DECUS
PROGRAM LIBRARY

DECUS NO.	8-195
TITLE	POLY BASIC
AUTHOR	L. Elekman R. Lary
COMPANY	Digital Equipment Corporation Maynard, Massachusetts
DATE	Submitted: September 16, 1969
SOURCE LANGUAGE	

POLY BASIC

DECUS Program Library Write-up DECUS No. 8-195

DESCRIPTION

POLY BASIC is a stand-alone system designed for a PDP-8, 8I, 8L, with an ASR (or KSR)-33 or 35 Teletype and a random access input/output device. The devices for which it is currently implemented are DF32 Disk, RFØ8 Disk, and TU55 DECtape.

THE BASIC LANGUAGE

AN EXAMPLE

The following example is a complete BASIC program for solving a system of two simultaneous linear equations in two variables:

$$ax + by = c$$
$$dx + ey = f$$

and then solving two different systems, each differing from this system only in the constants c and f. You should be able to solve this system, if ae - bd is not equal to 0, to find that

$$x = \frac{ce - bf}{ae - bd} \quad \text{and} \quad y = \frac{af - cd}{ae - bd.}$$

If ae - bd = 0, there is either no solution or there are infinitely many, but there is no unique solution. If you are rusty on solving such systems, take our word for it that this is correct. For now, we want you to understand the BASIC program for solving this system.

Study this example carefully; in most cases the purpose of each line in the program is self-evident, then read the commentary and explanation.

```
1Ø READ A, B, D, E
15 LET  G = A*E-B*D
2Ø IF  G= Ø THEN 65
3Ø READ  C, F
37 LET  X = (C*E-B*F)/G
42 LET  Y = (A*F-C*D)/G
55 PRINT  X, Y
6Ø GO TO 3Ø
65 PRINT  "NO UNIQUE SOLUTION"
7Ø DATA 1, 2, 4,
8Ø DATA 2, -7, 5
85 DATA 1, 3, 4, -7
9Ø END
```

Immediately we observe several things about this sample program. First, we see that the program uses only capital letters, since the Teletype has only capital letters. We also see that the letter "oh" is distinguished from the numeral "zero" by having a diagonal slash through the "zero". We make the distinction since, in a computer program, it is not always possible to tell from the context whether the letter or the numeral was intended, unless they have a different appearance. This distinction is made automatically while typing, since the Teletype has one key for "oh" and another for "zero"; and one key for "one", another for the letter "i", and no key for the lower case letter "l".

A second observation is that each line of the program begins with a number. These numbers are called line numbers (and may range from 1 through 4095) and serve to identify the lines, each of which is called a statement. Thus, a program is made up of statements, most of which are instructions to BASIC. Line numbers also serve to specify the order in which the statements are to be performed by BASIC, therefore, you may type your line numbers in any order, however, best results will be obtained if they are in ascending order. As you typed, BASIC sorts out and edits the program, putting the statements into the order specified by their line numbers. (This editing process facilitates the correcting and changing of programs, as explained later.)

A third observation shows that each statement starts, after its line number, with an English word which denotes the type of the statement. There are several types of statements in BASIC, eight of which are discussed in this chapter. Seven of these eight appear in the sample program, above.

A fourth observation, not at all obvious from the program, is that spaces have no signficance in BASIC, except in messages which are to be printed out, as in line number 65 above. Thus, spaces may be used or not used to improve the appearance of a program and make it more readable. Statement 10 could have been typed as 10READA,B,D,E and statement 15 as 15LETG = A*E-B*D.

With this perface, let us go through the example, step by step. The first statement, 10, is a READ statement. It must be accompanied by one or more DATA statements. When BASIC encounters a READ statement while executing your program, it will cause the variables (A,B,D,E) listed after the READ to be given values according to the next available numbers in the DATA statements (lines 70, 80, and 85). In the example, we read A in statement 10 and assign the value 1 to it from statement 70, similarly with B and 2, and with D and 4. At this point, we have exhausted the available data in statement 70, but there is more in statement 80, so we pick up from it the number 2 to be assigned to E.

Next we go to statement 15, which is a LET statement, and encounter a formula to be evaluated. (The asterisk "*" is used to denote multiplication.) In this statement, we direct BASIC to compute the value of AE - BD, and to call the result G. In general, a LET statement directs BASIC to set a variable equal to the formula on the right side of the equal sign.

We know that if G is equal to zero, the system has no unique solution, therfore, we ask in line 20, if G is equal to zero. If BASIC discovers a yes answer to the question, it is directed to go to line 65, where it prints NO UNIQUE SOLUTION. From this point, it would go to the next statement, but lines 70, 80, and 85 give it no instructions since DATA statements are not executed, therefore, it goes to line 90 which tells it to END the program.

If the answer to the question "Is G equal to zero?" is no, as it is in this example, BASIC goes on to the next statement, in this case 30. (Thus, an IF-THEN tells BASIC where to go if the IF condition is met, or to go on to the next statement if it is not met.) BASIC is now directed to read the next two entries from the DATA statement, -7 and 5, (both are in statement 80) and to assign them to C and F respectively. BASIC is now ready to solve the system

$$\begin{aligned} x + 2y &= -7 \\ 4x + 2y &= 5 \end{aligned}$$

In statements 37 and 42, we direct BASIC to compute the value of X and Y according to the formulas provided. Note that we must use parentheses to indicate that CE - BF is divided by G; without parenthese, only BF would be divided by G, which would let $X = CE - \frac{BF}{G}$.

BASIC is told to print the two values computed, that of X and that of Y, in line 55, then it moves on to line 60 where it is directed back to line 30. If there are additional numbers in the DATA statements, as there are here in 85, it is told in line 30 to take the next number and assign it to C, and the one after that to F. BASIC is now ready to solve the system

$$\begin{aligned} x + 2y &= 1 \\ 4x + 2y &= 3 \end{aligned}$$

As before, it finds the solution in 37 and 42 and prints them out in 55, and then is directed in 60 to go back to 30.

In line 30 BASIC reads two more values, 4 and -7, which are found in line 85, and then proceeds to solve the system

$$\begin{aligned} x + 2y &= 4 \\ 4x + 2y &= -7 \end{aligned}$$

and to print out the solutions. It is directed back to 30, but there are no more pairs of numbers available for C and F in the DATA statements. BASIC then informs you that it is out of data by typing ERROR DA and stops.

Let us look at the importance of the various statements. For example, what would have happened if we had omitted line number 55? The answer is simple; BASIC would have solved the three systems and then told us when it was out of data. However, since it was not asked to tell us (PRINT) its answers, the solutions would be BASIC's secret. What would have happened if we had left out line 20? In the problem just solved nothing would have happened, but if G were equal to zero, we would have given BASIC the impossible task of dividing by zero in 37 and 42, and it would tell us so by printing ERROR DØ. If we left out statement 60, BASIC would have solved the first system, printed out the values of X and Y, and then gone on to line 65 where it would be directed to print NO UNIQUE SOLUTION. It would do this and then stop.

