SYNTAX

Figure 2-1 (on next page) will serve to illustrate some BASIC fundamentals. The program (lines numbered 1Ø through 2ØØ in Figure 2-1) is a typical BASIC program. The REMARK statement in the first line (line 1Ø) and the printed results (following the command RUN) indicate that the program computes and prints interest payments.

2.1 LINES

Each line of the program begins with a number and is terminated with the RETURN key which is non-printing. The line number must be an integer from 1 to 8191. Statements are executed in the ascending order of their line numbers regardless of the sequence in which they appear. To allow later insertion of new lines, it is advisable to number lines by fives or tens.

Notice in Figure 2-1 that each line number is followed by a word indicating what BASIC is to do or how it is to handle the data that follows the word. There are various types of BASIC statements, each is identified by the word which introduces the statement (LET, FOR, IF, DEF, etc.).

All BASIC statements and computations must be written on a single line; they cannot be continued onto a following line. However, more than one statement may be written on a single line when each statement after the first is preceded by a colon. (Multiple statement lines are a special feature of PDP-11 BASIC.) For example:

```
1Ø INPUT A,B,C
```

is a single statement line, whereas

```
2Ø LET X=11: PRINT X,Y,Z: IF X=A THEN 1Ø
```

is a multiple statement line containing three statements: LET, PRINT, and IF. Most statements may be used anywhere in a multiple statement line; exceptions are noted in the discussion of each statement.

```
10 REMARK -- PROGRAM TO COMPUTE INTEREST PAYMENTS
20 PRINT "INTEREST IN PERCENT";: INPUT J
26 LET J = J/100
30 PRINT "AMOUNT OF LOAN";: INPUT A
40 PRINT "NUMBER OF YEARS";: INPUT N
50 PRINT "NUMBER OF PAYMENTS PER YEAR";: INPUT M
60 LET N = N*M
65 LET I = J/M
70 LET B = 1+I
75 LET R = A*I/(1-1/B↑N)
78 PRINT
80 PRINT "AMOUNT PER PAYMENT ="; R
85 PRINT "TOTAL INTEREST     ="; R*N-A
88 PRINT
90 LET B= A
95 PRINT " INTEREST      APP TO PRIN    BALANCE"
100 LET L = B*I
110 LET P = R-L
120 LET B = B-P
130 PRINT L, P, B
140 IF B>=R GOTO 100
150 PRINT B*I, R-B*I
160 PRINT "LAST PAYMENT =" B*I+B
200 END


RUN
INTEREST IN PERCENT? 9
AMOUNT OF LOAN? 2500
NUMBER OF YEARS? 2
NUMBER OF PAYMENTS PER YEAR? 4

AMOUNT PER PAYMENT =  344.9615
TOTAL INTEREST     =  259.6921

 INTEREST      APP TO PRIN    BALANCE
 56.25          288.7115       2211.288
 49.75399       295.2075       1916.081
 43.11182       301.8497       1614.231
 36.3202        308.6413       1305.59
 29.37577       315.5857       990.0043
 22.2751        322.6864       667.3178
 15.01465       329.9469       337.371
 7.590847       337.3707
LAST PAYMENT =  344.9618

STOP AT LINE  200
READY
```

Figure 2-1. A BASIC Program with Results

2.2 REMARK STATEMENT, REM

The REMARK statement is used to insert notes and comments in user programs. Any legal character (see Appendix C) may appear in a REMARK statement. The word REMARK can be abbreviated to REM.

It is often useful to put the name of the program and information on what it does in a REM statement at the beginning. Remarks throughout the body of a long program will help later debugging by explaining the functions of various sections of the program.

REM statements have no effect on the running of the program. They do, however, take up storage space which can be critical in small memory configurations.

When used in a multiple statement line, the REM statement must be last since BASIC ignores everything on a line after REM.

2.3 STOP AND END STATEMENTS

The program in Figure 2-1 concludes with an END statement. The END statement should be included in programs intended for long term use. It assures that they can be executed in other than the single user BASIC environment.

The END statement must be placed at the end of the program, i.e., it must have the highest line number of any program statement.

STOP statements can be used anywhere in a program to terminate program execution; they mark the logical places for program termination rather than the physical end of the program.

A program intended only for temporary use can be written without an END or STOP statement as a terminator; it will terminate normally at the highest numbered line.

STOP is useful in debugging user programs. It can be placed at critical points in the program so that during execution the program will halt at these points. At the halt, PRINT can be used in immediate mode to examine the value of specific variables. If program execution appears correct at that point, execution may be continued with an immediate mode branch to the statement following STOP (see GOTO, Section 3.10), or the STOP statement may be deleted and the program restarted using RUN.

2.4 RUN COMMAND

When the user program has been typed in deferred mode, as in Figure 2-1, it is ready for execution with the RUN command.

When the RUN command is issued, BASIC executes the program. If a fatal error is detected, execution is halted and an error message is printed. If an error is non-fatal, an error message is printed and program execution is continued.

2.5 DATA INPUT

Following the RUN command in Figure 2-1 (after line 200) four questions appear followed by question marks. The questions result from the PRINT statements at lines 20, 30, 40 and 50. The question marks result from the INPUT statements at the end of these multiple statement lines.

The numbers appearing after the questions (underlined for documentation purposes) were typed by the user. How to input to and output from a running user program is described in Chapter 3.

2.5.1 Constants

BASIC accepts constants expressed as integers, decimal numbers, or in exponential format, i.e., a decimal number times some power of ten, such as 23.4E2 which is equivalent to 2340. The E can be read as "times ten to the __th power", where the power is specified by the number following the E.

The following rules apply to numbers in exponential format.

1. The exponent may be unsigned if positive (1234.56E 3). A negative exponent must be signed, e.g., 1234.56E-3.
2. The exponent must be in the range -98ØØ to +98ØØ.

Results of computations are printed out as integer or decimal numbers when the result will fit in nine spaces, i.e., seven digits plus a sign (space when positive) and a decimal point. Outside this range they are printed out in exponential or E format in a maximum of fifteen characters, i.e., sign (if negative) or space, decimal point plus seven significant digits, E symbol, sign or space, and exponent of up to four digits. Shown below are some user-supplied values with their equivalents as printed out by BASIC.

Value Typed In	Value Printed by BASIC
.Ø1	.Ø1
9.9ØØØØØE-3	.ØØ99
9999999	9999999
1.ØØØØØØE6	1ØØØØØØ
2.718281828459Ø45	2.718282
.ØØØØØØØØØ36218	.36218E-9

2.5.2 Variables

A variable is a symbol which represents a number. It is formed by a single letter or a letter followed by a single digit. For example:

I B3 X C8

Variables are used in the program where actual values are not known when writing the program and where certain values are expected to change.

2.5.3 Expressions

Expression, when referred to throughout the manual, implies a constant, a variable, or combination of either or both separated by arithmetic operators and parenthesized, if necessary. Example:

A+B-4/C*1.2+7/(1+A)

Table 2-1 below lists the symbols used to indicate arithmetic operations in BASIC expressions. Section 2.5.4 explains the rules under which BASIC expressions are evaluated.

2.5.4 Evaluation of Expressions

In evaluating expressions, BASIC performs arithmetic operations in the order of priority indicated in Table 2-1. Parenthesized portions of expressions are evaluated first. Nested parenthesized groups are evaluated beginning with the innermost grouping, working outward.

TABLE 2-1

Arithmetic Operators

Symbol in BASIC	Example	Algebraic Notation	Priority	Function
↑ (SHIFT/N)	A↑B	A^B	1.	Exponentiation (raise A to the B^{th} power)[1]
*	A*B	AB	2.	Multiplication
/	A/B	$\frac{A}{B}$	2.	Division
+	A+B	A+B	3.	Addition
-	A-B	A-B	3.	Subtraction

[1] A must be positive and non-zero. See Appendix A, Section 7.

Through the use of parentheses the order of priority of arithmetic operations, and the final value of an expression, can be changed.

This can be shown by evaluating the following expressions, which are alike except for the inclusion of parentheses in the second:

```
1)  A*B↑2+C/2
2)  A*((B↑2+C)/2)
```

Letting A=7, B=2, and C=4 we get:

```
1)  7*2↑2+4/2
```

Evaluating according to BASIC's rules we get, in successive steps:

```
7*4+4/2      then
28 + 2       and finally
3Ø
```

The second expression yields:

```
2)  7*((2↑2+4)/2)      then
    7*((4+4)/2)        then
    7*(8/2)            and finally
    28
```

2.5.5 Arithmetic Functions

In addition to constants and variables, the call names of BASIC arithmetic functions - SIN(X), COS(X), LOG(X), etc., as described in Chapter 4, can be used as expressions or elements of expressions. The external function EXF(X), as described in Chapter 8, may also be used in this manner.

CHAPTER 3

COMPUTATIONAL, DATA INPUT AND OUTPUT, AND LOGICAL STATEMENTS

3.1 LET STATEMENT

The LET statement is used to assign a value to a variable. The general format of the LET statement is:

LET <variable> = <expression>

The following four statements illustrate two forms of the LET statement.

```
10 LET A=1
20 LET B=2
30 LET C=A+1
40 LET X=A+B+C+1
```

When these statements are executed, the value of X at line 40 will be 1+2+2+1 or 6.

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

3.2 PRINT STATEMENT

The PRINT statement is used to output (print) data on the teleprinter. The general format of the PRINT statement is:

PRINT {list}

where the list may consist of an expression, a text string, or both. As the braces indicate, the list is optional.

The PRINT statement alone:

```
20 PRINT
```

may be used to roll the teleprinter platen, inserting a blank line in the program printout.

PRINT statements can be used to perform calculations: by evaluating the expression contained in the list and printing out its constant value. The value resulting from the computa-

tion is printed out but not retained for use in subsequent statements. The preceding example (Section 3.1) could have been written as follows, substituting a PRINT statement for the final LET statement to obtain printed results:

```
10 LET A=1
20 LET B=2
30 LET C=A+1
40 PRINT A+B+C+1
RUN
 6

STOP AT LINE    40
READY
```

The PRINT statement may be used anywhere in a multiple statement line.

3.3 FORMATTING PRINTOUT

By placing a comma after the expression in the PRINT statement, subsequent values will be printed on a single line, as the following example demonstrates.

```
10 LET A=5
20 FOR B=1 TO 5
30 PRINT A+B,
40 NEXT B

RUN
 6              7              8              9              10

STOP AT LINE    40
READY
```

BASIC formats the Teletype print spaces into five print zones of 14 spaces each. When an item in a PRINT statement is followed by a comma, the next value to be printed will appear in the next available print zone.

If the PRINT statement contains several items (e.g., constants and variables), inclusion of an extra comma between two items causes a print zone to be skipped. For example:

```
30 LET A=1: LET B=2: LET C=3
40 PRINT A,,B,C,1.4
RUN
 1                             2              3              1.4

STOP AT LINE    40
READY
```

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

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

STOP AT LINE   40
READY
```

If a tighter grouping of printed values is desired, the **semicolon can be used in the same manner as the comma. It will cause each value to be printed two spaces to the right of the preceding printout. A semicolon following the last item in the list will cause the next printed value to appear** two spaces to the right of the preceding value on the same print line (provided the end of the print line has not been reached). The following example shows various uses of the semicolon and the comma.

```
10 DATA 1,2,3
20 READ A,B,C
30 PRINT A;B;C;
40 PRINT A;B;C
50 PRINT A,B,C
60 PRINT A,;B,C
RUN
 1  2  3  1  2  3
 1             2              3
 1             2              3

STOP AT LINE   60
READY
```

3.4 CHARACTER STRINGS AND EXPRESSIONS IN PRINT STATEMENTS

The PRINT statement may be used to print a message, comment, or any string of characters. This is done by delimiting the characters to be printed with quotation marks. For example:

```
10 PRINT "THIS IS A TEXT STRING"
RUN
THIS IS A TEXT STRING

STOP AT LINE   10
READY
```

A single PRINT statement can be used to print character strings and computed values. For example:

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

would cause BASIC to print the following (where X=83.4):

```
AVERAGE GRADE IS  83.4
```

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

When a comma is separating a text string and a following PRINT list item, the following item is printed beginning at the next available print zone. Semicolons separating text strings and other items are ignored. However, a semicolon appearing as the last item of a PRINT list will always suppress the line feed/carriage return. In the following example the 2 printed in line 30 immediately follows the "Z" text string in line 20. The space separating the Z and the 2 is the sign space; if X equalled minus two, the minus sign would appear between the Z and 2 without a space.

```
10 LET X=2
20 PRINT "GRADE","40";;;"Z";
30 PRINT X
RUN
GRADE          40Z  2

STOP AT LINE   30
READY
```

Any algebraic expression in a PRINT statement will be evaluated with the current values of the variables and the result will be printed. For example:

```
 5 LET A=78.86
10 PRINT "THE RESULT IS" 40*10+23.82-A
RUN
THE RESULT IS 344.96

STOP AT LINE   10
READY
```

The above demonstrates the omission of the format control characters following a text string, as well as the ability of the PRINT statement to print text and perform calculations.

3.5 FOR AND NEXT STATEMENTS

The FOR and NEXT statements are used to mark the beginning and ending points of program loops. Any statements between the FOR statement and its corresponding NEXT statement will be executed repeatedly according to conditions supplied within the FOR statement.

The general format of the FOR statement is:

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

The general format of the NEXT statement is:

NEXT<variable>

The variable following FOR in the FOR statement is referred to as the "control variable". The same variable must appear in the NEXT statement which defines the end of the loop.

The upper and lower values represented by the expressions preceding and following TO in the FOR statement are referred to as the "range" of the control variable. Each value within the range is computed and assigned to the control variable for one iteration of the loop.

The following example illustrates the FOR-NEXT statement and the formation of a program loop.

```
10 LET A=5
20 FOR B=1 TO 5
30 PRINT A+B
40 NEXT B
50  <next statement>
```

In line 20 the variable B is assigned the values 1, 2, 3, 4, and 5. Since it cannot have each of these values simultaneously, a loop is formed beginning with the FOR statement at line 20 and ending with the NEXT statement at line 40. The statements within the loop are re-executed five times, each time with a new value of B. The NEXT statement causes repeated jumps to line 20 until B reaches its final assigned value of 5. When these statements are executed, BASIC prints the values 6, 7, 8, 9, and 10. When B reaches its ultimate value of 5, control passes to whatever statement may follow line 40.

Each FOR statement within a program must be succeeded by a NEXT statement; NEXT cannot be used without a preceding FOR statement.

3.5.1 Use of STEP

In the preceding example, line 2Ø could have been written

```
20 FOR B=1 TO 5 STEP 1
```

The STEP modifier is optional when the step value desired is +1. It is required when the step value desired is other than +1, for example:

```
10 FOR A=1 TO 8 STEP 2
20 PRINT A
30 NEXT A
40  <next statement>
```

When these statements are executed BASIC prints the initial value of 1, then adds 2 to arrive at the next value, etc. It will print out 1, 3, 5, and 7, then program control will revert to line 4Ø, the next statement outside of the loop.

A negative step value is legal:

```
10 LET A=5
20 FOR M=A TO 1 STEP -1
30 PRINT M
40 NEXT M
```

When executed these statements will print the values 5, 4, 3, 2, and 1.

For positive step values, the loop is executed as long as the control variable is less than or equal to its final value. For negative step values the loop continues as long as the control variable is greater than or equal to its final value.

If the initial value is greater than the final value and a positive step is indicated, or if the initial value is less than the final value and a negative step value is indicated, the body of the loop is not executed.

The FOR and NEXT statements can be written in a multiple statement line, provided: FOR is the first statement on its line, and NEXT is the last statement on its line.

3.5.2 Control Variable Value

The initial and final values of the control variable are computed only once — upon initial entry to the FOR loop. The control variable value can be modified within the loop.

Upon exit from the loop, the control variable retains the last value used within the loop, as shown in the following program.