One very natural question arises from the seemingly arbitrary numbering of the statements: Why this selection of line numbers? The answer is that the particular choice of line numbers is arbitrary, as long as the statements are numbered in the order in which we want BASIC to follow in executing the program. We could have numbered the statements 1, 2, 3, ..., 13, although we do not recommend this numbering. We would normally number the statements 10, 20, 30, ..., 130, allowing additional statements to be inserted later. Thus, if we find that we have left out two statements between those numbered 40 and 50, we can give then any two numbers between 40 and 50, and in the editing and sorting process, BASIC will put them in their proper place.

Another question arises from the seemingly arbitrary placing of the elements of data in the DATA statements: Why place them as they have been in the example program? Here again the choice is arbitrary and we need only put the numbers in the order that we want them read (the first for A, the second for B, the third for D, the fourth for E, the fifith for C, the sixth for F, the seventh for the next C, etc.) In place of the three statements numbered 70, 80, and 85, we could have put

```
75 DATA 1, 2, 4, 2, -7, 5, 1, 3, 4, -7
```

or we could have written, perhaps more naturally,

```
7Ø  DATA  1, 2, 4, 2
75  DATA  -7, 5
8Ø  DATA  1, 3
85  DATA  4, -7
```

to indicate that the coefficients appear in the first data statement and the various pairs of right-hand constants appear in the subsequent statements.

The program and the resulting run is shown below exactly as it appears on the Teletype.

```
LISTNH

1Ø  READ A, B, D, E
15  LET G=A*E-B*D
2Ø  IF G=Ø THEN 65
3Ø  READ C,F
37  LET  X=(C*E-B*F)/G
42  LET  Y=(A*F-C*D)/G
55  PRINT  X, Y
6Ø  GO TO 3Ø
65  PRINT  "NO UNIQUE SOLUTION"
7Ø  DATA 1, 2, 4
8Ø  DATA 2, -7, 5
85  DATA 1, 3, 4, -7
9Ø  END

RUN

TEST 1  BASIC -69
```

```
4                    -5.5
.666666              .166666
-3.66666             3.83333

ERROR  DA  AT  1Ø57          (An indication that the program ran
                              out of data)
READY
```

READY, the last line in the printout above, is explained in Operating the POLY BASIC System.

After typing the program, we type RUN followed by a carriage-return. Up to this point BASIC stores the program and does nothing with it. It is the RUN command which directs BASIC to execute your program.

The message ERROR DA here may be ignored since it means your program has made an attempt to read more data than you have made available in DATA statments.

Arithmetic Operations

BASIC can add, subtract, multiply, divide, extract square roots, raise a number to a power, and find trigonometric functions such as sine and cosine. We shall now learn how to tell BASIC to perform these various operations in the order that we want them done.

BASIC performs its primary function (that of computation) by evaluating formulas which are supplied in a program. These formulas are very similar to those used in standard mathematical calculation, with the exception that all BASIC formulas must be written on a single line. Five arithmetic operators can be used to write a formula:

Symbol	Example	Meaning
+	A+B	Addition (add B to A)
-	A-B	Subtraction (subtract B from A)
*	A*B	Multiplication (multiply B by A)
/	A/B	Division (divide A by B)
↑ or **	X↑2 or X**2	Raise to the power (find X^2)

We must be careful with parentheses to make sure that those things which we want together are grouped together. We must also understand the order in which BASIC operates. For example, if we type A+B * C↑D, BASIC will first raise C to the power D, multiply this result by B, and then add A to the resulting product. This is the same convention as is usual for: $A+B \times C^D$. If this is not the order intended, then we must use parentheses to indicate a different order. For example, if it is the product of B and C that we want raised to the power D, we must write

A+(B * C)↑D; or, if we want to multiply A+B by C to the power D, we write (A+B) * C↑D. We could even add A to B, multiply the sum by C, and raise the product to the power D by writing ((A+B) * C)↑D. The order of priorities is summarized in the following rules:

1. The formula inside parentheses is computed before the parenthesized quantity is used in further computations.

2. In the absence of parentheses in a formula involving addition, multiplication, and the raising of a number to the power, BASIC first performs exponentiation, then performs the multiplication, and the addition comes last. Division has the same priority as multiplication, and subtraction the same as addition.

3. In the absence of parentheses in a formula involving only multiplication and division, the operations are performed from left to right, just as they are read. Addition and subtraction is performed from left to right also.

These rules which are illustrated in the previous example, tell us that when BASIC is faced with A - B - C, it will (as usual) subtract B from A and then C from the difference; with A/B/C, it will divide A by B and that quotient by C; with A↑B↑C, it will raise the number A to the power B and take the resulting number and raise it to the power C. To avoid a question of priority, you may put parentheses in as necessary to eliminate possible ambiguities.

Functions -

In addition to the five arithmetic operators, BASIC can evaluate several mathematical functions. These functions are given special three-letter names, as the following list shows:

Functions	Interpretation	
SIN (X)	Find the sine of X	X interpreted as a number, or as an angle measured in radians.
COS (X)	Find the cosine of X	
ART (X)	Find the arctangent of X	
EXP (X)	Find e^X (2.712818)	
LOG (X)	Find a natural logarithm of X($\log_e X$)	
ABS (X)	Find the absolute value of X (IXI)	
SQR (X)	Find the square root of X ($\sqrt{X}$)	

Three other functions are also available in BASIC: INT, RND, and SGN; these are reserved for explanation in the section ADVANCED BASIC - Functions.

In place of X, we may substitute any formula or any number in parentheses following any of these functions. For example, BASIC may be asked to find $\sqrt{4+X^3}$ by writing SQR(4+X↑3), or the arctangent of 3X - $2e^X$ + 8 by writing

```
ART  (3*X-2* EXP(X)+8)
```

If the value of $(\frac{5}{6})^7$ is needed, you can write the two line program

```
1Ø PRINT (5/6)↑7
2Ø END

RUN NH

.279Ø81

READY
```

and BASIC will find the decimal form of this expression and print it out in less time than it took to type either line. (The command RUN NH is identical to RUN except that no heading is printed.)

Numbers and Variables -

Since we have mentioned numbers and variables, it should be understood how to write numbers for BASIC and what variables are allowed. A number may be positive or negative and may contain up to 10 significant digits (of which only the first 7 are retained) but it must be expressed in decimal form. For example, all of the following are numbers in BASIC: 2, -3.675, 123456789, -.98765432, and 483.4156. The following are not numbers in BASIC: 14/3, $\sqrt{7}$, and .001234567890. The first two are formulas but not numbers, and the last one has more than 10 significant digits. BASIC may be asked to find the decimal expansion of 14/3 or $\sqrt{7}$, and to do something with the resulting number, but neither may be included in a list of DATA. Further flexibility is gained by use of the letter E (exponent), which stands for "times ten to the power"; thus, .00123456789 may be written in any of several forms: .123456789E-2 or 123456789E-11 or 1234.56789E-6. Ten million may be written as 1E7 (or 1E+7) and 1969 as 1.969E3 (or 1.969E+3). E7 is not written as a number, but as 1E7 to indicate that it is 1 that is multiplied by 10^7. Numbers must be in the range 1.0E-614<N<1.0E614.

The BASIC program performs computations in this E (or floating-point) format. Results are printed out in decimal format for numbers in the range 0.01<N<1000000. Trailing decimal points are omitted. Leading and trailing zeroes are also omitted, except when the value is zero or when the number is in the range $0.01 \leq N < 0.1$. Numbers outside the range $0.1 \leq N < 1000000$ are printed out in E format.

A numerical variable in BASIC is denoted by any letter, or by any letter followed by a single digit. The computer, therefore, will interpret E7 as a variable, along with A, X, N5, J0, and M1. A variable in BASIC stands for a number, usually one that is not known to the programmer at the time the program was written. Varibles are given or assigned values by LET and READ statements. The value so assigned will not change until the next time a LET or READ statement is encountered with a value for that variable. However, all variables are set equal to zero before a RUN; thus, it is only necessary to assign a value to a variable when a value other than zero is required.

Although BASIC does little in the way of correcting, during computation it will sometimes help you when you forget to indicate absolute value. For example, if BASIC is asked for the square root of -7 it will give the square root of 7 with the error message for the square root of a negative number.

Symbols of Relation -

Six other mathematical symbols, symbols of relation, are used in BASIC, and these are used in IF-THEN statements where it is necessary to compare values. An example of the use of these relation symbols was given in the example program in the section Functions.

Any of the following six standard relations may be used:

Symbol	Example	Meaning
=	A = B	Is equal to (A is equal to B)
<	A < B	Is less than (Ais less than B)
< = or = <	A<=B	Is less than or equal to (Ais less than or equal to B)
>	A > B	Is greater than (A is greater than B)
= > or > =	A=> B	Is greater than or equal to (A is greater than or equal to B)
<>	A<>B	Is not equal to (A is not equal to B)

LOOPS

We are frequently interested in writing a program in which one or more portions are performed not just once but a number of times, perhaps with slight changes each time. To write the simplest program, the one in which the portion to be repeated is written just once, we use the programming device known as a loop.

Programs which use loops can be best illustrated and explained by two programs which print out a table of the first 100 positive integers together with the square root of each. Without a loop, the program would be 101 lines long and read:

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

With the following program using one type of loop, we can obtain the same table with 5 lines instead of 101:

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

Statement 10 gives the value of 1 to X and "initializes" the loop. In line 20, both 1 and its square root are printed. Then, in line 30, X is increased by 1, to 2. Line 40 asks whether X is less than or equal to 100; an affirmative answer directs BASIC back to line 20. Here BASIC prints 2 and $\sqrt{2}$, and goes to 30. Again X is increased by 1; this time to 3, and at 40 it goes back to 20. This process is repeated (line 20 (print 3 and $\sqrt{3}$), line 30 (X=4), line 40(since 4<100 go back to line 20), etc.) until the loop has been traversed 100 times. Then, after it has printed 100 and its square root X becomes 101. BASIC now receives a negative answer to the question in line 40 (X is greater than 100, not less than or equal to it), it does not return to 20 but moves on to line 50, and ends the program. All loops contain four characteristics:

1. Initialization (line 10),

2. the body (line 20),

3. modification (line 30), and

4. an exit test (line 40).

Because loops are so important and because loops of the type just illustrated arise so often, BASIC provides two statements to specify a loop even more simply. They are the FOR and NEXT statements, and their use is illustrated in the program:

```
1Ø  FOR  X=1  TO  1ØØ
2Ø  PRINT  X,  SQR(X)
3Ø  NEXT  X
5Ø  END
```

In line 10, X is set equal to 1, and a test is set up, like that of line 40 in the previous example program. Line 30 carries out two tasks: X is increased by 1, and the test is made to determine whether to go back to 20 or go on. Thus, lines 10 and 30 take the place of lines 10, 30, and 40 in the previous program, and they are easier to use.

Note that the value of X is increased by 1 each time we go through the loop. If we wanted a different increase, as in increments of 5, we could specify it by writing

```
1Ø FOR X=1 TO 1ØØ STEP 5
```

and BASIC would assign 1 to X on the first time through the loop, 6 to X on the second time through, 11 on the third time, and 96 on the 20th and last time through the loop. (Another step of 5 would take X beyond 100.) The program would proceed to the end after printing 96 and its square root. The STEP may be positive or negative, and we could have obtained the first table, printed in reverse order, by writing line 10 as

```
1Ø FOR X=1ØØ TO 1 STEP -1
```

In the absence of a STEP instruction, a step size of + 1 is assumed.

More complicated FOR statements are allowed. The initial value, the final value, and the step size may all be formulas of any complexity. For example, if N and Z have been specified earlier in the program, we could write

```
FOR X= N + 7 * Z TO (Z-N)/3 STEP (N-4*Z)/1Ø
```

For a positive step-size, the loop continues as long as the control variable is less than or equal to the final value. For a negative step-size, the loop continues as long as the control variable is greater than or equal to the final value.

If the initial value is greater than the final value (less than, for negative step-size) then the body of the loop will be executed once.

It is often useful to have loops within loops, called nested loops, which can be expressed with FOR and NEXT statements, however, they must actually be nested and must not cross, as the following skeleton examples illustrates:

Allowed

```
FOR X
  FOR Y
  NEXT Y
NEXT X
```

Not Allowed

```
FOR X
  FOR Y
NEXT X
NEXT Y
```

Allowed

```
FOR X
  FOR Y
    FOR Z
    NEXT Z
    FOR W
    NEXT W
  NEXT Y
  FOR Z
  NEXT Z
NEXT X
```

LISTS and TABLES

In addition to the ordinary variables used by BASIC, there are variables which can be used to designate the elements of a list or a table. These are used where we might ordinarily use a subscript or a double subscript; for example, the coefficients of a polynomial (a_0, a_1, a_2, ...) or the elements of a matrix ($b_{i,j}$). The variables which we use in BASIC consists of a legal BASIC variable name, which we call the name of the list, followed by the subscripts in parentheses. Thus, we might write A(0),A(1),A(2),etc., for the elements of the list A and B3(0,0),B3(1,0),B3(2,3), etc., for the elements of the table (or matrix) B3.

When using subscripts, a DIMENSION (DIM) statement may be used to indicate that BASIC must save extra space for the list or table. The dimension statement consists of the command DIM, a space, and the variable followed by the largest subscript (parenthesized) to be assigned. If this statement is not given for a list it is as if the statement DIM A(10) were given. As shown in the following examples, more than one variable may be declared in a single DIM statement.

The list A(0),A(1),...,A(10) may be entered into a program very simply by the lines:

```
Ø5  DIM  A(1Ø)
1Ø  FOR I=Ø  TO  1Ø
2Ø  READ A(I)
3Ø  NEXT I
4Ø  DATA 2, 3, -5, 5, 2.2, 4, -7, 123, 4, -4, 3
```

Statement 05 may be omitted, as 10 is the assumed value if no DIM statement is given for A.