```
 5 REM - TEST FOR CONTROL VARIABLE VALUE
 6 REM - AT EXIT FROM "FOR" LOOP
10 FOR A=1 TO 8 STEP 2
20 PRINT A;
30 NEXT A
33 REM - NEXT STATEMENT IS OUTSIDE THE LOOP.
34 REM - IT PRINTS FINAL COMPUTED VALUE OF
35 REM - CONTROL VARIABLE "A".
40 PRINT : PRINT A

RUN
 1  3  5  7
 7

STOP AT LINE   40
READY
```

When the variable is tested with the PRINT statement at line 40, outside of the loop, it is found to have a value of 7. The program above stopped at line 40, as is indicated by the message STOP AT LINE 40.

3.5.3 Nested FOR Statements

FOR statements can be nested to allow the programming of loops within loops, as shown below.

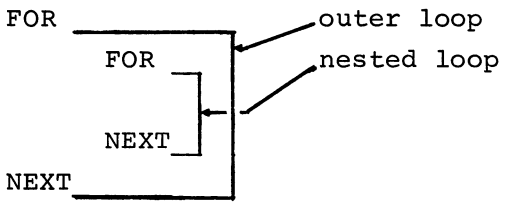
```
10 FOR A=1 TO 5
20 FOR B=2 TO 10 STEP 2
30 LET X=A + B
40 NEXT B
50 NEXT A
55 PRINT X

RUN
 15

STOP AT LINE   55
READY
```

Lines 20 to 40 are executed 25 times -- five times for each value of A in line 10.

Correct nesting of FOR loops is diagrammed below.



All statements of a nested loop are executed after the FOR statement of the outer loop and before the NEXT statement of the outer loop is encountered.

Additional levels of nesting are possible. However, deep nesting is costly in terms of working storage (see Appendix A for specific data on storage requirements). For a practical example of nested loops, see Section 3.12.3.

3.6 DATA and READ Statements

In previous examples, LET statements have been used to assign constant values to single variables; if more variables were needed, more LET statements were included. However, in programs requiring many variables and constant values, READ and DATA statements should be used.

The DATA statement introduces a numeric constant, or a series of constants, into a program. READ associates variable names sequentially with the constant values supplied by DATA statements.

READ and DATA statements must accompany one another in user programs, but they need not be paired. If nine variables appear in one or more READ statements, there must be at least nine constants in one or more DATA statements (see Section 3.7 for the exception).

In the following example, all data is introduced in a single DATA statement. It is used at separate points in the program by two READ statements.

```
10 DATA 1,5,3,7,9
20 READ A,B
30 LET X = A+B
40 PRINT A,B,X
50 READ V,Q,R
60 LET X = X+V+Q+R
70 PRINT
80 PRINT X,V,Q,R

RUN
 1             5             6

 25            3             7             9

STOP AT LINE   80
READY
```

In executing the above program BASIC ignores the DATA statement until it encounters a READ statement. It then goes back to the lowest numbered statement line of the program to search for a DATA statement. Above, it finds one at line 10, the first line of the program.

Taking constant values sequentially, it associates them with variables in the READ statement, also taken sequentially: A is assigned a value of 1, and B of 5. Establishing a pointer at the next data element, 3, it reverts to line 30, the next unexecuted statement.

In line 30 a new variable, X, is introduced with a LET statement and given a computed value of 6.

At line 50 another READ statement is encountered containing three new variables. This time BASIC does not search for the DATA statement but refers to its pointer to obtain the next unused data element. Variables V, Q, and R are assigned the constant values 3, 7, and 9.

In line 60 a computation is performed and in line 70 the latest computed value of X and the assigned values of V, Q, and R are printed.

(Note that X in line 60 retains its computed value from line 30. Line 60 is evaluated as X=6+3+7+9.)

Several error messages are associated with DATA and READ statements; see error codes 19, 20, 21, and 123 in Chapter 6. With regard to 123, each variable in a program must be in a READ statement, in a LET statement to the left of the = symbol, or in a FOR statement prior to use in an expression or PRINT list.

The DATA statement cannot be included in a multiple statement line; it must be the only statement on a numbered line.

The READ statement may be placed anywhere on a multiple statement line.

3.7 RESTORE STATEMENT

The RESTORE statement makes it possible to recycle through DATA statements beginning with the lowest numbered DATA statement in a program. For example:

```
40 DATA 1,2
50 READ A,B
60 PRINT A,B
70 RESTORE
80 READ C,D
90 PRINT C,D
RUN
 1             2
 1             2

STOP AT LINE   90
READY
```

The RESTORE statement at line 70 allows the READ statement at line 80 to obtain values from the DATA statement, even though the same values were used previously in the READ statement at line 50. Without the RESTORE statement an error message, indicating lack of data for the READ statement, would have occurred at line 80.

In the following program, the RESTORE statement causes the second READ statement, at line 60, to take constant values from the first DATA statement, at line 10, rather than from the second, at line 50.

```
10 DATA 1,2
20 READ A,B
30 PRINT A,B
40 RESTORE
50 DATA 3,4
60 READ C,D
70 PRINT C,D
RUN
 1             2
 1             2

STOP AT LINE   70
READY
```

The RESTORE statement may be used anywhere in the program -- on a line by itself or anywhere in a multiple statement line.

RESTORE has no effect in programs without DATA and READ statements.

3.8 INPUT STATEMENT

This statement is used to enter data from the keyboard while the program is running. The data is typed in as BASIC asks for it. For example:

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

will cause BASIC to pause during execution, print a question mark, and wait for the user (you) to type three numerical values. The constant values must be separated by commas and terminated with the RETURN key. Only one question mark is printed for each INPUT statement.

If enough values are not supplied, BASIC will print:

```
ERROR 121 AT LINE nnn
```

If too many values are supplied, BASIC will print:

```
ERROR 122 AT LINE nnn
```

where nnn is the line number of the INPUT statement. In either case, BASIC will print another question mark and wait for the requested input.

The INPUT statement may be used anywhere in the program -- on a line by itself or anywhere in a multiple statement line.

3.9 GOSUB AND RETURN STATEMENTS

It is often desirable to write a group of statements (a subroutine) only once in a program and then branch to the subroutine repeatedly from various points in the program. The last statement of the subroutine should be a RETURN statement. The GOSUB statement is used to branch (jump) to the first statement of the subroutine.

The RETURN statement must be used in conjunction with GOSUB. However, a single RETURN statement will suffice for a single subroutine branched to by more than one GOSUB, as illustrated below. The subroutine in lines 75 through 78 is entered from the GOSUB's in lines 20, 40, and 55, and exited by the RETURN in line 78.

```
 5 PRINT "X", "X↑2", "X↑3", "X↑4", "X↑5"
10 FOR A = 1 TO 5
15 LET X = A
20 GOSUB 75
25 NEXT A
28 FOR A = 1 TO 5
30 LET X = A↑2
40 GOSUB 75
45 NEXT A
48 FOR A = 1 TO 5
50 LET X = A↑3
55 GOSUB 75
60 NEXT A
65 STOP
75   FOR J = 1 TO 5
76   PRINT X↑J,
77   NEXT J
78   RETURN
80 END

RUN
X             X↑2           X↑3           X↑4           X↑5
 1             1             1             1             1
 2             4             8             16            32
 3             9             27            81            243
 4             16            64            256           1024
 5             25            125           625           3125
 1             1             1             1             1
 4             16            64            256           1024
 9             81            729           6561          59049
 16            256           4096          65536         1048576
 25            625           15625         390625        9765626
 1             1             1             1             1
 8             64            512           4096          32768
 27            729           19683         531441        .1434891E 8
 64            4096          262144        .1677722E 8   .1073742E 10
 125           15625         1953125       .2441406E 9   .3051758E 11

STOP AT LINE   65
READY
```

Both GOSUB and RETURN can be written in multiple statement lines, provided they are the final statements in their respective lines.

3.9.1 Nested Subroutines

Subroutines may contain calls to other subroutines. When written this way, the inner subroutine call is referred to as nested. Such ordering might appear as follows, where a GOSUB statement at line 20 calls a subroutine at line 100, which in turn calls a nested subroutine at lines 117 and 150. Note in the following example that one RETURN statement serves all three GOSUBs.

```
10 DATA 2,4
15 READ A,B
20 GOSUB 100
25 STOP
100 LET X=A
115 LET Y=B
117 GOSUB 180
130 LET X=X↑2
140 LET Y=(Y↑2)
150 GOSUB 180
160 LET X=A + 1/A
170 LET Y=B - 1/A
180 PRINT X,Y
182 RETURN

RUN
 2              4
 4              16
 2.5            3.5

STOP AT LINE   25
READY
```

Such nesting can be carried to any level. The only restriction is: for each nested subroutine entered a return address (line number) must be stored until the RETURN statement is executed. In systems with limited core space, deep nesting may reduce working storage below the requirement for the user program. (See Appendix A for storage requirements.)

An example of nested subroutine calls is illustrated in Figure 3-1.

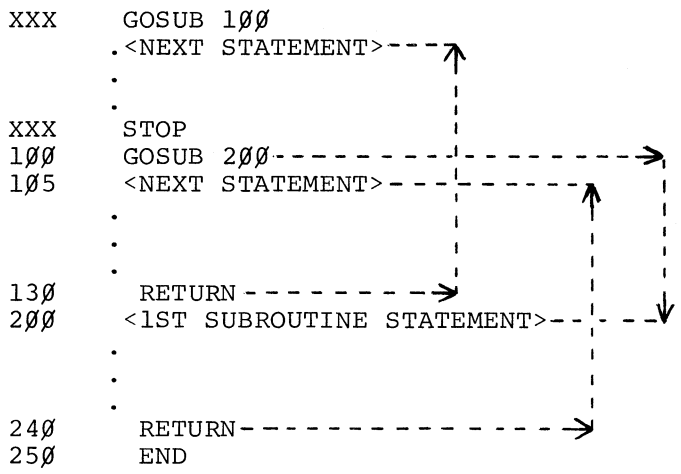


Figure 3-1. Nested Subroutine Call

In Figure 3-1, above, from line 100 until line 240 two return addresses are stored.

3.10 UNCONDITIONAL BRANCH, GOTO STATEMENT

The GOTO statement causes an immediate jump to an indicated line number. Program execution continues sequentially again beginning with the statement jumped to, as shown in the following example:

```
10 DATA 1,2,3,4,5
15 READ X
20 PRINT X+1
40 GOTO 15

RUN
 2
 3
 4
 5
 6

ERROR    20 AT LINE   15
READY
```

The general format of the statement is :

```
1Ø  GOTO <line number>
```

Statements 15, 20, and 40 in the above example constitute a loop. These three statements are re-executed continuously with a new value of X each time line 15 is executed, until the last constant value previously introduced at line 10 is used.

Program execution ends with an error message, indicating that all data values have been used. This error message does not indicate that the program printout is inaccurate (error messages are explained in Chapter 6).

In the following example, the GOTO statement provides a jump to an INPUT statement so that the program loops continually.

```
10 INPUT A
20 PRINT A,
30 LET A=A↑A
40 PRINT A
50 PRINT
60 GOTO 10
```

If written on a multiple statement line, GOTO must be the last statement on the line.

3.11 CONDITIONAL BRANCH, IF STATEMENT

The IF statement supplies a conditional program branch. The general format of the statement is:

```
45  IF <expression> <operator> <expression> [THEN <statement>  ]
                                            [THEN <line number>]
                                            [GOTO <line number>]
```

NOTE

The IF-THEN <statement> sequence is a special feature of PDP-11 BASIC.

THEN can be followed by any statement including another IF statement. If a line number follows THEN, the IF-THEN statement operates in the same manner as IF-GOTO.

When BASIC encounters the IF statement, it evaluates the expressions and compares them according to the requirements of the logical operator, shown in Table 3-1. If the test condition is met, the remainder of the statement beginning with THEN or GOTO is executed. If the test condition is not met, the next sequentially numbered statement after the IF statement is executed.

In the following example, a FOR-NEXT loop is in lines 10 to 35; the range of the control variable is 1 to 10. The IF-GOTO statement is used to limit the range of the variable contained in the FOR-NEXT loop. Operation of the loop continues until the relationship A>4 is true, then immediately branches to line 55.

TABLE 3-1. Logical Operators

Standard Symbol	BASIC Version	Example	Meaning
=	=	A=B	A equal to B
<	<	A<B	A less than B
≤	<=	A<=B	A less than or equal to B
>	>	A>B	A greater than B
≥	>=	A>=B	A greater than or equal to B
≠	<>	A<>B	A not equal to B

```
10 FOR A=1 TO 10
20 LET X=A↑2
25 IF A>4 GOTO 55
30 PRINT X
35 NEXT A
40 PRINT "VALUE OF A IS " A
55 STOP

RUN
 1
 4
 9
 16

STOP AT LINE   55
READY
```

If line 10 is changed to

```
10 FOR A=1 TO 3
```

the conditional statement at line 30 will have no effect on the program. The loop will end normally when A reaches a value of 3 and program execution will continue with the PRINT statement at line 40.

In the following example, THEN is followed by a line number at line 45 and by a statement at lines 40 and 50.

At line 40, if X is less then or equal to 70, BASIC goes to line 45 and the PRINT statement at line 40 is not executed.

At line 45, if X is less then or equal to 70, BASIC goes to line 50 and the jump (or branch) to line 20 is not executed.

At line 50, if X is greater than 70, BASIC goes to line 55 and the PRINT statement at line 50 is not executed.

```
10 REM - PROGRAM TO ASSESS GRADES
20 PRINT
25 INPUT A,B,C,D
30 LET X = (A+B+C+D)/4
35 PRINT: PRINT
40 IF X>=70 THEN PRINT X; "SATISFACTORY GRADE"
45 IF X>=70 THEN 20
50 IF X<70 THEN PRINT "FAILED THIS COURSE"
55 GOTO 20

RUN

?55,67,78,89


 72.25 SATISFACTORY GRADE

?55,44,32,21


FAILED THIS COURSE

?
```

The IF statement can be used anywhere in a multiple statement line except when a GOSUB or GOTO statement follows THEN. In this case, the restriction on GOSUB applies, i.e., the IF statement must be the last statement on the line.

A THEN-GOSUB statement provides a conditional jump to a subroutine. GOSUB must be the last statement on the line. Completion of the subroutine returns control to the statement on the line following the IF statement (the next line in numerical sequence). An extension of our previous example will illustrate this statement. (Note the conditional GOSUB in line 80.)