We can enter a 3x5 table into a program by writing:

```
Ø5  DIM  B(2,4)
1Ø  FOR H=Ø TO 2
2Ø  FOR  J=Ø TO 4
3Ø  READ  B(H,J)
4Ø  NEXT J
5Ø  NEXT H
6Ø  DATA 2, 3, -5, -9, 2
7Ø  DATA 4, -7, 3, 4, -2
8Ø  DATA 3, -3, 5, 7, 8
```

The variable name associated with a list or table, when used as a scalar name, refers to the first element of the list or table. For example, if the statement DIM C(5), Q3(2,7) was in the program then C would be equivalent to C(0) and Q7 to Q7(0,0). However, the same letter may not be used to denote both a list and a table in the same program. The form of the subscript is quite flexible, and you might have the list item B(H+K) or the table items B(H,K) or Q(A(3,7),B-C).

A list and a run of a problem which uses both a list and a table is shown below. The program computes the total sales of each of five salesmen, all of whom sell the same three products. The list P gives the price/item of the three products and the table S tells how many items of each product each man sold. The program indicates that product No. 1 sells for $1.25 per item, No. 2 for $4.30 per item, and No. 3 for $2.50 per item; and also that salesman No. 1 sold 40 items of the first product, 10 of the second, and 35 of the third, and so on. The program reads in the price list in lines 40-80, using data in lines 910-930. The same program could be used again, modifying only line 900 if the prices change, and only lines 910-930 to enter the sales in another month.

Since the DIM statement is not executed, it may be entered into the program on any line before END and prior to use of the list or table; it is convenient, however, to place DIM statements near the beginning of the program.

```
5 DIM P(3),S(3,5)
1Ø FOR I=1 TO 3
2Ø READ P(I)
3Ø NEXT I
4Ø FOR I=1 TO 3
5Ø FOR J=1 TO 5
6Ø READ S(I,J)
7Ø NEXT J
8Ø NEXT I
9Ø FOR J=1 TO 5
1ØØ LET S=Ø
11Ø FOR I=1 TO 3
12Ø LET S=S + P(I)*S(I,J)
13Ø NEXT I
14Ø PRINT "TOTAL SALES FOR SALESMAN"J,"$"S
15Ø NEXT J
9ØØ DATA 1.25,4.75,2.5Ø
91Ø DATA 4Ø,2Ø,37,29,42
92Ø DATA 1Ø,16,3,21,8
93Ø DATA 35,47,29,16,33
99Ø END

RUN  NH

TOTAL SALES FOR SALESMAN 1  $  185
TOTAL SALES FOR SALESMAN 2  $  218.5
TOTAL SALES FOR SALESMAN 3  $  133
TOTAL SALES FOR SALESMAN 4  $  176
TOTAL SALES FOR SALESMAN 5  $  173

READY
```

ELEMENTARY BASIC STATEMENTS

This section contains a short and concise description of each type of BASIC statement discussed earlier in this chapter and adds one statement to the list. In each form, a line number is assumed, and brackets denote a general type, thus, [variable] refers to any variable, which is a single letter, possibly followed by a single digit.

LET

This statement is not a statement of algebraic equality, but rather a command to BASIC to perform certain computations and to assign the answer to a certain variable. Each LET statement is of the form:

LET [variable] = [formula] .

For example:

```
1ØØ  LET X=X + 1
259  LET  W7=(W-X4↑3) * (Z-A/(A-B)) - 17
```

READ and DATA

A READ statement is used to assign to the listed variable, values obtained from a DATA statement. Neither statement is used without the other type. A READ statement causes the variables listed in it to be given, in order, the next available numbers in the collection of DATA statements. Before the program is run, BASIC takes all of the DATA statements in the order in which they appear and creates a large data block. Each time a READ statement is encountered anywhere in the program, the data block supplies the next available number or numbers. If the data block runs out of data with a READ statement still asking for more, the program is assumed to be done and an OUT OF DATA message is received.

Since data must be read in before it can be worked with, READ statements normally occur near the beginning of a program. The location of DATA statements is arbitrary, as long as they occur in the correct order. A common practice is to collect all DATA statements and place them just before the END statement.

Each READ statement is of the form:

READ [sequence of variables]

and each DATA statement is of the form:

DATA [sequence of numbers]

Example:

```
15Ø  READ  X,  Y,  Z,  X1,  Y2,  Q9
33Ø  DATA  4,  2,  1.7
34Ø  DATA  6.734E-3,  -174.321,  3.1415926
```

```
234  READ B(K)
263  DATA  2,  3,  5,  7,  9,  11,  1Ø,  8,  6,  4

1Ø  READ  R(I,J)
44Ø  DATA  -3,  5,  -9,  2.37,  2.9876,  -437.234E-5
45Ø  DATA  2.765,  5.5576,  2.3789E2
```

Remember that only numbers are put in a DATA statement, and that 15/7 and $\sqrt{3}$ are formulas, not numbers.

PRINT

The PRINT statement has a number of different uses, which are discussed in more detail in the section ADVANCED BASIC. The common uses are described below.

1. To print out the result of some computation:

```
1ØØ  PRINT  X,  SQR(X)
135  PRINT  X,  Y,  Z,  B*B-4*A*C,  EXP(A-B)
```

The first will print X and then several spaces to the right of that number (X), it's square root. The second will print five different numbers:

$$X,\ Y,\ Z,\ B^2 - 4AC, \text{ and } e^{A-B}$$

BASIC will compute the two formulas and print them, as long as values have been given to A, B, and C. It can print up to five numbers per line in this format.

2. To print out verbatim a message included in the program:

```
1ØØ  PRINT "NO UNIQUE SOLUTION"
43Ø  PRINT "X VALUE", "SINE", "RESOLUTION"
```

Both have been encountered in the example programs. The first prints the simple statement; the second prints the three labels with spaces between them. The labels in 430 automatically line up with three numbers called for a PRINT statement.

3. A combination of 1. and 2. above:

```
15Ø  PRINT  "THE VALUE OF X IS"  X
 3Ø  PRINT  "THE SQUARE ROOT OF" X,  "IS" SQR(X)
```

If the first has computed the value of X to be 3, it will print out:

```
THE VALUE OF X IS 3
```

If the second has computed the value of X to be 625, it will print out:

```
THE SQUARE ROOT OF 625 IS 25
```

4. To skip a line (explained in the section ADVANCED BASIC).

We have seen examples of the first three in our previous example programs. Each type is slightly different in form, but all start with PRINT after the line number.

GO TO

In a program there are times when you do not want all commands executed in the order that they appear in the program. If we do not want the program to go to the END statement yet, but to go through the same process for a different value, we direct BASIC to go back to a certain line with a GO TO statement; in the form:

GO TO [line number]

Example:

```
15Ø  GO  TO  75
```

IF -- THEN

There are times when we are interested in jumping the normal sequence of commands, if a certain relationship holds. For this we use an IF--THEN statement, sometimes called a conditional GO TO statement. Each such statement is of the form:

If [formula] [relation][formula] THEN [line number]

Example:

```
4Ø  IF  SIN(X) <= M  THEN  8Ø
2Ø  IF  G = Ø  THEN 65
```

The first asks if the sine of X is less than or equal to M, and directs the computer to skip to line 80 if it is. The second asks if G is equal to 0, and directs the computer to skip to line 65 if it is. In each case, if the answer to the question is no, BASIC will go to the next line of the program.

FOR and NEXT

We have already encountered the FOR and NEXT statement in our loops, and have seen that they go together, one at the entrance to the loop and one at the exit, directing BASIC back to the entrace again. Every FOR statement is of the form:

FOR [variable] = [formula] TO [formula] STEP [formula]

Most commonly, the expressions will be integers and the STEP omitted. In the latter case, a step size of one is assumed. The accompanying NEXT statement is simple in form, but the variable must be precisely the same one as that following FOR in the FOR statement. Its form is NEXT [variable] . The variable used in the FOR and NEXT statements may not be subcripted.

Examples:

```
 3Ø    FOR X=Ø TO 3 STEP D
 8Ø    NEXT X
12Ø    FOR X4 = (17+COS(Z))/3 TO 3*SQR(1Ø) STEP 1/4
235    NEXT X4
24Ø    FOR X=8 TO 3 STEP -1
456    FOR J=-3 TO 12 STEP 2
```

Notice that the step size may be a formula (1/4), a negative number (-1), or a positive number (2). In the example with lines 120 and 235, the successive values of X4 will be .25 apart, in increasing order. In line 240, the successive values of X will be 8, 7, 6, 5, 4, 3. In line 456, on successive trips through the loop, J will take on values -3, -1, 1, 3, 5, 7, 9, and 11.

If the initial, final, or step-size values are given as formulas, these formulas are evaluated once and for all upon entering the FOR statement. The control variable can be changed in the body of the loop; of course, the exit test always uses the latest value of this variable.

If you write 50 FOR Z=2 TO -2, without a negative step size, the body of the loop will be performed once and BASIC will then proceed to the statement immediately following the corresponding NEXT statement.

DIM

Whenever we want to enter a list or a table, we must use a DIM statement to inform BASIC to save sufficient room for the list or table. Examples:

```
2Ø  DIM  H(35)
35  DIM  Q(5,25)
```

An alternate way of writing this would be:

```
2Ø  DIM H(35),  Q(5,25)
```

The first would enable us to enter a list of 35 items, or 36 if we use H(0); and the latter a table 5x25, or by using row 0 and column 0 we get a 6x26 table. All tables must be dimensioned, but a list which is not dimensioned is assumed to have 11 elements numbered 0 through 10.

END

Every program must have an END statement, and it must be the statement with the highest line number in the program. Its form is simple: a line number with END.

```
999  END
```

ERRORS AND DEBUGGING

It may occasionally happen that the first run of a new problem will be free of errors and give the correct answer, but it is much more common that errors will be present and will have to be corrected. Errors are of two types: errors of form (or syntactical errors) which prevent the running of the program; and logical errors (bugs) in the program which cause BASIC to produce wrong answers or no answers at all.

Errors of form will cause error messages to be printed. Logical errors are often much harder to uncover, particularly when the program gives answers which seem to be nearly correct. In either case, after the errors are discovered, they can be corrected by changing lines, by inserting new lines, or by deleting lines from the program. As indicated in the last section, a line is changed by typing it correctly with the same line number; a line is inserted by typing it with a line number between those of two existing lines: and a line is deleted by typing its line number and pressing the RETURN key. Notice that you can insert a line only if the original line numbers are not consecutive integers. For this reason, most programmers will start out using line numbers that are multiples of five or ten, but that is a matter of choice.

These corrections can be made at any time, when in the editing phase and either before or after a run, by simply retyping the offending line with its original line number.

MORE ABOUT PRINT

The uses of the PRINT statement were previously described, but more detail is presented in this section. Although the format of answers is automatically supplied for the beginner, the PRINT statement permits a greater flexibility for the more advanced programmer who wishes a different format for his output.

The Teletype line is divided into five zones of fourteen spaces each. Some control of the use of these comes from the use of the comma: a comma is a signal to move to the next print zone or, if the fifth print zone has just been filled, to move to the first print zone of the next line.

For example, if you were to type the program

```
1Ø   FOR N=1 TO 15
2Ø   PRINT N
3Ø   NEXT N
4Ø   END
```

BASIC would print 1 at the beginning of a line, 2 at the beginning of the next line, and so on, finally printing 15 on the fifteenth line. But, by adding a comma to line 20 to read

```
2Ø   PRINT N,
```

you would have the numbers printed in the zones, reading

```
 1          2          3          4          5
 6          7          8          9          1Ø
 11         12         13         14         15

 READY
```

More compact output can be obtained by use of the semicolon. If a label (expression in quotes) is followed by a semicolon, the label is printed with no space after it. If a variable is followed by a semicolon, its value is printed in the following format:

first, a space, then, a minus sign for negative numbers
then, the numerical value,
then, two spaces

Thus, printing a list of numbers in semicolon format will pack them in the closest readable form.

If you wanted the numbers printed in this fashion, but more tightly packed, you would change line 20 to replace the comma by a semicolon:

```
2Ø   PRINT N;
```

and the result would be printed

```
 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15
```

You should remember that a label inside quotation marks is printed just as it appears and also that the end of a PRINT signals a new line, unless a comma or semicolon is the last symbol.

Thus, the instruction

```
50  PRINT X, Y
```

will result in the printing of two numbers and the return to the next line, while

```
50  PRINT  X,  Y,
```

will result in the printing of these two values and no return. The next number to be printed will occur in the third zone, after the values of X and Y in the first two.

Since the end of a PRINT statement signals a new line,

```
250  PRINT
```

will cause BASIC to advance the Teletype paper one line. It will put a blank line in your program, if you want to use it for vertical spacing of your results, or it causes the completion of a partially filled as illustrated in the following fragment of a program:

```
50   FOR M = 1 TO N
110  FOR J = 0 TO M
120  PRINT B(M,J);
130  NEXT J
140  PRINT
150  NEXT M
```

This program will print B(1,0) and next to it B(1,1). Without line 140, BASIC would then go on printing B(2,0),B(2,1), and B(2,2) on the same line, and then B(3,0),B(3.1), etc. Line 140 directs BASIC, after printing the B(1,1) value corresponding to M=1, to start a new line and to do the same thing after printing the value of B(2,2) corresponding to M=2, etc.

The instructions

```
50  PRINT "BASIC"; " ON "; " THE ";
51  PRINT " PDP-8"; " /I "
52  END
```

will result in the printing of

```
BASIC ON THE PDP-8/I
```

The following rules for the printing of numbers will help you in interpreting your printed results:

1. If a number is an integer, the decimal point is not printed. If the integer contains more than six digits, the number will be printed in E format; the Teletype will give you the first digit, followed by
   (a) a decimal point,
   (b) the next five digits, and
   (c) and E, followed by the appropriate signed integer.

For example, it will take 32,437,580 and write it as 3.243758E +07.