```
10 REM - PROGRAM TO ASSESS STUDENT GRADES
15 PRINT
20 PRINT "STUDENT NO -";
30 INPUT A
40 PRINT "ENTER QUARTERLY GRADES -";
50 INPUT B,C,D,E
60 LET X = (B+C+D+E)/4
70 IF X<70 THEN PRINT "FAILED COURSE -"; X
80 IF X>=70 THEN GOSUB 100
90 GOTO 15
100 IF X>93 THEN PRINT "HONORS -"; X
110 IF X<=95 THEN PRINT "PASSING GRADE -"; X
120 RETURN
```

```
RUN

STUDENT NO -?1
ENTER QUARTERLY GRADES -?45, 55, 60, 40
FAILED COURSE - 50Ø

STUDENT NO -?2
ENTER QUARTERLY GRADES -? 98, 97, 92, 100
HONORS - 96.75

STUDENT NO -?3
ENTER QUARTERLY GRADES -? 65, 75, 80, 77
PASSING GRADE - 74.25

STUDENT NO -?↑P              (CTRL/P was typed here)
READY
```

The IF statement at line 70 allows the PRINT statement on the same line to be executed only if the value of X is less than 70. The IF statement at line 80 causes a jump to the subroutine at lines 100 through 120 if the value of X is greater than or equal to 70.

The RETURN at line 120 is to line 90 which, in turn, causes a jump to line 15 to keep the program looping.

3.12 SUBSCRIPTED VARIABLES

BASIC allows the use of subscripted variables for handling data arrays. Any variable name formulated according to the rules in Section 2.5.2 can be given a series of values: the variable names reference the specific array; the subscripts reference a particular data item in the array. Typical subscripted or array variables are:

A(Ø), A(1), A(2), A(3)...A(n)

when used to reference a one-dimensional array, or list, and

A7(Ø,Ø), A7(Ø,1), A7(1,Ø), A7(1,1), ..., A7(n,n)

when used to reference the data items of the two-dimensional array A7 (BASIC is limited to arrays of two dimensions).

NOTE

Array names consisting of a letter followed by a number are legal; this is a special feature of PDP-11 BASIC.

NOTE

Subscripts in PDP-11 BASIC begin with (0) for a one-dimensional array and (0,0) for a two-dimensional array. For these initial items the subscript need not be coded with the array name. A(0,0), A(0), and A are the same variable.

3.12.1 The DIMension Statement

Prior to creating a data array of any size, a DIM statement must be executed to reserve storage space. The statement must contain the array name and maximum potential subscript:

```
20 DIM A1(4,4)
```

Because subscripts in PDP-11 BASIC begin with (0) or (0,0), the statement above will reserve space for a 5 x 5 array, or one capable of containing 25 discrete items.

Dimensioning a previously used variable is illegal. For example, the following will produce an error message.

```
10 LET A3 = C↑C + B/3
    .
    .
    .
40 DIM A3(2,5)
```

The greatest possible subscript for any array variable is (255) for a variable representing a list, or (255,255) for a variable representing a two-dimensional array. Appendix A, Section A.3 provides data for computing potential array size for a given amount of core.

3.12.2 Generating Arrays

Array variables have special relationships which simplify operations on multiple data items. A list can be formulated, and unique subscripts generated at printout, as follows. (Note that lines 10 to 40 in the example build the array and fill it with zeroes to prepare it for data entry.)

```
10 DIM A(19)
20 FOR I=0 TO 19
30 LET A(I)=0
40 NEXT I
45 LET I=0
50 INPUT X
60 LET A(I)=X
70 LET I=I + 1
80 GOTO 50
RUN
?5
?6
?7
?8
?9
?↑P                          (CTRL/P was typed here)
READY
PRINT A
 5
PRINT A(1)
 6
PRINT A(2)
 7
PRINT A(3)
 8
PRINT A(4)
 9
PRINT A(5)
 0
PRINT A(6)
 0
PRINT A(29)

ERROR      6 AT LINE     0
READY
```

Above, execution of the program is stopped with CTRL/P (see Section 5.5) after the introduction of five constants. PRINT is then used in immediate mode to examine the initial array elements. An attempt to print the contents of an array location beyond the limit set in the DIM statement produces the error message in the next to last line of the example program.

The previous program is continued below to demonstrate how an array can be printed out automatically.

```
85 REM - LINES 90 THRU 120 PRINT THE LIST SEQUENTIALLY
90 FOR I=0 TO 19
100 PRINT "A("I")", A(I)
110 NEXT I
120 STOP
125 REM - LINES 130 THRU 150 PRINT A PORTION OF THE
126 REM - LIST IN REVERSE ORDER.
130 FOR I=6 TO 0 STEP -1
135 PRINT "A("I")", A(I)
140 IF I=0 THEN STOP
150 NEXT I
```

```
RUN
?1
?2
?3
?4
?5
?↑P                                          (CTRL/P was typed here)
READY
GOTO 90
A( Ø )          1
A( 1 )          2
A( 2 )          3
A( 3 )          4
A( 4 )          5
A( 5 )          Ø
A( 6 )          Ø
A( 7 ↑P)        Ø                            (CTRL/P was typed here)

READY
GOTO 130
A( 6 )          Ø
A( 5 )          Ø
A( 4 )          5
A( 3 )          4
A( 2 )          3
A( 1 )          2
A( Ø )          1

STOP AT LINE  14Ø
READY
```

3.12.3 Two-Dimensional Arrays

The two-dimensional array is logically represented as a table of two or more rows and columns. A 3 x 3 array, designated M, is represented below:

rows \ columns	Ø	1	2
Ø	M(Ø,Ø)	M(Ø,1)	M(Ø,2)
1	M(1,Ø)	M(1,1)	M(Ø,2)
2	M(2,Ø)	M(2,1)	M(2,2)

Note that the initial constant in the subscript represents the row designation.

In the following example program, space is reserved for a two-dimensional array (line 10) and it is zero-filled (lines 20 - 45). In lines 20 and 25, the order of subscripts is optional: it does not matter whether the array is zero-filled starting with the first or last array item. However, if the order is reversed:

```
20 FOR J=Ø TO 4
25 FOR I=Ø TO 4
```

then the order in lines 40 and 45 must be reversed:

```
40 NEXT I
45 NEXT J
```

This is to conform to the rule for nested FOR-NEXT loops: the range of the nested loop must lie entirely within the range of the outer loop.

```
10 DIM A(4,4)
20 FOR I=Ø TO 4
25 FOR J=Ø TO 4
30 LET A(I,J)=Ø
40 NEXT J
45 NEXT I
50 FOR I=Ø TO 4
55 FOR J=Ø TO 4
60 INPUT X
70 LET A(I,J)=X
80 NEXT J
85 NEXT I
100 FOR I=Ø TO 4
105 FOR J=Ø TO 4
110 PRINT "A("I","J")", A(I,J)
120 NEXT J
125 NEXT I

RUN
?1
?2
?3
?4
?5
?6
?6
?↑P                                (CTRL/P was typed here)
READY
GOTO 100
A( Ø , Ø )      1
A( Ø , 1 )      2
A( Ø , 2 )      3
A( Ø , 3 )      4
A( Ø , 4 )      5
A( 1 , Ø )      6
A( 1 , 1 )      6
A( 1 , 2 )      Ø
A( 1 , 3 )      Ø
A( 1 , 4↑P )    Ø                  (CTRL/P was typed here)

READY
```

Lines 50 - 85 provide for data input from the keyboard, and limit inputs to the number of data spaces available in the array.

Lines 100 - 125 provide for printout and labelling of the array data items. The order of subscripts in lines 100 - 125 must correspond with the order in lines 50 - 85 if the data is to be printed out and labelled in the same order in which it was entered.

The question marks following RUN were printed by the INPUT statement; the integers were typed in by the user. (The entire array was not filled.)

Program printout appears after the immediate mode GOTO 100 statement. Since only seven data items were entered, the program began to print zeroes after the seventh array item.

CHAPTER 4

MATHEMATICAL FUNCTIONS

4.1 INTRODUCTION

BASIC contains ten functions to perform mathematical operations. They relieve the user from programming his own routines to calculate such things as square roots, logarithms, etc.

These functions have a three-letter call name followed by a parenthesized argument. They are pre-defined and may be used anywhere in a program.

Call Name	Function
SIN(x)	Sine of x, where x is expressed in radians
COS(x)	Cosine of x, where x is expressed in radians
ATN(x)	Arctangent of x is returned as an angle in radians in range $\pm\pi/2$
SQR(x)	Square root of x
EXP(x)	Exponential of x
LOG(x)	Natural logarithm of x ($LOG_e(x)$)
ABS(x)	Absolute value of x
INT(x)	Truncate fraction part of x (truncates to the largest integer not greater than x)
RND(x)	Generate random number between Ø and 1
SGN(x)	Return a value indicating the sign of x

The argument x to the functions can be a constant, a variable, an expression, or another function.

The first four functions (SIN, COS, ATN, SQR), termed extended functions, may be deleted when loading BASIC (as explained in Chapter 7). Deleting the extended functions provides an extra 300 words of storage for user programs.

If floating-point exponentiation is not needed, the EXP and LOG functions may be deleted (see Chapter 7), providing another 250 words of user storage. When these functions are deleted, all exponentiation will be done by repetitive multiplications, result-

ing in slower processing time but providing more storage space for user programs.

4.2 USAGE

Function calls, consisting of the function name followed by a parenthesized argument, can be used as expressions or as elements of expressions anywhere that expressions are legal.

The functions SIN(X), COS(X), ATN(X), SQR(X), EXP(X), and LOG(X) produce a value accurate to ± 1 in the 10^{-8} position.

4.2.1 Sine and Cosine Functions, SIN(X) and COS(X)

The sine and cosine functions require an argument angle expressed in radian measure. If the angle is stated in degrees, conversion to radians may be done using the identity:

$$\text{<radians>} = \text{<degrees>} \quad (\frac{\pi}{180})$$

In the following example program, 3.14159265 is used as a nominal value for π. P is set equal to this value at line 20. At line 40 the above relationship is used (in the expression within the LET statement) to convert the input value into radians.

```
10 REM - CONVERT ANGLE (X) TO RADIANS, AND
11 REM - FIND SIN AND COS
20 LET P = 3.14159265
25 PRINT "DEGREES", "RADIANS", "SINE", "COSINE"
30 INPUT X
40 LET Y = X*P/180
60 PRINT X, Y, SIN(Y), COS(Y)
70 GOTO 30

RUN
DEGREES        RADIANS        SINE           COSINE
?0
 0              0              0              1
?10
 10             .1745329       .1736482       .9848078
?20
 20             .3490658       .3420201       .9396926
?30
 30             .5235988       .5             .8660254
?360
 360            6.283185      -.2925836E-8    1
?45
 45             .7853982       .7071068       .7071068
?90
 90             1.570796       1              .1462918E-8
?↑P
READY
```

4.2.2 Arctangent Function, ATN(X)

The arctangent function returns a value in radian measure, in the range $+\frac{\pi}{2}$ to $-\frac{\pi}{2}$ corresponding to the value of a tangent supplied as the argument (X).

In the following program, input is an angle in degrees. Degrees are then converted to radians at line 40. At line 50 the radian value (Y) is used with the SIN and COS functions to derive the tangent of the input angle according to the identity:

$$TAN(X) = \frac{SIN(X)}{COS(X)}$$

At line 70 the tangent value, Z, is supplied as argument to the ATN function to derive the value found in column 4 of the printout under the label ATN(X). Also in line 70 the radian value of the arctangent function is converted back to degrees and printed in the fifth column of the printout as a check against the input value shown in the first column.

```
10 LET P = 3.14159265
20 PRINT "SUPPY AN ANGLE IN DEGREES"
25 PRINT "ANGLE","ANGLE","TAN(X)","ATAN(X)","ATAN(X)"
26 PRINT "(DEGS)","(RADS)",,,"(DEGS)"
30 INPUT X
40 LET Y = X*P/180
50 LET Z = SIN(Y)/COS(Y)
70 PRINT X,Y,Z,ATN(Z),ATN(Z)*180/P
85 PRINT
90 GOTO 30

RUN
SUPPY AN ANGLE IN DEGREES
ANGLE          ANGLE          TAN(X)          ATAN(X)         ATAN(X)
(DEGS)         (RADS)                                         (DEGS)
?0
 0              0              0               0               0

?45
 45             .7853982       1               .7853982        45

?90
 90             1.570796       .6835653E 9     1.570796        90

?↑P
READY
```

4.2.3 Square Root Function, SQR(X)

This function derives the square root of any positive value as shown below.

```
10 INPUT X
20 LET X = SQR(X)
30 PRINT X
40 GOTO 10

RUN
?16
 4
?100
 10
?1000
 31.62278
?123456789
 11111.11
?17
 4.123106
?25E2
 50
?1970
 44.38468
?↑P
READY
```

4.2.4 Exponential Function, EXP(X)

The exponential function raises the number e to the power x. EXP is the inverse of the LOG function. The relationship is

```
LOG(EXP(X)) = X
```

The following program prints the exponential equivalent of an input value. Note that the output values derived below are used as input to the LOG function in Section 4.2.5.

```
10 INPUT X
20 PRINT EXP(X)
40 GOTO 10

RUN
? 4
 54.59815
? 10
 22026.47
? 9.421006
 12344.99
? 4.60517
 99.99998
? 25
 .720049E 11
? ↑P
READY
```

4.2.5 Logarithm Function, LOG(X)

The LOG function derives the logarithm to the base e of a given value. In the following program at line 20, the LOG function is used to convert an input value to its logarithmic equivalent.

```
10 INPUT X
20 PRINT LOG(X)
30 GOTO 10

RUN
?54.59815
 4
?22026.47
 10
?12345
 9.421006
?100
 4.60517
?.720049E11
 25
?↑P
READY
```

Logarithms to the base e may easily be converted to any other base using the following formula:

$$\log_a N = \frac{\log_e N}{\log_e a}$$

where a represents the desired base. The following program illustrates conversion to the base 10.

```
 1 REM - CONVERT BASE E LOG TO BASE 10 LOG.
 5 PRINT "VALUE","BASE E LOG","BASE 10 LOG"
15 INPUT X
17 PRINT X,
20 PRINT LOG(X),
40 PRINT LOG(X)/LOG(10)
50 GOTO 15
60 END
READY
RUN
VALUE          BASE E LOG     BASE 10 LOG
?4
 4              1.386294       .60206
?250
 250            5.521461       2.39794
?5
 5              1.609438       .69897
?60
 60             4.094345       1.778151
?100
 100            4.60517        2
?↑P
READY
```

4.2.6 Absolute Function, ABS(X)

The ABS function returns an absolute value for any input value. Absolute value is always positive. In the following program, various input values are converted to their absolute values and printed.