2. For any decimal number, no more than seven significant digits are printed.

3. For a number less than 0.0125, the E notation is used.

4. Trailing zeroes after the decimal point are not printed. The following program, in which we print out powers of 2, shows how numbers are printed.

```
10  FOR N = -5 TO 16
20  PRINT 2↑N;
30  NEXT N
40  END

RUN

0.03125 0.0625 0.125 0.25 0.5 1 2 4 8 16 32 64 128 256
512 1024 2048 4096 8192 16384 32768 65536
```

FUNCTIONS

Three functions were mentioned in THE BASIC LANGUAGE; they are described below.

The INT Function

The INT function frequently appears in algebraic computation as X, and it gives the greatest integer not greater than X. Thus, INT (2.35) = 2, INT (-2.35) = -2, and INT (12) = 12.

One use of the INT function is to round numbers. We may use it to round to the nearest integer by asking for INT (X + .5). This will round 2.9, for example, to 3, by finding INT (2.9 + .5) = INT (10*X+.5)/10↑2 will round X correct to two decimal places, and INT (X*10↑D + .5)/10↑D between two integers up to the larger of the integers).

INT can also be used to round to any specific number of decimal places. For example, INT (10*X+.5)/10↑2 will round X correct to two decimal places, and INT (X*10↑D+ .5)/10↑D will round X correct to D decimal places.

The RND Function

The function RND produces a random number between 0 and 1. Note that the argument in this function is not used.

If we want the first twenty random numbers, we write the program below and we get twenty six-digit decimals.

```
1Ø  FOR L= 1 TO 2Ø
2Ø  PRINT RND(1),
3Ø  NEXT L
4Ø  END

RUN NH


.222990      .267154     .365537   .781468     .754354
.591682      .627343     .775799   1.2Ø355E-1  .375576
.987772      .382303     .486249   .449Ø7Ø      5.94372E-2
.582546      .16Ø21Ø     .952712   .725329     .867454
```

On the other hand, if we want twenty random one-digit integers, we could change line 20 to read:

```
2Ø  PRINT  INT(1Ø*RND(Ø)),
RUN NH
```

and we would then obtain:

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

READY
```

We can vary the type of random numbers we want. For example, if we want 20 random numbers ranging from 1 to 9 inclusive, we could change line 20 as shown

```
2Ø  PRINT  INT(9*RND(1)+1);
RUN

4 8 7 9 3 8 8 6 6 9 6 9 6 6 3 1 4 7 6 7
```

or we can obtain random numbers which are integers from 5 to 24 inclusive by changing line 20 as in the following example:

```
2Ø  PRINT INT(2Ø*RND(1)+5);
RUN

13  22  18  23  1Ø  22  22  17  17  24  16  22  17  17
11   6  12  18  17  18
```

In general, if we want our random numbers to be chosen from the A integers of which B is the smallest, we would call for INT (A*RND(1)+B).

The SGN Function

The SGN function assigns the value 1 to any positive number, 0 to zero, and -1 to any negative number, thus, SGN (7.23) = 1, SGN(0) 0, and SGN (-.2387) = -1.

DEF Statement

In addition to the standard functions, you can define any other function which you expect to use a number of times in your program by use of a DEF statement. The name of the defined function must be three letters, the first two of which must be FN. Hence, you may define up to 26 functions, e.g., FNA , FNB, etc. These use a DEF statement is shown in the following example.

```
1Ø  DEF FNX(X)=X*X
2Ø  READ A,B
3Ø  PRINT A,B,FNX(B)
4Ø  GO TO 2Ø
5Ø  DATA 1,2,3,4,5,6
6Ø  END

RUN

FNX          BASIC-69

             1                    2                    4
             3                    4                    16
             5                    6                    36

ERROR    DA     AT    1Ø36

READY
```

GOSUB and RETURN

When a particular part of a program is to be performed more than one time, or possibly at several different places in the overall program, it is most efficiently programmed as a subroutine. The subroutine is entered with a GOSUB statement, where the number is the line number of the first statement in the subroutine. For example,

```
9Ø  GOSUB  21Ø
```

directs BASIC to jump to line 210, the first line of the subroutine. The last line of the subroutine should be a RETURN command directing BASIC to return to the earlier part of the program. For example,

```
35Ø  RETURN
```

will tell BASIC to go back to the first line numbered greater than 90, and to continue the program there.

The following example, a program for determining the greatest common divisor of three integers using the Euclidean Algorithm, illustrates the use of a subroutine. The first two numbers are selected in lines 30 and 40 their greatest common divisor (GCD) is determined in the subroutine, lines 200-310. The GCD just found is called X in line 60, the third number is called Y in line 70, and the subroutine is entered from line 80 to find the GCD of these two numbers. This number is, of course, the greatest common divisor of the three given numbers and is printed out with them in line 90.

You may use a GOSUB inside a subroutine to perform yet another subroutine. This would be called nested GOSUBs. In any case, it is absolutely necessary that a subroutine be left only with a RETURN statement, using a GOTO or an IF-THEN to get out of a subroutine will not work properly. You may have several RETURNs in the subroutine so long as exactly one of them will be used.

```
1Ø  PRINT "A","B","C","GCD"
2Ø  READ  A,B,C
3Ø  LET X=A
4Ø  LET Y=B
5Ø  GOSUB 2ØØ
6Ø  LET X=G
7Ø  LET Y=C
8Ø  GOSUB 2ØØ
9Ø  PRINT A,B,C,G
1ØØ  GO TO 2Ø
11Ø  DATA 6Ø,9Ø,12Ø
12Ø  DATA 38456,64872,98765
13Ø  DATA 32,384,72
2ØØ  LET Q INT(X/Y)
21Ø  LET R=X-Q*Y
22Ø  IF R=Ø THEN 3ØØ
23Ø  LET X=Y
24Ø  LET Y=R
25Ø  GO TO 2ØØ
3ØØ  LET G=Y
31Ø  RETURN
32Ø  END
```

```
RUN

GCD          BASIC-69


A                    B                    C                    GCD
 6Ø                   9Ø                   12Ø                   3Ø
 38456                64872                98765                 1
 32                   384                  72                    8

ERROR    DA    AT    1Ø53

READY
```

INPUT

There are times when it is desirable to have data entered during the running of a program. This is particularly true when one person writes the program and enters it into the computer's memory, and other persons are to supply the data. This may be done by an INPUT statement, which acts as a READ statement but does not draw numbers from a DATA statement. If, for example, you want the user to supply values for X and Y into a program, you will type

```
4Ø INPUT X, Y
```

before the first statement which is to use either of these numbers. When it encounters this statement, BASIC will type a question mark. The user types two numbers, separated by a comma or blank, presses the RETURN key, and BASIC goes on with the rest of the program.

Frequently an INPUT statement is combined with a PRINT statement to make sure that the user knows what the question mark is asking for. You might type:

```
2Ø PRINT "YOUR VALUES OF X, Y, and Z ARE";
3Ø INPUT X, Y, Z
```

and BASIC will type out

```
YOUR VALUES OF X, Y, and Z ARE?
```

Without the semicolon at the end of line 20, the question mark would have been printed on the next line.

Data entered via an INPUT statement is not saved with the program.

MISCELLANEOUS STATEMENTS

Several other BASIC statements that may be useful from time to time are STOP, REM and RESTORE.

STOP Statement

STOP is equivalent to GOTO xxxxx, where xxxxx is the line number of the END statement in the program. It is useful in programs having more than one natural finishing point. For example, the following two program portions are equivalent.