```
10 INPUT X
20 LET X = ABS(X)
30 PRINT X
40 GOTO 10

RUN
?-35.7
 35.7
?2
 2
?25E10
 .25E 12
?105555567
 .1055556E 9
?10.12345
 10.12345
?-44.555566668899
 44.55557
?↑P
READY
```

4.2.7 Integer Function, INT(X)

The integer function returns the value of the greatest integer not greater than x. For example:

```
PRINT INT(34.67)
 34
```

```
PRINT INT(-5.1)
-6
```

The INT of a negative number is a negative number with the same or larger absolute value, i.e., the same or smaller algebraic value. For example:

```
PRINT INT(-23.45)
-24
PRINT INT(-14.39)
-15
```

The INT function can be used to round numbers to the nearest integer, using INT(X+.5). For example:

```
PRINT INT(34.67+.5)
 35
```

```
PRINT INT(-5.1+.5)
-5
```

INT(X) can be used to round to any given decimal place or integral power of 10, by using the following expression as an argument:

```
(X*10↑D+.5)/10↑D
```

where D is an integer supplied by the user.

```
10 REM - INT FUNCTION EXAMPLE.
15 PRINT
20 PRINT "NUMBER TO BE ROUNDED:"
25 INPUT A
40 PRINT "NO. OF DECIMAL PLACES:"
45 INPUT D
60 LET B = INT(A*10↑D + .5)/10↑D
70 PRINT "A ROUNDED = " B
80 GOTO 15
90 END

RUN

NUMBER TO BE ROUNDED:
?55.65842
NO. OF DECIMAL PLACES:
?2
A ROUNDED =  55.66

NUMBER TO BE ROUNDED:
?78.375
NO. OF DECIMAL PLACES:
?-2
A ROUNDED =  100

NUMBER TO BE ROUNDED:
?67.38
NO. OF DECIMAL PLACES:
?-1
A ROUNDED =  70

NUMBER TO BE ROUNDED:
?↑P
READY
```

4.2.8 Random Number Function, RND(X)

The random number function produces a pseudo-random number, or random number set, between 0 and 1. The numbers are reproducible in the same order for later checking of a program. The argument (x) is not used and can be any number; it serves only to standardize all BASIC function representations. For example:

```
10 REM - RANDOM NUMBER EXAMPLE.
25 PRINT "RANDOM NUMBERS:"
30 FOR I = 1 TO 15
40 PRINT RND(0),
50 NEXT I
60 END
RUN
RANDOM NUMBERS:
 .1002502       .9648132       .8866272       .6364441       .8390198
 .3061218       .285553        .9582214       .1793518       .4521179
 .9854126E-1    .5221863       .2462463       .7778015       .450592

STOP AT LINE   60
READY
```

To obtain random digits from 0 to 9, change line 40 to read

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

and run the program again. This time the results will be printed as follows.

```
40 PRINT INT(10*RND(0)),
RUN
RANDOM NUMBERS:
 1              9              8              6              8
 3              2              9              1              4
 0              5              2              7              4

STOP AT LINE   60
READY
```

It is possible to generate random numbers over a given range. If the open range (A,B) is desired, use the expression:

```
(B-A) * RND(0) + A
```

The following program produces a random number set in the open range 4,6 (the extremes, 4 and 6, are never reached).

```
10 REM - RANDOM NUMBER SET IN OPEN RANGE 4,6.
20 FOR B = 1 TO 15
30 LET A = (6-4) * RND(0) + 4
40 PRINT A,
50 NEXT B
60 END

RUN
 4.2005         5.929626       5.773254       5.272888       5.67804
 4.612244       4.571106       5.916443       4.358704       4.904236
 4.197083       5.044373       4.492493       5.555603       4.901184

STOP AT LINE   60
READY
```

4.2.9 RANDOMIZE Statement

The RANDOMIZE statement causes the random number generator to calculate different random numbers every time the program is run. When executed, RANDOMIZE causes the RND function to choose a random starting value to produce random results. For example:

```
10 REM - RANDOM NUMBERS USING RANDOMIZE.
15 RANDOMIZE
25 PRINT "RANDOMIZED NUMBERS:"
30 FOR I = 1 TO 4
40 PRINT RND(0),
50 NEXT I
60 END

RUN
RANDOMIZED NUMBERS:
 .7785034E-1    .1632385       .2787781       .2035217
STOP AT LINE    60
READY
RUN
RANDOMIZED NUMBERS:
 .8417053       .1678467E-2    .4347229       .5932312
STOP AT LINE    60
READY
RUN
RANDOMIZED NUMBERS:
 .6651917       .2846375       .7210999       .7648621
STOP AT LINE    60
READY
```

Removing the RANDOMIZE statement and changing line 25:

```
DELETE 15
READY
25 PRINT "REPRODUCIBLE RANDOM NUMBER SET."
```

program output is as follows.

```
RUN
REPRODUCIBLE RANDOM NUMBER SET.
 .1002502       .9648132       .8866272       .6364441
STOP AT LINE    60
READY
RUN
REPRODUCIBLE RANDOM NUMBER SET.
 .1002502       .9648132       .8866272       .6364441
STOP AT LINE    60
READY
RUN
REPRODUCIBLE RANDOM NUMBER SET.
 .1002502       .9648132       .8866272       .6364441
STOP AT LINE    60
READY
```

4.2.10 Sign Function, SGN(X)

The sign function returns the value 1 if x is a positive value, 0 if x is 0, and -1 if x is negative. For example:

```
PRINT SGN(3.42)
 1

PRINT SGN(-42)
-1

PRINT SGN(23-23)
 Ø
```

The following example program illustrates the use of the SGN function.

```
1Ø REM - SGN FUNCTION EXAMPLE.
2Ø READ A,B,C
25 PRINT "A  = "A, "B  = "B, "C  = "C
3Ø PRINT "SGN(A) ="SGN(A), "SGN(B) ="SGN(B),
4Ø PRINT "SGN(C) ="SGN(C)
5Ø DATA -7.32, .44, Ø
6Ø END
READY
RUN
A  = -7.32    B  =  .44     C  =  Ø
SGN(A) =-1    SGN(B) = 1    SGN(C) = Ø

STOP AT LINE   6Ø
READY
```

4.3 USER-DEFINED FUNCTIONS

In some programs it may be necessary to use the same mathematical expression in several places, often using different data. BASIC's user-defined function enables you to define unique operations or expressions and call them as you would the standard mathematical functions, i.e., sine, cosine, square root, etc.

The user-defined function is identified by a three-letter call name followed by a parenthesized argument. The first two letters of the function name are FN and the third letter may be any letter in the alphabet; thus, as many as 26 unique user-defined functions are possible. The parenthesized argument is explained below.

The user-defined function must be defined before it can be used. It is defined using the DEF statement; the general format is:

```
DEF FNa(variable) = expression
```

where "a" can be any letter. For example,

```
10 DEF FNX(S) = S↑2+4
```

would cause the statement

```
20 LET R = FNX(4)
```

to be evaluated as LET R=4↑2+4, or

```
20 LET R = FNX(2)
```

to be evaluated as LET R=2↑2+4. The function name, FNX(S), represents the expression, S↑2+4, in the DEF statement, and when called, the parenthesized argument, (4) or (2), is equated with the function variable, (s).

When called, the parenthesized argument may be any legal expression; the value of the expression is substituted for the function variable. In the following example, the function FNZ at line 10 will square whatever is substituted for the function variable. At line 30 the expression 2+A evaluates to 4, which is then squared to give a printed result of 16.

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

RUN
 16

STOP AT LINE   30
READY
```

Here's another example:

```
10 DEF FNA(Z) = Z+A+B
20 DATA 2,4
30 READ A,B
40 LET F = FNA(9) + 1
50 PRINT F

RUN
 16

STOP AT LINE   50
READY
```

The function name may be used recursively, as in the following example. The expression in line 30 is evaluated as (2+(2*4)) before being squared in line 10.

```
10 DEF FNA(X) = X↑2
20 LET A = 2
30 PRINT FNA(2+A*FNA(2))

RUN
 100

STOP AT LINE    30
READY
```

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

```
10 DEF FNX(X) = X↑2
20 DEF FNX(X) = X+X
30 PRINT FNX(6)
40 END

RUN
 36

STOP AT LINE    40
READY
```

The function variable need not appear in the function expression; later substitution of an expression for the variable will have no affect, as illustrated below.

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

RUN
 7

STOP AT LINE    30
READY
```

CHAPTER 5

BASIC COMMANDS

BASIC's commands are used to perform various operations such as executing a program, stopping program execution, punching a program on paper tape, loading a program from paper tape, etc.

5.1 LIST

To obtain a clean copy of an edited program, type LIST followed by the RETURN key. The program will be printed on the Teletype and BASIC will again print READY.

To print a single line, type its line number after LIST. To print a series of lines, follow LIST with the initial and ending line numbers; these lines plus all lines between them will be printed. For examples:

```
LIST 37
```

prints line 37;

```
LIST 37,126
```

prints lines 37 through 126;

```
LIST
```

prints the whole program.

5.2 DELETE

The DELETE command is used to erase lines from the program. The DELETE command requires an argument. This prevents accidental deletion of the entire program.

If only one line number is given after the DELETE command, that one line will be deleted. If two line numbers separated by a comma are given, those two lines and all lines between them will be deleted. For example:

```
DELETE 54
```

will erase only line 54;

```
DELETE 36,876
```

will erase lines 36 through 876;

```
DELETE 1,8191
```

will erase the entire user program storage area.

NOTE

If, for example, DELETE 10,5 is typed where the initial line number is greater than the second, BASIC will respond with READY, but no lines will be deleted.

5.3 SAVE

SAVE will punch the user's BASIC program on the low-speed or high-speed punch, depending on your response to BASIC's initial dialogue at load time (see Chapter 7).

When using the high-speed paper tape punch, turn the punch on before issuing the SAVE command. When using the low-speed punch, type SAVE, turn the punch on, and then type the RETURN key.

NOTE

When BASIC finishes punching the saved program it punches 64 frames of trailer tape and then types READY. If the low-speed punch is used, READY is also punched into the tape and will cause errors when the tape is read. (READY will be interpreted as an immediate READ Y). To avoid this problem the user must:

a. Turn the punch off while trailer tape is still being punched; or

b. If punching is complete, tear the tape within the trailer section, eliminating the extraneous code which appears at the end.

5.4 OLD

The OLD command is used to read in user programs which were previously saved on paper tape. Either the low-speed or high-speed reader may be used, depending on system configuration and your response to BASIC's initial dialogue (see Section 7.1).

To read in a paper tape, place the tape in the appropriate reader and then type the OLD command followed by the RETURN key. After the tape stops, BASIC prints READY.

NOTE

When the low-speed reader is used, CTRL/P must be typed with the tape stops; BASIC then prints READY.

As the tape is read in, BASIC scans each line. Any of several fatal errors may be detected as the saved program is scanned. These will cause printout of an error message and termination of the read-in.

In executing the OLD command, BASIC automatically clears core before starting the reader. Any programs in core will be lost, with the exception of EXF function programs (see Chapter 8).

Once in core the saved program can be edited, added to, or executed in the same manner as a newly-created program.

5.5 STOPPING A RUN, CTRL/P

Typing CTRL/P causes BASIC to stop execution at the end of its current task or statement, print ↑P, and then READY. There may be a delay from the time CTRL/P is typed until BASIC prints READY. If, for example, CTRL/P is typed while a PRINT statement is executing, BASIC will complete the PRINT statement before typing READY.

CTRL/P has no affect on variable values. Therefore, after typing CTRL/P, variable values may be examined or used in PRINT statements.

5.6 RUN

The RUN command causes deferred mode programs to begin execution starting at their lowest numbered line. When RUN is used to re-run a program previously executed, all variable values are deleted from memory. If the RND(X) function is included within the program, it is reset to its initial starting value.

5.7 COMMANDS IN USER PROGRAMS[1]

It is possible to include the BASIC commands RUN, LIST, DELETE, SAVE, and OLD, preceded by a line number, in user programs.

---

[1]This usage is a special feature of PDP-11 BASIC.

When the command is encountered, the requested operation will be performed.

When RUN is executed in a program, it causes the program to restart. This can be useful to obtain repeated results from a program.

When the LIST, DELETE, SAVE, and OLD commands are used in a program, BASIC exits from the program and prints READY upon completion of the commanded operation.

CHAPTER 6

ERROR MESSAGES

6.1 FORMAT

BASIC checks program statements and input for errors, and prints an appropriate error message for each error detected. Messages are printed in the following format:

```
ERROR xxx AT LINE yyy
```

where xxx is the error code, as described below, and yyy is the number of the line in which the error occurred. If yyy appears as 0, it indicates that the error was made while in command mode, that is, the error was made at the last line typed in.

Error codes from 0 to 64 indicate a fatal error -- program execution will halt with the printing of the error message. Codes from 65 to 127 indicate a non-fatal error -- the program may continue to run after the error message is printed.

6.1.1 Fatal Errors

Error Code	Meaning
Ø	User storage overflow (see Appendix A, Sec. A.6)
1	Unrecognizable statement
2	Illegal GOTO or GOSUB
3	Illegal character terminating a statement (usually caused by an ill-formed statement which causes the statement operation to end prematurely).
4	RETURN without corresponding GOSUB
5	Badly formed subscript
6	Subscript not in range Ø to 255 or exceeds maximum set by program
7	Mismatched parentheses
8	Illegal LET
9	Illegal relational operator in IF
1Ø	Illegal IF
11	Illegal PRINT
12	Input line too long (exceeds 80 characters)
13	Bad dimension in DIM statement
14	Not enough storage for the array
15	Badly formed DEF statement
16	Illegal line number or dimension value
17	DIM of previously declared or used item. It is illegal to dimension any item which has been previously used in any way.
18	Bad variable in INPUT list
19	Bad variable in READ list

Error Message	Meaning
2Ø	Out of data in READ list
21	Bad DATA statement format
22	Illegal FOR statement
23	No NEXT matching FOR
24	NEXT without FOR
25	Unmatched quotes in statement
26	EXP function improperly set up (bad linkage in location 50)
27	Ill-formed expression (probably missing exponent on E format number)

6.1.2 Non-Fatal Errors

Error Code	Meaning
12Ø	Illegal characters on input
121	Not enough data typed to INPUT
122	Too much data typed to input
123	Non-existent variable
124	Number too large to fix (probably a subscript combination out of range)
125	Divide/multiply overflow or underflow
126	Square root of negative number
127	Logarithm of negative or zero number or exponential overflow

CHAPTER 7

LOADING AND STARTING BASIC

7.1 INITIAL DIALOGUE

BASIC is supplied as a single binary paper tape and is loaded using the Absolute Loader (see Appendix D). Upon completion of loading, BASIC will type its name and version number followed by:

```
*O
```

which stands for option. The user may respond with parameters from the following list in any order separated by commas and terminated by typing the RETURN key. These parameters are:

L	Use the low-speed reader/punch for the SAVE and OLD commands instead of the high-speed reader/punch. If there is no high-speed reader/punch, the low-speed reader/punch will be automatically selected.
D	Delete the extended functions, i.e., SIN, COS, ATN, SQR.
E	Delete EXP and LOG, as well as the extended functions.
H	Halt before entering the interpreter to allow loading of the EXF function (see Chapter 8).
X	Where X may be any integer between 4 and 28 to override automatic assignment of memory. The automatic process assigns to BASIC all memory available in the processor. This command allows the user to force BASIC to use less than the maximum configuration and is stated in increments of 1K.

If a user types only the RETURN key in response to *O, the options are as follows:

1. If a high-speed reader exists on the system it will be utilized.
2. All extended functions will be retained.
3. BASIC will utilize all of available memory for the user storage.
4. BASIC will not halt to allow loading an EXF function.

7.2 LONG FORM OF DIALOGUE

If a user does not understand the allowable options, he can type a question mark and the RETURN key to get the long form of the initialization. The long form will also result if his response to *O is in error. This will cause the following questions to be printed, which require individual YES or NO answers.