```
25Ø  GO  TO  999          25Ø  STOP
34Ø  GO  TO  999          34Ø  STOP
999  END                  999  END
```

REM Statement

REM provides a means for inserting explanatory remarks in a program. BASIC completely ignores the remainder of that line, allowing the programmer to follow the REM with directions for using the program, with identifications of the parts of a long program, or with anything else that he wants. Although what follows REM is ignored, its line number may be used in a GOTO or IF-THEN statement.

```
1ØØ  REM  INSERT DATA IN LINES 9ØØ-998. THE FIRST
11Ø  REM  NUMBER IS N,  THE NUMBER OF POINTS.  THEN
12Ø  REM  THE DATA POINTS THEMSELVES ARE ENTERED,  BY

2ØØ  REM  THIS IS A SUBROUTINE FOR SOLVING EQUATIONS
  .
  .
  .
3ØØ  RETURN
  .
  .
  .

52Ø  GOSUB 2ØØ
```

RESTORE Statement

Sometimes it is necessary to use the data in a program more than once. The RESTORE statement permits reading the data as many additional times as it is used. Whenever RESTORE is encountered in a program, BASIC restores the data to its original state. A subsequent READ statement will then start reading the data all over again. A word or warning: if the desired data are preceded by code numbers or parameters, superfluous READ statements should be used to pass over these numbers. As an example, the following program portion reads the data, restores the data to its original state, and reads the data again. Note the use of line 570 to "pass over" the value of N, which is already known.

```
1ØØ  READ N
11Ø  FOR I=1 TO N
12Ø  READ X
```

```
  .
  .
  .
200  NEXT I
  .
  .
  .
560  RESTORE
570  READ X
580  FOR I=1 TO N
  .
  .
  .
```

OPERATING THE POLY BASIC SYSTEM

To load and initiate the system refer to the Loading Procedure in Appendix C.

While typing a program the following rules apply:

1. If a line is typed with the same line number a a line already in the program, the old line is replaced by the new one. If the new line was only a line number and a carriage-return then both the old and new lines vanish.

2. If a line is typed with a line number which does not already exist in the program the line is inserted into the program in numerical order.

3. If a mistake is detected in the line currently being typed, you have two options, the back-arrow (←) deletes one character every time it is typed and is not inserted into the line itself. The "ALT MODE" character (also called "ESCAPE" and "PREFIX") deletes the entire line, prints "DELETED" and returns the carriage so that the line can be retyped.

4. The following characters are ignored by POLY BASIC - rubouts, blank tape, leader (Control - shift - P), and line-feeds.

The POLY BASIC system obeys 17 commands, only the first two letters of the command are significant.

SCRATCH

SCRATCH initializes the users program area to contain no statements.

RENAME

RENAME renames the users program area - the machine types back "NEW FILE NAME - " and the user may then input a name of up to six characters followed by a carriage-return. These characters become the new name of the program.

NEW

NEW is a combination of SCRATCH and RENAME.

SAVE

SAVE saves the current program on the system device (Disk or tapes), if a file name with the same name already exists on the system device, it is unsaved (see comment on "UNSAVE") and replaced by the current program. If there is insufficient room on the system device to save the program the message "NO SPACE" will be typed. However, the old copy of the program, if one existed, has been deleted.

OLD

OLD recalls a previously saved program from the system device. The computer types "OLD FILE NAME - " and the user types the name of the program he wishes to recall. If the program is not found, the computer types "NO SUCH FILE".

LENGTH

LENGTH types the length of the current user file in terms of blocks on the system device (blocks are 128 words in length). The length of a file can range from 1 to 24.

CATALOG

CATALOG types a list of all programs that are currently saved on the system device. Each program name is followed by its length in blocks. This is useful in determining how much room is left on the system device. The maximum number of blocks which can be used on each type of system device is:

125+256n for DF32 Disk, where n=the number of extra disks (Ø<=n<=3)
192Ø+2Ø48n for RFØ8 Disk, where n = the number of extra disks (n=Ø or 1 only)
1344 for TU55 DECtape or PDP-12 Tape.

The listing of the catalog may be stopped at the end of the currently printing line by striking any keyboard character.

UNSAVE

UNSAVE deletes the program with the same name as the current program from the system device. UNSAVE is fast on disk but may require from several seconds to a few minutes on DECtape. If the program to be unsaved is not on the system device the message "NO SUCH FILE" is typed.

TAPE

TAPE informs the computer that you are about to feed in a paper tape that was either prepared off-line or created by turning on the punch during a "LIST" type command. The computer will then know not to automatically type a line-feed after a carriage-return is inputted, and to ignore lines which are neither text nor legal commands.

KEY

KEY reenters the normal mode of processing, i.e., all carriage-returns are automatically followed by a line-feed and illegal input causes the message "WHAT??" to be typed. Any legal command given under "TAPE" mode also reenters this mode.

LIST

This command lists all or part of the current program. It has several forms:

A) LIST lists the entire program with a one line heading of the program name and date.

B) LISTNH lists the entire program without a heading.

C) LIST n lists users program starting at the first line with line number =>n. Printing a heading first.

D) LISTNHn is the same as LIST n but no heading is printed.

These listings can be terminated at any time by striking any key on the Teletype. If the Teletype punch is on during the execution of a LISTNH command a tape will be punched that can later be read in using the TAPE command. The recommended way to do this is to first type LISTNH without a carriage-return, then turn on the Teletype punch and type a few dozen rubout characters (using the REPT and RUBOUT keys) and a carriage-return. When the listing and punching stops type a few dozen rubouts and rip the tape off of the punch.

ECHO

Successive uses of the ECHO command will turn the Teletype from the half-duplex mode (all typed characters automatically print) to the full-duplex mode (no typed characters print) and back to half-duplex. If one were to type the sequence