```
DO YOU NEED THE EXTENDED FUNCTIONS?
HIGH-SPEED READER/PUNCH?
SET UP THE EXTERNAL FUNCTION?
```

The final question shown below requires a numerical answer between 4 and 28, as described under X in Section 7.1 above, or a RETURN key if automatic assignment of memory is desired.

```
MEMORY?
```

After the response to all queries, BASIC types:

```
READY
```

7.3 RESTARTING BASIC

BASIC may be restarted by setting the ENABLE/HALT switch to HALT, placing 0 in the switch register, pressing LOAD ADDRess, setting ENABLE/HALT to ENABLE, and then pressing START.

All program text and data will be cleared in a restart.

7.4 LOADING THE EXF PROGRAM

If the H option is among those typed in response to BASIC's *O query, or if question 4 of the long form is answered YES, BASIC first attends to any other parameters that have been entered, then halts to allow loading of the assembler-coded program (see Chapter 8).

The binary program is loaded using the Absolute Loader, as described in Appendix D. Since the user program ends with a transfer address of 52 (see Chapter 8 for EXF program requirements), BASIC is signalled when the load is complete and prints:

```
READY
```

If the user program is on more than one tape, only the last may contain the transfer address.

CHAPTER 8

USING ASSEMBLY LANGUAGE PROGRAMS WITH BASIC

NOTE

The user must be familiar with the PAL-11A Assembler and assembly language as described in the PDP-11 Paper Tape Software Programming Handbook, DEC-11-GGPA-D, and the PDP-11 Handbook 1970.

8.1 DESCRIPTION

The BASIC function call, EXF, when written as any other function within an expression, serves to link the user's BASIC program and a PAL-11A assembly language program coded for compatibility with BASIC and resident in core.[1]

When BASIC encounters the EXF function call it passes control to the assembly language program, which can, in turn, "borrow" BASIC's internal routines, such as ITOA (Integer-TO-ASCII) or EVALuate.

The EXF function call can be used as an expression, or as an element of an expression, anywhere that an expression is legal in BASIC syntax:

```
10 LET X = EXF(X+5)
```

or

```
10 PRINT X, EXF(X)
```

8.1.1 Format of Function Call

The external function is called with its three-letter mnemonic followed by a parameter list in parentheses. For example, consider the following.

```
EXF(2)
EXF(SIN(X+5))
```

and

```
EXF(X↑2,XY,BC#@,A*Y)
```

[1] Once in core, the external program can only be deleted by reloading BASIC.

are all valid forms of the EXF function.

The EXF call is always to the same external program. However, the external program may have several entry points to perform differing functions. Designators for such entry points may be in the parameter list following the function call name anywhere except as the first parameter. The external program must recognize and utilize such entry point designators.

8.1.2 Evaluation

Before yielding control to the external program, BASIC evaluates the first item of the parameter list. The value is stored, and register R0 is set to point to its storage location.

R1 is set to point to the comma following the first argument, or, if there is only one argument, to the character following the final parenthesis (to facilitate return to the BASIC program when EXF has completed execution).

It is the user's responsibility to evaluate and use any argument after the first. However, if any of the remaining arguments are valid BASIC expressions, the BASIC subroutine EVAL can be called from the external program to evaluate them.

To illustrate, the sample EXF calls in Section 8.1.1, above, will be used. In the first example:

```
EXF(2)
```

no evaluation is necessary. The value 2 is placed in a storage location and R0 is set to point to it.

In the second example,

```
EXF(SIN(X+5))
```

SIN(X+5) will be evaluated and stored as above.

In the third example, X↑2 will be evaluated and R1 will point to the comma between X↑2 and XY to allow the external program to pick up the remaining parameters. XY and BC#@ are not valid BASIC expressions and must, therefore, be dealt with by the user program. The final

parameter, A*Y, is in legal BASIC terminology: The BASIC subroutine EVAL can be called to perform the evaluation (see Section 8.3.1).

8.1.3 Recursive EXF Calls

EXF calls of the form:

```
EXF(EXF(X))
```

are acceptable. The EVAL routine is recursive (as are all of BASIC's internal functions) to allow recursive function calls.

Statements of the type above, however, will cause the external function to be re-entered before the first call is complete; the external function must also be re-entrant if this type of call is utilized.

8.2 REQUIREMENTS FOR THE EXTERNAL ROUTINE

1. The binary routine must be loaded into the highest memory available to the user program. For example, a routine 50_{10} bytes long which is to be loaded into an 8K system would start at location 37410 since it is required to end at location 37472. This allows BASIC to properly set up its stack and user program area.

The upper limit for routines in all configurations is as follows:

Core Size	Highest Free Location
4K	17472
8K	37472
12K	57472
16K	77472
20K	117472
24K	137472
28K	157472

2. The first word of the routine must be the starting address (entry point) of the routine.

3. The address of the first word of the routine must be loaded into location 50. This may be done when the routine is assembled with PAL-11A by placing the following lines of code at the beginning of the user routine; assume label SEXF is the entry point of the external routine:

```
.=5Ø            ;REFERENCE LOCATION 5Ø
.WORD SEXF      ;STORE FIRST WORD ADDRESS
```

4. The transfer address must be specified as follows:

```
.END 52
```

If more than one tape comprises the routine, only the last tape loaded should have a transfer address.

5. All exits from the user routine should be in the form of a jump to location 52.

6. A user routine may place items on the stack for temporary storage as long as they are removed before exit. The one exception is that upon exit from the user routine the top of the stack must contain the function value in 3-word floating-point format.

7. Upon entry to the function, the parameter list evaluation is as follows:

   a. R0 contains the address of the numeric value (3 word floating point) of the first parameter which has been previously evaluated by BASIC.

   b. R1 points to the comma following the first parameter or, if there is none, to the character following the closing parenthesis.

Item b, above, allows the programmer to specify arguments in addition to the one evaluated by BASIC.

8. Upon exit, register R1 must point to the character immediately following the closed parenthesis which ends the function call, and register R5 must be restored.

The user must define and interpret all of the parameters following the first, and assure that a closed parenthesis exists for the function call.

It is not recommended that the EXF function be used in less than an 8K PDP-11 system, unless all optional BASIC functions are deleted. The user area will be reduced in size rather excessively by the EXF function (see Appendix A).

WARNING

It is up to the programmer to assure that storage overflow does not occur. Upon entry to the EXF routine, R5 points to the highest address currently in use by BASIC. The programmer must check to make sure that when he uses the R6 stack that he

never overflows the area used by BASIC (R6 may never contain a lower address than R5). This overflow condition must be checked every time EXF is used, since the BASIC storage is dynamically used and changes often.

8.3 USING BASIC'S INTERNAL ROUTINES FROM "EXF"

8.3.1 EVAL

The programmer may use BASIC's EVAL function to evaluate arguments in a parameter list of the following rules are followed.

1. The parameter to be evaluated must follow all the rules for a normal BASIC expression.
2. EVAL must be called once for each parameter to be evaluated.
3. A parameter must be followed by a comma or a right parenthesis.

Calling EVAL is done using a TRAP call with the value 104536. Upon entry to EVAL, R1 must point to the start of the character string to be evaluated and R5 must have, or be restored to, the same value it had when the EXF routine was entered by BASIC.

Control will be returned to the user routine at the instruction following the EVAL call.

Registers 2, 3, and 4 will contain the value of the expression. Register 1 will contain the address where the scan failed (a comma causes a scan failure, and is therefore an effective delimiter). If the scan fails on any character other than a right parenthesis, R1 will point to the character where the failure occurred. If the scan ended on a right parenthesis, R1 will point to the character following the parenthesis and the V (overflow) bit in the status register will be set. If the V-bit is set on any parameter other than the last or is cleared on the last parameter, a mismatched parenthesis error has occurred.

Errors may occur in the evaluation as follows:

1. If a parenthesis (other than the special case mentioned above) is missing or improperly placed in an expression to be evaluated, a fatal error call is made from EVAL. The user routine will not regain control.

2. If a storage overflow occurs due to an evaluation, a fatal error call is made. The user routine will not regain control. (See Appendix A.6.)

3. If a non-existent variable is referenced, the value of the non-existent variable is assumed to be zero for the evaluation, and the user routine will regain control normally.

When EVAL is called, all registers are used.

The EVAL routine is fully recursive to allow nested function calls. However, care must be taken when using calls of the form:

```
EXF(EXF(X))
```

since the EXF function will be re-entered a second time before the first call is complete (the EXF function must also be re-entrant). Also, deep nesting can cause an overflow if the available user storage is minimal.

8.3.2 Other Routines Available to the User

The following routines are also available to the user routine. For interface information, see Table 8-1 below.

ATOF	-	Convert ASCII string to floating-point number
ATOI	-	Convert ASCII string to integer number
ITOA	-	Convert integer to ASCII string
IMUL	-	Integer multiply
ADDF	-	Floating-point addition
SUBF	-	Floating point subtraction
NEGF	-	Floating-point negative
DIVF	-	Floating-point divide
MULF	-	Floating-point multiply
CMPF	-	Floating-point compare
FIX	-	Convert floating-point number to integer
FLT	-	Convert integer to floating-point number.

Additional routines in BASIC may be used only if the programmer becomes extremely familiar with the internal BASIC structures.

Do not use the routines GTOPR or GETOP as they are used by EVAL and they, in turn, use EVAL in a recursive fashion.

TABLE 8-1

Usage Data for BASIC Functions

NAME	TRAP CALL	INPUT	OUTPUT	REGISTERS USED
ATOF	1Ø44Ø6	Pointer to ASCII String in R1	3-word F.P. Number stored where RØ pointed at entry. R1 points to 1st illegal character	ALL
ATOI	1Ø441Ø	Pointer to ASCII	Number in RØ. R1 points to first illegal character	ALL
ITOA	1Ø4412	Number in R1	Pointer to output area in RØ	ALL
IMUL	1Ø4416	Numbers in RØ and R1	High order in RØ Low order in R1	ALL
ADDF	1Ø442Ø	Pointers to num-	Result where RØ pointed at entry	ALL
SUBF	1Ø4422	Pointers to num- bers in RØ and R1	Result where RØ pointed at entry	ALL
NEGF	1Ø4424	Pointer to number in R1	Result where RØ pointed at entry	ALL
DIVF	1Ø4426	Pointers to num- bers in RØ and R1	Result where RØ pointed at entry	ALL
MULF	1Ø443Ø	Pointers to num- bers in RØ and R1	Result where RØ pointed at entry	ALL
CMPF	1Ø4434	Pointers to num- bers in RØ and R1	Same condition codes as CMP in- struction.	ALL
FIX	1Ø444Ø	Number in R2, R3, R4	Result in RØ	RØ, R2, R3, R4
FLT	1Ø4436	Number in R1	Result where RØ pointed at entry	RØ-R4

CHAPTER 9

DEMONSTRATION PROGRAMS

The following programs illustrate some typical BASIC instruction sequences and, additionally, provide models for some useful program types, e.g., loan payment computation and measurement conversion. Some of the programs were typed into core and LISTed for inclusion below.

Program 1:

This program illustrates the use of the comma and semicolon in PRINT statements. Also at line 30, the PRINT statement is used without a variable list to insert a blank line in the printout.

```
LIST

5 REM - USE OF CONTROL CHARACTERS IN "PRINT".
10 READ A, B, C
20 PRINT A, B, C, A↑2, B↑2, C↑2
30 PRINT
40 PRINT A; B; C; A↑2; B↑2; C↑2
50 DATA 2, 4, 6
99 END
READY
RUN
 2             4             6             4             16
 36

 2  4  6  4  16  36

STOP AT LINE   99
READY
```

Program 2:

A program to compute grade averages. Line 40 to 120 (FOR-NEXT statements) form a program loop. Within this loop, lines 75 to 90 form an inner nested loop. Note lines 100 and 130 where computations are performed within PRINT statements.

```
LIST

10 REM - PROGRAM TO TAKE AVERAGE OF STUDENT GRADES AND CLASS GRADES.
20 PRINT "HOW MANY STUDENTS, HOW MANY GRADES PER STUDENT";
30 INPUT A, B
35 LET V = 0
40 FOR J = 1 TO A
50 PRINT "STUDENT NUMBER IS: "; J
60 PRINT "ENTER GRADES:"
70 LET M = 0
75 FOR K = 1 TO B
80 INPUT G
85 LET M = M+G
90 NEXT K
100 PRINT "AVERAGE GRADE IS: "; M/B
105 PRINT
110 LET V = V+M/B
120 NEXT J
130 PRINT "CLASS AVERAGE IS: "; V/A
140 END
READY

RUN
HOW MANY STUDENTS, HOW MANY GRADES PER STUDENT?3,4
STUDENT NUMBER IS:  1
ENTER GRADES:
?88
?92
?85
?79
AVERAGE GRADE IS:  86

STUDENT NUMBER IS:  2
ENTER GRADES:
?70
?87
?0
?76
AVERAGE GRADE IS:  58.25

STUDENT NUMBER IS:  3
ENTER GRADES:
?78
?86
?82
?74
AVERAGE GRADE IS:  80

CLASS AVERAGE IS:  74.75

STOP AT LINE  140
READY
```

Program 3:

This program provides an algorithm for computing roots. Extra spaces in the formatting of lines 30 to 70 have no effect on program execution.

```
10 PRINT "NUMBER","SQUARE ROOT","CUBE ROOT","FOURTH ROOT"
20 FOR N = 1 TO 10
30    PRINT N,
40    FOR R = 2 TO 4
50       PRINT N↑(1/R),
60    NEXT R
70    PRINT
80 NEXT N
99 END

RUN
NUMBER          SQUARE ROOT     CUBE ROOT       FOURTH ROOT
 1               1               1               1
 2               1.414214        1.259921        1.189207
 3               1.732051        1.44225         1.316074
 4               2               1.587401        1.414214
 5               2.236068        1.709976        1.495349
 6               2.44949         1.817121        1.565085
 7               2.645751        1.912931        1.626577
 8               2.828427        2               1.681793
 9               3               2.080084        1.732051
 10              3.162278        2.154435        1.778279

STOP AT LINE   99
READY
```

Program 4:

This program illustrates the use of RESTORE. At line 20, the READ statement associates the first data element in line 90 with the variable N. The next data element in line 90 (1), is assigned to the next variable, X, in the READ statement in line 30. The program iterates three times more through the loop formed by statements in lines 30 and 60, printing the remaining values in lines 90 and 95.

At line 65 the DATA statement pointer is reinitialized. Therefore the first constant in the reinitiated DATA list, 4, is available when the "READ X" statement is executed. This value is printed out and the program iterates three more times through the FOR...NEXT loop in statements 80 and 86, printing out the succeeding three values in lines 90-95.

```
10 REM - PROGRAM TO ILLUSTRATE USE OF RESTORE.
20 READ N
25 PRINT "VALUES OF X ARE:"
30 FOR I=1 TO N:  READ X:  PRINT X,
60 NEXT I
65 RESTORE
70 PRINT:  PRINT "SECOND LIST OF X VALUES ARE:"
71 PRINT "RESTORE STATEMENT USED HERE."
80 FOR I=1 TO N:  READ X:  PRINT X,
85 NEXT I
90 DATA 4, 1, 2, 3, 4
99 END

RUN
VALUES OF X ARE:
 1             2             3             4
SECOND LIST OF X VALUES ARE:
RESTORE STATEMENT USED HERE.
 4             1             2             3
STOP AT LINE   99
READY
```

Program 5:

This program utilizes the INTeger function at line 25, to round off to integer notation the computed value of the variable B.

```
 1 REM -- AN EXAMPLE OF THE "INT" FUNCTION.
 2 REM --- BASIC WILL CONTINUE TO ASK FOR THE NEXT
 3 REM ---  NUMBER TO BE ROUNDED.  THERE IS NO "END"
 4 REM ---  STATEMENT.
 5 REM --- TYPE CTRL/P TO TERMINATE THE PROGRAM.
10 PRINT
15 PRINT "NUMBER TO BE ROUNDED:":  INPUT A
20 PRINT "NO. OF DECIMAL PLACES:":  INPUT D
25 LET B = INT(A*10↑D+.5)/10↑D
30 PRINT "A ROUNDED =" B
35 GOTO 10

RUN

NUMBER TO BE ROUNDED:
? 55.65842
NO. OF DECIMAL PLACES:
? 2
A ROUNDED = 55.66

NUMBER TO BE ROUNDED:
? 67.89
NO. OF DECIMAL PLACES:
? 0
A ROUNDED = 68

NUMBER TO BE ROUNDED:
? 78.375
NO. OF DECIMAL PLACES:
? 1
A ROUNDED = 78.4
```

(continued on next page)

```
NUMBER TO BE ROUNDED:
? 78.375
NO. OF DECIMAL PLACES:
? -3
A ROUNDED = 0

NUMBER TO BE ROUNDED:
? 78.375
NO. OF DECIMAL PLACES:
? 4
A ROUNDED = 78.375

NUMBER TO BE ROUNDED:
?↑P
READY
```

Program 6:

This program uses the INT function in testing for the largest factor of a given number. Since it starts with a value greater than that of the largest possible integer and works downward, the first number which qualifies as a factor will also be the largest factor.

```
 5 REM - BASIC WILL PAUSE WHEN CALCULATING
 6 REM -  THE LARGEST FACTOR.
10 PRINT "NUMBER","LARGEST FACTOR"
20 FOR N = 1001 TO 1020 STEP 2
30 PRINT N,
40 FOR F = INT(N/2) TO 1 STEP -1
50 IF N/F <> INT(N/F) THEN 80
60 PRINT F
70 GOTO 90
80 NEXT F
90 NEXT N
99 END

RUN
NUMBER         LARGEST FACTOR
 1001           143
 1003           59
 1005           335
 1007           53
 1009           1
 1011           337
 1013           1
 1015           203
 1017           339
 1019           1

STOP AT LINE   99
READY
```

Program 7:

This program converts metric length measurements to feet. By substituting an INPUT statement for the DATA and READ statements in lines 100, 240, and 250, the program could be converted to an on-line calculator program.

```
100 READ M,C
110 LET M1 = M+C/100
120 LET I = M1*39.37
130 LET F = INT(I/12)
140 LET I = I-12*F
150 PRINT M,"METERS,",C,"CENTIMETERS"
160 PRINT "CONVERTS TO:"
170 IF F = 0 THEN 190
180 PRINT F, "FEET,",
190 PRINT I, "INCHES"
200 PRINT
230 GOTO 100
240 DATA 1, 0
250 DATA 0, 2.54, 0, 60, 2, 5
260 END

RUN
 1            METERS,        0            CENTIMETERS
CONVERTS TO:
 3            FEET,          3.37         INCHES

 0            METERS,        2.54         CENTIMETERS
CONVERTS TO:
 .999998      INCHES

 0            METERS,        60           CENTIMETERS
CONVERTS TO:
 1            FEET,          11.622       INCHES

 2            METERS,        5            CENTIMETERS
CONVERTS TO:
 6            FEET,          8.7085       INCHES


ERROR    20 AT LINE  100
READY
```

Program 8:

The use of subroutines is illustrated here. GOSUB is contained in line 80 and RETURN in lines 160, 190, and 210.

There are three paths through the program; which one is taken depends upon the number of possible solutions to the quadratic equation. Program switches, operated through logical tests, are contained in lines 140 and 170.

```
  1 REM -- THIS PROGRAM ILLUSTRATES "GOSUB" AND "RETURN",
  2 REM --   AS WELL AS CERTAIN MATHEMATICAL FUNCTIONS.
 10 DEF FNA(X) = ABS(INT(X))
 20 INPUT A, B, C
 30 GOSUB 100
 40 LET A = FNA(A):  LET B = FNA(B):  LET C = FNA(C)
 70 PRINT
 80 GOSUB 100
 90 STOP
100 REM -- THIS SUBROUTINE PRINTS THE SOLUTIONS OF
101 REM --   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:":  PRINT
185 PRINT "X=" (-B+SQR(D))/(2*A) "  AND"
186 PRINT "X=" (-B-SQR(D))/(2*A)
190 RETURN
200 PRINT "IMAGINARY SOLUTIONS:":  PRINT
201 PRINT "X= (" -B/(2*A) "," SQR(-D)/(2*A) ")  AND"
203 PRINT "X= (" -B/(2*A) "," -SQR(-D)/(2*A) ")"
210 RETURN
300 END

RUN
?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
X= ( 0 ,-1 )

STOP AT LINE   90
READY


RUN
?2, 4, 6
THE EQUATION IS:  2 *X↑2 +  4 *X +  6
IMAGINARY SOLUTIONS:

X= (-1 , 1.414214 )  AND
X= (-1 ,-1.414214 )

THE EQUATION IS:  2 *X↑2 +  4 *X +  6
IMAGINARY SOLUTIONS:

X= (-1 , 1.414214 )  AND
X= (-1 ,-1.414214 )

STOP AT LINE   90
READY
```

This program computes the amount of interest, principal, balance and monthly payments for a given loan. It employs a user-defined function in line 5 and a DIMension statement in line 10.

```
  5 DEF FNT(X) = INT(X*100+.5)/100
 10 DIM P(20)
 20 FOR I = 1 TO 20
 30 READ P(I)
 40 NEXT I
 45 PRINT
 50 PRINT "ENTER AMOUNT OF LOAN ";;
 60 INPUT L
 70 FOR I = 1 TO 20
 80 IF L <= P(I) THEN 100
 90 NEXT I
100 LET R = 5+5*I
101 PRINT
103 PRINT "MONTHLY PAYMENT =$";R
104 PRINT
105 PRINT "FINANCE","APPLIED TO","CLOSING"
106 PRINT "CHARGE","BALANCE","BALANCE"
111 LET A = L
112 LET T = 0
113 LET N = 0
115 LET F = .01*L
120 LET Q = R-F
125 LET T = T+F
126 LET N = N+1
130 LET L = L-Q
140 IF L >= 0 THEN 180
150 LET Q = L+R
160 LET L = 0
180 PRINT FNT(F), FNT(Q), FNT(L)
190 IF L > 0 THEN 115
193 PRINT: PRINT
196 PRINT N; "PAYMENTS"
197 PRINT "TOTAL FINANCE CHARGE =$" FNT(T)
198 PRINT "OVERALL % OF LOAN =" FNT(T/A*100);"%"
199 PRINT "PERSONAL LOAN EQUIVALENT =" FNT(12*T/(N*A)*100);"%"
200 PRINT
210 STOP
215 DATA 200, 300, 400, 500, 720, 840, 960, 1080, 1200
216 DATA 1320, 1440, 1560, 1680, 1800, 1920, 2040, 2160
217 DATA 2280, 2400
218 DATA 1E100
220 END
```

```
RUN

ENTER AMOUNT OF LOAN ?100

MONTHLY PAYMENT =$ 10

FINANCE        APPLIED TO     CLOSING
CHARGE         BALANCE        BALANCE
 1              9              91
 .91            9.09           81.91
 .82            9.18           72.73
 .73            9.27           63.46
 .63            9.37           54.09
 .54            9.46           44.63
 .45            9.55           35.08
 .35            9.65           25.43
 .25            9.75           15.68
 .16            9.84           5.84
 .06            5.9            0

 11 PAYMENTS
TOTAL FINANCE CHARGE =$ 5.9
OVERALL % OF LOAN = 5.9 %
PERSONAL LOAN EQUIVALENT = 6.43 %


STOP AT LINE  210
READY
```

Program 10:

A DIMension statement at line 20 initializes and reserves storage for the values of two subscripted variables. Variable A represents a one-dimensional, and variable B a two-dimensional, array.

```
10 REM -- PROGRAM DEMONSTRATING READING OF
15 REM --  SUBSCRIPTED VARIABLES.
20 DIM A(5), B(2,3)
25 PRINT "A(I) WHERE A=1 TO 5:"
30 FOR I = 1 TO 5
35 READ A(I):  PRINT A(I);
40 NEXT I
45 PRINT:  PRINT
50 PRINT "B(I,J) WHERE I=1 TO 2"
55 PRINT "          AND J=1 TO 3:"
60 FOR I = 1 TO 2:  PRINT
65 FOR J = 1 TO 3
70 READ B(I,J):  PRINT B(I,J);
75 NEXT J:  NEXT I
80 DATA 1, 2, 3, 4, 5,
85 DATA 11, 12, 13, 21, 22, 23
90 PRINT
99 END
```

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

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

 11   12   13
 21   22   23

STOP AT LINE   99
READY
```

Program 11:

The DIMension statement in line 20 forms a 7 by 11 data array. Values are assigned lines 45 and 50 to column 0 and row 0; the remaining values are zero. The array is printed out utilizing two loops within lines 55 to 70.

```
10 REM -- A MATRIX CHECK PROGRAM.
20 DIM A(6,10)
25 FOR I = 0 TO 6
30 FOR J = 0 TO 10:  LET A(I,J) = 0
35 NEXT J
40 NEXT I
45 FOR I = 0 TO 6:  LET A(I,0) = I
50 FOR J = 0 TO 10: LET A(0,J) = J
55 PRINT A(I,J);
60 NEXT J
65 PRINT
70 NEXT I
75 END

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

STOP AT LINE   75
READY
```

APPENDIX A

IMPLEMENTATION NOTES

A.1 COMPARISON WITH DARTMOUTH BASIC

BASIC as implemented on the PDP-11 corresponds with Dartmouth BASIC except as noted below.

A.1.1 Special Features

1. Use of BASIC statements in immediate mode.
2. Ability to use any BASIC command (RUN, LIST, etc.) in deferred mode (with a line number).
3. Recursive subroutine calls.
4. User programs can be halted (with CTRL/P) without clearing of variables. These can be examined with the PRINT statement.
5. Multiple statement lines.
6. Array names consisting of a letter followed by a number.

A.1.2 Restrictions

1. No TAB function in PRINT.
2. No ON statement.
3. No MATrix operations.
4. No character string manipulation.
5. All array variables must be declared in a DIM statement before being used.
6. No expressions of the form (-A)↑B. The variable A must be positive and non-zero.

A.2 USER STORAGE REQUIREMENTS

BASIC can be run in the minimal 4K PDP-11/20 configuration. With the BASIC program in core, and deducting space reserved for the Bootstrap and Absolute Loaders, approximately 450 words are left for total user storage (program storage plus working storage).[1]

Any additional 4K core memory increments are available for user storage unless restricted at load time (see Chapter 7). A 12K configuration would normally provide 8K plus approximately 450 words of user storage.

---

[1] Approximately 1000 words are available if BASIC's arithmetic functions are deleted at load time.

A.3 FACTORS AFFECTING PROGRAM SIZE

BASIC statement names (LET, PRINT, etc.) are stored as single byte codes with values ranging from 140 to 174_8. They are reconstructed for output. Any spaces typed within statement names are ignored, e.g., LE T is accepted as LET.

The text of a user program is stored two characters per word. Beyond the space requirement for statement names and text, BASIC program size is dependent on the following:

1. The number and size of arrays.
2. The number of nested FOR loops.
3. The number of nested subroutines.
4. The number of variables.
5. The number of user-defined functions.

Storage required by the above numbered elements can be estimated as follows:

1. Each array requires a two-word entry for the array identifier and three words of storage for each array element.
2. Each FOR-NEXT loop requires eight storage words. This space is returned as each loop finishes cycling.
3. Each subroutine requires one word during operation.
4. Each simple variable (one- or two-character) requires five storage words.
5. User-defined functions require three storage words plus the number of words required for the function definition.

A.4 IMMEDIATE MODE STATEMENTS

The following BASIC statements can be executed in immediate mode:

LET	INPUT
PRINT	GOTO
READ	GOSUB
RESTORE	RETURN

The LET, PRINT, and INPUT statements can be used independently to perform on-line calculations. The remaining statements, however, are used in debugging deferred mode programs and executing them in special ways (GOTO, for example, can be used to begin program execution at a point other than the first statement).

A.5 COMMANDS IN DEFERRED MODE

PDP-11 BASIC makes no distinction between statements, such as LET and DATA, and what are usually classified as commands, e.g., RUN, LIST, etc. All statements in the command category (RUN, LIST, DELETE, SAVE, and OLD) can be coded in deferred mode, i.e., with a preceding line number. When a deferred mode command is encountered in a user program it is executed and BASIC returns to command mode. RUN is an exception: it causes a program to be re-executed from the lowest numbered statement each time it is encountered.

A.6 STORAGE OVERFLOW

Storage overflow can be caused by a user program being too large or generating more data than can be stored in the program storage area. An overflow is indicated by error code 0; a fatal error.

Storage overflow can be caused by any of the following.

1. Too many variables and/or arrays.
2. Too many nested FOR-NEXT loops,
3. Illegal use of the DEF statement. For example:

```
1Ø DEF FNA(X) = X↑2 + FNA(X)
```

   which causes an infinite amount of storage to be required whenever FNA is used in an expression.

4. Excessive nesting of GOSUB statements. For example:

```
 10 LET Y = 1
 20 INPUT X
 30 GOSUB 100
 40 PRINT X; "! ="; Y
 50 STOP
 90 REM -- CALCULATE X FACTORIAL.
100 LET X = X-1
110 IF X>1 THEN GOSUB 100
120 LET Y = Y*X
130 LET X = X+1
140 RETURN
150 END
```

```
RUN
? 10
 10 ! = 362880

STOP AT LINE   50
READY

RUN
?4500

ERROR      0 AT LINE  110
READY
```

The program above ran successfully until the value assigned to X became so large that more data was generated than could be stored, causing the fatal error condition.

In order to conserve space to allow a program to execute properly, the following actions should be taken.

1. Remove all excess spaces within the program. Since spaces are ignored in BASIC, they do not affect the operation of the program, only its reading ease for the user. For example:

```
10LETY=1
20INPUTX
```

   etc.

2. Remove all REM statements. REM statements consume core and are not executed, hence they can be eliminated to make room for executable statements.

3. Use multiple statement lines. Statements can be combined on a single line, separated only by a colon, thereby eliminating the extra character spaces consumed by the line numbers and spaces. For example:

```
120LETY=Y*X:LETX=X+1
```

   instead of:

```
120 LET Y = Y*X
130 LET X = X+1
```

4. Use subroutines and user-defined functions to perform repetitive processes instead of duplicating groups of statements throughout the program. This is especially important for long programs.

5. Use common expressions to eliminate the need to calculate the same values more than once. For example:

```
10LETZ=A+B*C:LETY=3*Z-2*Z
```

   instead of:

```
10LETY=3*(A+B*C)-2*(A+B*C)
```

6. A final alternative would be to segment the program, i.e., make two or more programs out of the one program.

Storage overflow can cause the latter portion of the user program to be destroyed. For example, consider the following (we use an illegal DEF statement to produce an infinite amount of storage to be required and three RESTORE statements to represent subsequent program statements).

```
10 DEF FNA(X) = X↑2 + FNA(X)
20 PRINT FNA(9)
30 RESTORE
40 RESTORE
50 RESTORE


RUN

ERROR      0 AT LINE    20
READY
```

which causes an infinite amount of storage to be required whenever FNA is used in line 20.

After an overflow, LIST the program to determine whether it is still intact.

```
LIST

10 DEF FNA(X) = X↑2 + FNA(X)
20 PRINT FNA(9)
30 RESTORE
40 RESTORE
8&ASV&@<HD&2HBREM↑P            (CTRL/P was typed here)
2
READY
```

If the program already exists on paper tape (it has been previously SAVEd), then reload it with the OLD command. Otherwise, lengthy programs should then be SAVEd and then read in using the OLD command (OLD restores core), or, if the program is not too large, restart BASIC at location zero (which also restores core) and retype the program.

When the program is SAVEd on the low-speed punch, the punch may be stopped as soon as the garbled text starts listing on the teleprinter. If the high-speed punch is used, you must wait for the punch to stop since there is no visible means to see when the incorrect text starts. However, the OLD process will stop with an error message as soon as the incorrect text is encountered.

After reading in or retyping the program, it can be modified as suggested above so that it requires less core or generates less data. In addition, any missing statements must be retyped.

A.7 EXPONENTIATION

As long as the extended functions (in particular, LOG and EXP) have not been deleted, PDP-11 BASIC always performs exponentiation with LOG and EXP. Thus, small inaccuracies are introduced even if an integer is being raised to an integral power. For example:

```
PRINT 2↑2-4
 .2235174E-7
```

shows that 2↑2 is calculated as

```
4.00000002235174
```

This will cause the following program to run as shown:

```
10 IF 2↑2 = 4 THEN 20
11 STOP
20 STOP
RUN

STOP AT LINE   11
READY
```

Since 2↑2 is not exactly calculated as 4, the THEN 20 portion of the IF-THEN is not executed. However, the statement PRINT 2↑2 will print a precise 4 as a result. This is because at most eight decimal places are printed and, to this precision, 2↑2 is 4.

If this problem arises, there are several methods available for coping with the proglem. If an integer is being raised to an integer power, use INT to discard the fractional part:

```
10 IF INT(2↑2) = 4 THEN 20
11 STOP
20 STOP
RUN

STOP AT LINE   20
READY
```

Another method is to do an "almost equal" compare by writing an IF as:

```
IF ABS(X↑N-Y) <= .1E-6 THEN 20
```

APPENDIX B

STATEMENTS, COMMANDS, FUNCTIONS

B.1 STATEMENTS

Statement	Example of Form	Explanation
LET	LET S1 = expression	Assign the value of the expression to the variable S1.
READ	READ V1,V2,...,Vn	Variables V1 through Vn are assigned to the value of the corresponding constants in the DATA string.
DATA	DATA N1,N2,...,Nn	Constants N1 through Nn are to be associated with corresponding variables in a READ statement.
PRINT	PRINT {list}	Output elements of list to Teletype in accordance with format control characters (, or ;); evaluate expressions, if any, and print constant values.
GOTO	GOTO n	Transfer control to line n and continue execution from there.
IF THEN	IF S1 r S2 THEN n IF S1 r S2 THEN stmt	If the relationship r between the formulas or constants S1 and S2 is true then transfer control to line n, or execute the statement; if not, continue in regular sequence.
IF-GOTO	IF S1 r S2 GOTO n	Same as IF-THEN n.
FOR-TO	FOR V=E1 to E2 STEP E3	Used to implement loops; the variable V is set equal to the expression E1. From this point the loop cycle is completed following which V is incremented after each cycle by E3 until its value is greater than or equal to E2. If STEP E3 is omitted, E3 is assumed to be +1.
NEXT	NEXT V	Used to return to the FOR statement and execute the loop again until V is greater than or equal to E2.
DIM	DIM V(S)	Enables the user to create a table or array with the specified number of elements where V is the variable name and S is the maximum subscript value. Any number of arrays can be dimensioned in a single DIM statement, separated by commas. (S starts at 0).

Statement	Example of Form	Explanation
GOSUB	GOSUB n	Allows the user to enter a subroutine at several points in the program. Control transfers to line n.
RETURN	RETURN	Must be at the end of each subroutine to enable control to be transferred to the statement following the matching GOSUB.
RANDOMIZE	RANDOMIZE	Enables the user to obtain an unreproducible random number sequence in a program using the RND function.
INPUT	INPUT V1,V2,...,Vn	Causes typeout of a ? to the user, waits for the user to supply the values of the variables V1 through Vn.
REMARK	REM	When typed as the first three letters of a line allows typing of remarks within the program.
RESTORE	RESTORE	Sets pointer back to the beginning of the string of DATA values.
DEFINE	DEF FNA(X)=F(X)	The user may define his own functions to be called within his program by putting a DEF statement at the beginning of a program. The function name must begin with FN and have three letters. The function is then equated to an expression F(X) which must be only one line long. Multiple arguments are not allowed.
STOP	STOP	Halts program execution (BASIC prints STOP AT LINE xxx and READY.
END	END	Last statement (highest numbered) in every program; signals end of the program coding.

B.2 EDIT AND CONTROL COMMANDS

Several commands for editing BASIC programs and for controlling their execution enable you to: delete lines, list your program, save programs on paper tape, load old programs from paper tape, etc.

B.3 COMMANDS

The commands may be given at any time and are not preceded by a line number.

Command	Action
DELETE n	Delete the line with line number n, an alternate form is to type the line number and the RETURN key.
DELETE n,m	Delete the lines with line numbers n through m inclusive.
LIST	List the entire program on the teleprinter.
LIST n	List line n.
LIST n,m	List lines n through m inclusive.
OLD	BASIC loads a paper tape program.
RUN	Execute the program currently in core.
SAVE	Save the contents of user storage on paper tape.
CTRL/P	Stops a running program, prints ↑ and returns to command mode.
CTRL/U	Erases an entire input line
RUBOUT	Erases single character each time key is depressed.

B.4 ARITHMETIC OPERATORS

Symbol In BASIC	Example	Meaning	Priority
↑	A↑B	Exponentiation	First
*	A*B	Multiplication	Second
/	A/B	Division	
+	A+B	Addition	Third
-	A-B	Subtraction	

A unary minus (negative number) has same priority as normal addition or subtraction.

B.5 LOGICAL OPERATORS

BASIC Symbol	Mathematical Symbol	BASIC Example	Meaning
=	=	A=B	A equal to B
<	<	A<B	A less than B
<=	≤	A<=B	A less than or equal to B
>	>	A>B	A greater than B
>=	≥	A>=B	A greater than or equal to B
<>	≠	<>	A not equal to B.

B.6 MATHEMATICAL FUNCTIONS

Function	Meaning
SIN(x)	[1]Sine of x (x is expressed in radians)
COS(x)	[1]Cosine of x (x is expressed in radians)
ATN(x)	[1]Arctangent of x is returned as an angle in radians in range $\pm\pi/2$
SQR(x)	[1]Square root of x
EXP(x)	[2]e raised to the x power
LOG(x)	[2]Natural logarithm of x
ABS(x)	Absolute value of x
INT(x)	Truncate fraction part of x (truncates to largest integer not greater than x)
RND(x)	Generate random number between 0 and 1
SGN(x)	Sign of x (+1 for positive X; 0 for X=0; -1 for negative x)
FNa(x)	User-defined function; a is any letter; used in DEF statement
EXF(x)	External Function (see Chapter 8)

[1]Extended function; may be deleted during initial dialogue.

[2]Also deletable.

APPENDIX C

ASCII CHARACTER SET

The legal character set in BASIC is:

```
BLANK (SPACE)
A-Z
0-9
* / - + ↑ = > < . , ; ( ) " :
```

Any other non-CTRL ASCII character with an octal value less than 140 is allowed between quotes in a PRINT statement.

EVEN PARITY BIT	7-BIT OCTAL CODE	CHARACTER	REMARKS
0	000	NUL	NULL, TAPE FEED, CONTROL SHIFT P.
	001	SOH	START OF HEADING; ALSO SOM, START OF MESSAGE, CONTROL A,
1	002	STX	START OF TEXT; ALSO EOA, END OF ADDRESS, CONTROL B,
0	003	ETX	END OF TEXT; ALSO EOM, END OF MESSAGE, CONTROL C,
1	004	EOT	END OF TRANSMISSION(END): SHUTS OFF TWX MACHINES, CONTROL D,
0	005	ENQ	ENQUIRY(ENQRY); ALSO WRU, CONTROL E,
0	006	ACK	ACKNOWLEDGE. ALSO RU, CONTROL F.
1	007	BEL	RINGS THE BELL. CONTROL G.
1	010	BS	BACKSPACE: ALSO FEO, FORMAT EFFECTOR. BACKSPACE SOME MACHINES, CONTROL H.
0	011	HT	HORIZONTAL TAB. CONTROL I.
0	012 (0101)	LF	LINE FEED OR LINE SPACE (NEW LINE): ADVANCES PAPER TO NEXT LINE, DUPLICATED BY CONTROL J.
1	013	VT	VERTICAL TAB (VTAB). CONTROL K.
0	014	FF	FORM FEED TO TOP OF NEXT PAGE (PAGE). CONTROL I.
1	015 (1011)	CR	CARRIAGE RETURN TO BEGINNING OF LINE. DUPLICATED BY CONTROL M.
1	016	SO	SHIFT OUT: CHANGES RIBBON COLOR TO RED. CONTROL N.
0	017	SI	SHIFT IN: CHANGES RIBBON COLOR TO BLACK. CONTROL O.
1	020	DLE	DATA LINK ESCAPE. CONTROL P (DC0).
0	021	DC1	DEVICE CONTROL 1, TURNS TRANSMITTER (READER) ON, CONTROL Q (XON).
0	022	DC2	DEVICE CONTROL 2, TURNS PUNCH OR AUXILIARY ON. CONTROL R (TAPE, AUX ON).
1	023	DC3	DEVICE CONTROL 3, TURNS TRANSMITTER (READER) OFF, CONTROL S (XOFF).
0	024	DC4	DEVICE CONTROL 4. TURNS PUNCH OR AUXILIARY OFF. CONTROL T (TAPE, AUX OFF).
1	025	NAK	NEGATIVE ACKNOWLEDGE: ALSO ERR. ERROR. CONTROL U.
1	026	SYN	SYNCHRONOUS IDLE (SYNC). CONTROL V.

012 DEC BIN
012 10 1010

EVEN PARITY BIT	7-BIT OCTAL CODE	CHARACTER	REMARKS
0	027	ETB	END OF TRANSMISSION BLOCK: ALSO LEM. LOGICAL END OF MEDIUM. CONTROL W.
0	030	CAN	CAN (CANCL). CONTROL X.
1	031	EM	END OF MEDIUM. CONTROL Y.
1	032	SUB	SUBSTITUTE. CONTROL Z.
0	033	ESC	ESCAPE. PREFIX. CONTROL SHIFT K.
1	034	FS	FILE SEPARATOR. CONTROL SHIFT L.
0	035	GS	GROUP SEPARATOR. CONTROL SHIFT M.
0	036	RS	RECORD SEPARATOR. CONTROL SHIFT N.
1	037	US	UNIT SEPARATOR, CONTROL SHIFT O.
1	040	SP	SPACE.
0	041	!	
0	042	"	
1	043	#	
0	044	$	
1	045	%	
1	046	&	
0	047	'	ACCENT ACUTE OR APOSTROPHE.
0	050	(	
1	051	)	
1	052	*	
0	053	+	
1	054	´	
0	055	-	
0	056	.	
1	057	/	
0	060	0	
1	061	1	
1	062	2	
0	063	3	
1	064	4	
0	065	5	
0	066	6	
1	067	7	
1	070	8	
0	071	9	
0	072	:	
1	073	;	
0	074	<	
1	075	=	
1	076	>	
0	077	?	
1	100	@	
0	101	A	
0	102	B	
1	103	C	
0	104	D	
1	105	E	
1	106	F	
0	107	G	
0	110	H	
1	111	I	
1	112	J	
0	113	K	
1	114	L	
0	115	M	
0	116	N	
1	117	O	
0	120	P	
1	121	Q	

EVEN PARITY BIT	7-BIT OCTAL CODE	CHARACTER	REMARKS
1	122	R	
0	123	S	
1	124	T	
0	125	U	
0	126	V	
1	127	W	
1	130	X	
0	131	Y	
0	132	Z	
1	133	[	SHIFT K
0	134	\	SHIFT L
1	135	]	SHIFT M
1	136	↑	SHIFT N } See footnote
0	137	←	SHIFT O
0	140	`	ACCENT GRAVE.
0	175		THIS CODE GENERATED BY ALT MODE.
0	176		THIS CODE GENERATED BY ESC KEY (IF PRESENT)
1	177	DEL	DELETE, RUB OUT.
			LOWER CASE ALPHABET FOLLOWS (TELETYPE MODEL 37 ONLY).
1	141	a	
1	142	b	
0	143	c	
1	144	d	
0	145	e	
0	146	f	
1	147	g	
1	150	h	
0	151	i	
0	152	j	
1	153	k	
0	154	l	
1	155	m	
1	156	n	
0	157	o	
1	160	p	
0	161	q	
0	162	r	
1	163	s	
0	164	t	
1	165	u	
1	166	v	
0	167	w	
0	170	x	
1	171	y	
1	172	z	
0	173	{	
1	174	|	

On some devices, the up-arrow (↑) and left-arrow (←) are replaced by the circumflex (^) and underline (_), respectively.

APPENDIX D

THE BOOTSTRAP AND ABSOLUTE LOADERS

D.1 THE BOOTSTRAP LOADER

The Bootstrap Loader should be toggled into the highest available core memory bank.

Location	Instruction
07	
xx7744	016701
xx7746	000026
xx7750	012702
xx7752	000352
xx7754	005211
xx7756	105711
xx7760	100376
xx7762	116162
xx7764	000002
xx7766	07 xx7400
xx7770	005267
xx7772	177756
xx7774	000765
xx7776	yyyyyy
07	177560

xx represents the highest available core memory bank. For example, the first location of the Loader would be one of the following, depending on memory size, and xx in all subsequent locations would be the same as the first.

Location	Memory Bank	Memory Size
017744	0	4K
037744	1	8K
057744	2	12K
077744	3	16K
117744	4	20K
137744	5	24K
157744	6	28K

The contents of location xx7776 (yyyyyy) in the Instruction column above should contain the device status register address of the paper tape reader to be used when loading the bootstrap formatted tape, specified as follows:

Teletype Paper Tape Reader	177560
High-Speed Paper Tape Reader	177550

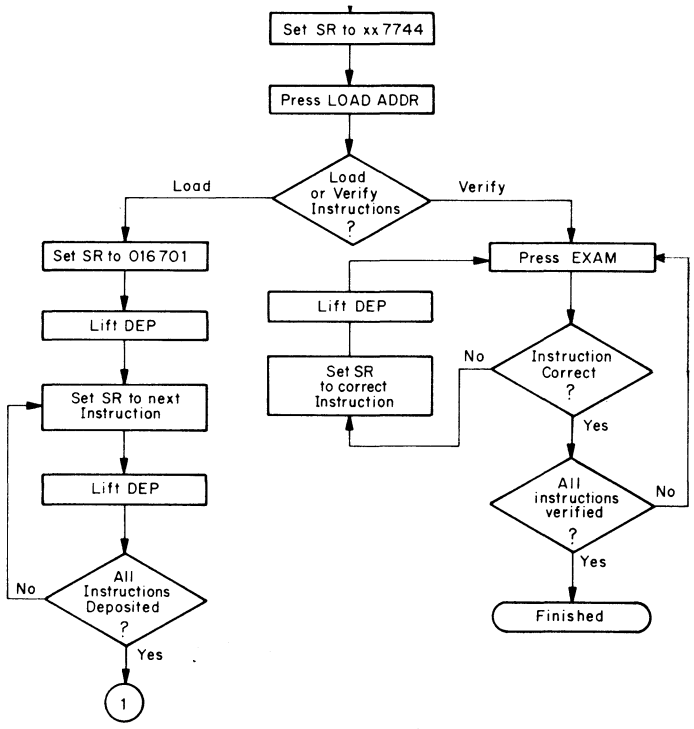


Figure D-1. Loading and Verifying the Bootstrap Loader

D.2 THE ABSOLUTE LOADER

D.2.1 Loading the Absolute Loader

The Bootstrap Loader is used to load the Absolute Loader into core (see Figure D-2). The Absolute Loader occupies locations xx7474 through xx7743, and its starting address is xx7500.

D.2.2 Loading with the Absolute Loader

When using the Absolute Loader, there are three types of loads available: normal, relocated to specific address, and continued relocation (see Figure D-3).

Optional switch register settings for the three types of loads are listed below:

Type of Load	Switch Register Bits 1-14	Bits 0
Normal	(ignored)	0
Relocated - continue loading where left off	0	1
Relocated - load in specified area of core	nnnnn (specified address)	1

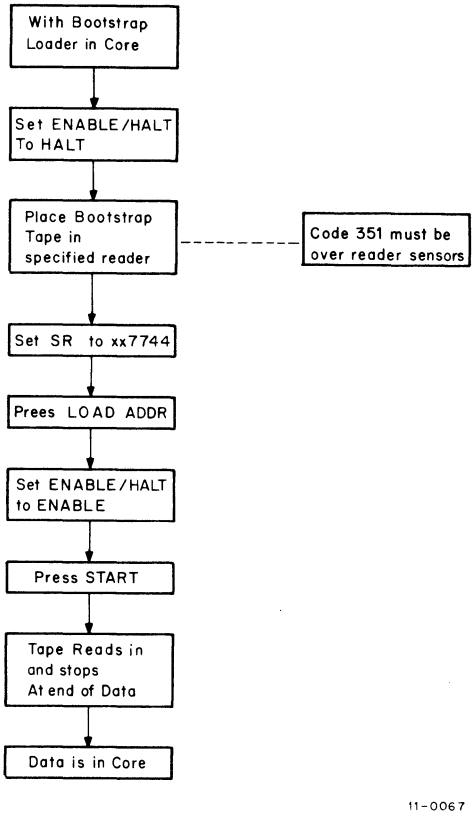


Figure D-2. Loading BASIC Into Core

APPENDIX E

OPERATING THE TELETYPE AND
HIGH-SPEED PAPER TAPE READER/PUNCH

E.1 OPERATING THE TELETYPE

The ASR33 Teletype is the basic input/output device for PDP-11 computers. It consists of a printer, keyboard, paper tape reader, and paper tape punch, all of which can be used either on-line under program control or off-line. The Teletype controls (Figure E-1) are described as they apply to the operation of the computer.

E.1.1 Power Controls

LINE	The Teletype is energized and connected to the computer as an input/output device, under computer control.
OFF	The Teletype is de-energized.
LOCAL	The Teletype is energized for off-line operation.

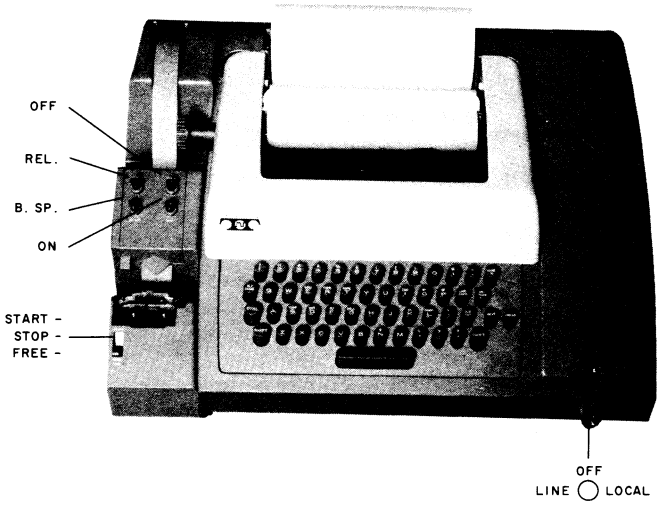


Figure E-1. ASR33 Teletype Console

E.1.2 Printer

The printer provides a typed copy of input and output at 10 characters per second, maximum.

E.1.3 Keyboard

The Teletype keyboard is similar to a typewriter keyboard. However, certain operational functions are shown on the upper part of some of the keytops. These functions are activated by holding down the CTRL key while depressing the desired key. For example, when using the Text Editor, CTRL/U causes Editor to ignore the current line of text.

Although the left and right square brackets are not visible on the keyboard keytops, they are shown in Figure E-2 and are generated by typing SHIFT/K and SHIFT/M, respectively. The ALT MODE key is identified as ESC (ESCape) on some keyboards.

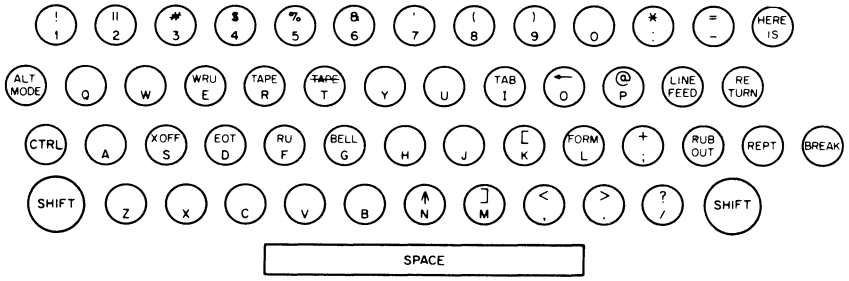


Figure E-2. ASR33 Teletype Keyboard

E.1.4 Paper Tape Reader

The paper tape reader is used to read data punched on eight-channel perforated paper tape at a rate of 10 characters per second maximum. The reader controls are shown in Figure E-1 and described below.

START	Activates the reader; reader sprocket is engaged and operative.
STOP	Deactivates the reader; reader sprocket wheel is engaged but not operative.
FREE	Deactivates the reader; reader sprocket wheel is disengaged.

The following procedure describes how to properly position paper tape in the low-speed reader.

a. Raise the tape retainer cover
b. Set reader control to FREE
c. Position the leader portion of the tape over the read pens with the sprocket (feed) holes over the sprocket (feed) wheel and with the arrow (printed or cut) pointing outward.

d. Close the tape retainer cover; tape should move freely.

e. Set reader control to STOP.

f. Set reader control to START, and the tape will be read into core.

E.1.5 Paper Tape Punch

The paper tape punch is used to perforate eight-channel rolled oiled paper tape at a maximum rate of 10 characters per second. The punch controls are shown in Figure E-1 and described below.

RELease	Disengages the tape to allow tape removal or loading.
B.SP	Backspaces the tape one space for each firm depression of the B.SP button.
ON	Activates the punch.
OFF	Deactivates the punch.

Blank leader/trailer tape is generated by:

1. Turning the TTY switch to LOCAL
2. Turning the LSP on
3. Typing the HERE IS key
4. Turning the LSP off
5. Turning the TTY switch to LINE

E.2 OPERATING THE HIGH-SPEED TAPE READER AND PUNCH UNITS

A high-speed paper tape reader and punch unit is pictured in Figure E-3 and descriptions of the reader and punch units follow.

E.2.1 Reader Unit

The high-speed paper tape reader is used to read data from eight-channel fan-folded (non-oiled) perforated paper tape photoelectrically at a maximum rate of 300 characters per second. Primary power is applied to the reader when the computer POWER switch is turned on. The reader is under program control. However, tape can be advanced past the photoelectric sensors without causing input by pressing the reader FEED button (see Figure 1-4).

E.2.2 Punch Unit

The high-speed paper tape punch is used to record computer output on eight-channel fan-folded paper tape at a maximum rate of 50

characters per second. All characters are punched under program control from the computer. Blank tape (feed holes only, no data) may be produced by pressing the FEED button (see Figure E-3). Primary power is available to the punch when the computer POWER switch is turned on.

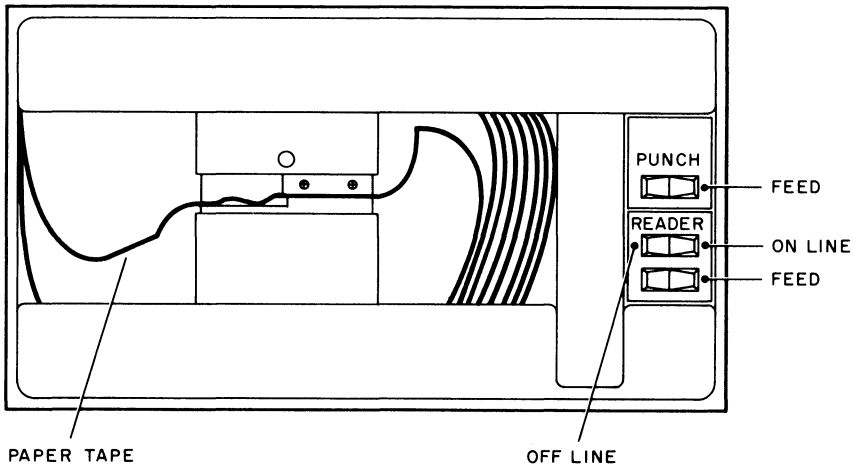


Figure E-3. High-Speed Paper Tape Reader and Punch

Paper tape is loaded into the reader as explained below.

1. Raise tape retainer cover.
2. Put tape into right-hand bin with channel one of the tape toward the rear of the bin.
3. Place several folds of blank tape through the reader and into the left-hand bin.
4. Place the tape over the reader head with feed holes engaged in the teeth of the sprocket wheel.
5. Close the tape retainer cover.
6. Depress the tape feed button until leader tape is over the reader head.

CAUTION

Oiled paper tape should not be used in the high-speed reader - oil collects dust and dirt which can cause reader errors.

While the FEED button is depressed, the punch produces feed-hole-only punched tape for leader/trailer purposes.

HOW TO OBTAIN SOFTWARE INFORMATION

Announcements of new and revised software, as well as programming notes, software problems, and documentation corrections are published by Software Information Service in the following newsletters:

Digital Software News for the PDP-8 and PDP-12

Digital Software News for the PDP-9/15 Family

Digital Software News for the PDP-11

These newsletters contain information to update the cumulative

Software Performance Summary for the PDP-8 and PDP-12

Software Performance Summary for the PDP-9/15 Family

Software Performance Summary for the PDP-11

The appropriate edition of the Software Performance Summary is included in each basic software kit for new customers. Additional copies may be requested without charge.

Any questions or problems on the articles contained in these publications or concerning the use of Digital's software should be reported to the Software Specialist or Sales Engineer at the nearest Digital office.

New and revised software and manuals, and current issues of the Software Performance Summary are available from the Program Library. To place an order, write to:

Program Library
Digital Equipment Corporation
146 Main Street, Building 1-2
Maynard, Massachusetts 01754

When ordering, include the code number and a brief description of the program or manual requested.

Digital Equipment Computer Users Society (DECUS) maintains a user library and publishes a catalog of available programs as well as the DECUSCOPE magazine for its members and non-members who request it. For further information, please write to:

DECUS
Digital Equipment Corporation
146 Main Street, Building 3-5
Maynard, Massachusetts 01754

BASIC
Programming Manual
DEC-11-AJPA-D

READER'S COMMENTS

Digital Equipment Corporation maintains a continuous effort to improve the quality and usefulness of its publications. To do this effectively we need user feedback -- your critical evaluation of this manual.

Please comment on this manual's completeness, accuracy, organization, usability, and readability.

Did you find errors in this manual? If so, specify by page.

How can this manual be improved?

Other comments?

Please state your position. _____ Date: _____

Name: _____ Organization: _____

Street: _____ Department: _____

City: _____ State: _____ Zip or Country _____

— — — — — — — — — — — — — — — — Fold Here — — — — — — — — — — — — — — — — — —

— — — — — — — — — — — — Do Not Tear - Fold Here and Staple — — — — — — — — — — —

FIRST CLASS
PERMIT NO. 33
MAYNARD, MASS.

BUSINESS REPLY MAIL
NO POSTAGE STAMP NECESSARY IF MAILED IN THE UNITED STATES

Postage will be paid by:

digital

Digital Equipment Corporation
Software Information Services
146 Main Street, Bldg. 3-5
Maynard, Massachusetts 01754