```
ECHO
TAPE     (Note: this will not appear on the Teletype)
```

and put a paper tape in the Teletype tape reader, the tape would read in silently. When it was finished you would type "ECHO" to reenter the normal mode and also reenter half-duplex mode.

HIGH

This command is only useful at installations with a high-speed reader and punch. It switches all output to the high-speed punch and accepts input from both the Teletype and the high-speed reader. It is important not to use the Teletype keyboard while the high-speed reader is reading, as this will rather randomly merge the two input sources. It is necessary after mounting a tape in the high-speed reader to type a character on the Teletype to start the reader, a rubout is recommended. HIGH automatically enters the full-duplex mode and the half-duplex mode cannot be reentered until a LOW command is given.

LOW

LOW restores the Teletype as the input and output device.

To read a tape from the high-speed reader:

A) Type HIGH↵.
B) Type TAPE↵. This will not print. However, a few characters will be typed on the high-speed punch.
C) Place the tape in the reader and hit RUBOUT. The tape will read to the end.
D) Type LOW↵. This will not print but the computer will type "READY" and everything will be back to normal.

To punch a tape on the high-speed punch:

A) Type HIGH↵.
B) Turn on the punch and feed about 4 fans of tape.
C) Type LISTNH↵. Do not touch the Teletype until the punch stops. When the punch stops feed about 2 fans of the tape and rip.
D) Type LOW↵. The computer will print "READY" and everything will be cool.

EDIT

This command resequences the users program so that the first line number is 1ØØ and the difference between successive line numbers is 1Ø. All "GOTO", "GOSUB" and "IF" statements are also changed to reflect the changed line numbers. The EDIT command can take up to four seconds on disk and up to twenty seconds on DECtape.

RUN

This command causes the BASIC system to compile and execute the current program. If the program has syntax errors an appropriate error message will be printed and the program will not execute. A heading is typed before the compilation is initiated. This can be deleted by typing the command as

RUN NH.

If a Teletype key is struck while the heading is printing then the run will be aborted. Likewise, if a key is struck while an error message is printing the compilation will be aborted at the end of the message. If the user wants to stop his program while it is executing he must depress the "BREAK" key on the Teletype. Compilation time varies from 0.5 to 0.7 seconds on disk and from 15 to 9Ø seconds on DECtape, depending on the size of the program to be compiled.

DEBUG

This command sets a compiler switch so that a symbol table is printed at the end of compilation. A sample table is given in Appendix B. This switch also causes the BASIC system to halt after typing any execution time error message (see Appendix A) so the user can examine core. This command is not too useful to people who do not know the standard DEC Floating Point representations and a little more about the BASIC system than this manual provides.

SPECIAL FEATURES OF POLY BASIC

POLY BASIC has several speical features which make it more useful. They are:

A) The character " \ " (shift-L) can be used to put more than one statement on a line.
Example:

```
1ØØ FOR I=1 TO 1Ø\PRINT I,\NEXTI\END
```

B) The character "#" can be used to obtain the ASCII value of any printing character except " ← " (value=233.), carriage-return (value=141.) and line-feed (value=138.).
Example:

```
1ØØ LET A=#B      sets A to 194. (octal 3Ø2)
```

C) The form 'exp , where "exp" is any legal arithmetic expression, appearing in a print statement will print the single ASCII character whose value is "exp".

D) The form 'variable in an input statement will input a single character from the Teletype and store its ASCII value in "variable".
Example:

```
1Ø FOR I=1 TO 99\ INPUT 'A(I) \IF A(I)=#. THEN 5Ø \NEXT I
5Ø PRINT
```

will input a sentence ending in a period from the Teletype and store it in array A.

E) The statement CHAIN "progname", where "progname" is the name of a saved program padded out with blanks to be six characters long, will terminate execution of the current program and compile and execute the program named progname.
Example:

```
9ØØ CHAIN "BLAH   " will execute the program named BLAH.
```

F) There is a "WRITE" statement in BASIC for writing data onto the system device. Its format is:

WRITE exp1, exp2, exp3... Where the EXPn are any legal BASIC expressions.

The WRITE and READ statements are interconnected in the following manner. Either statement will access the data word after the one accessed by the previous statement.
E. g. the sequence:

```
1ØØ FOR I=1 TO 1ØØ\READ A(I) \NEXT I
```

Will read 1ØØ words from the system device into the array A. The sequence:

```
1ØØ FOR I=1 TO 1ØØ\WRITE A(I) \NEXT I
```

Will write 1ØØ words from the array A onto the system device. And the sequence:

12Ø FOR I=1 TO 1ØØ\ READ A(I)\ WRITE A(I)\ NEXT I

Will read the first number into A(1). Write A(1) over the second number, read the third number into A(2), write A(2) over the fourth number, etc.

The RESTORE statement has the dual function of resetting the pointer to the beginning of the data and also making sure that the last few numbers written find their way onto the system device. Data written on the Disk in one program can be read by another if all of the following three conditions hold:

1) The second program is executed via a CHAIN statement in the first one,
2) The second program contains no DATA statements,
3) The first program executed a RESTORE statement just before the CHAIN statement.

Note: If the data pointer, which is incremented every time a number is read or written and reset to zero by every RESTORE statement, is allowed to get large enough, there is a danger of writing over the text of your program. If your program has a lenght of n then writing more than approximately 237Ø-42n numbers will overwrite your program. Writing more than 237Ø numbers will cause an error message and termination of your execution as this endangers parts of the system.

G) The statement PAUSE or PAUSE n will cause the computer to print a message ("ERROR PA AT xxxx") and halt at execution time. Execution may be continued by setting the switch register to Ø and pressing CONTINUE. The number n, if it exists, is used to reserve core for machine language programs which the user wants interfaced to BASIC. It is a decimal number representing the highest location which BASIC should use for storage of program and variables. BASIC usually uses up to location 256Ø decimal (5ØØØ octal), so that any value of n should be less than this and greater than about 1ØØØ decimal. For detailed instructions on the interfacing of user programs to BASIC see Appendix D.

APPENDIX A

ERROR MESSAGES

Three types of error messages can occur in BASIC. These messages and their interpretations are shown below.

DURING PROGRAM COMPILATION

The message consists of a two letter code, followed by the line number at which the error was detected.

MO xxxx	Your program is too large to be executed, try to make it smaller or reduce your variable dimensions.
EN xxxx	Your program did not have an END statement.
ST xxxx	A statement was used which is not a legal BASIC statement.
SX xxxx	The structure of the statement does not agree with BASIC syntax.
IC xxxx	You used a character which is illegal in the context you used it.
IN xxxx	The format of a statement number in the statment being processed is not valid.
PC xxxx	The parentheses in this line are not matched or are used improperly.
TO xxxx	Your program contains too many variables, constants, functions and line numbers. Try to cut down.
LI xxxx	There is an illegal constant in this line.
UQ xxxx	There were unmatching quotes in this line.
RE xxxx	A relational operator appeared where it should not.
UL xxxx	The statement number xxxx was referenced in a GOTO, GOSUB, or IF statement but not defined.
TB xxxx	Your program is much too big, usually caused by an extremely large excess of PRINT statements.
DO xxxx	A NEXT statement was found in which the variable did not agree with that in the previous FOR statement.
DN xxxx	At the end of the program there was a FOR statement without a NEXT corresponding to it.

NON-FATAL MESSAGES DURING EXECUTION

These messages print but do not cause program execution to terminate. The form of these messages is "ERROR xx AT yyyy" where xx is the error code and yyyy is the core location at which the error occurred.

Code	Explanation
DØ	A division by zero has occurred, the result of the division is set to some huge number(about 1Ø ↑ 5ØØ) and execution continues.
SQ	A negative square root was attempted - the square root of the absolute value is used.
PA	A PAUSE statement was executed. - This is not an error, merely an informative printout. The machine will halt but execution can be resumed by setting the switch register to zero and press CONTINUE.

FATAL EXECUTION TIME MESSAGES

These messages have the same format as the non-fatal messages except that they terminate execution and return to the editing phase of BASIC.

Code	Explanation
PD	The pushdown list overflowed - This usually means that functions and GOSUBs are nested too deeply. Making the program a little smaller will leave more room for pushdown.
GS	Similiar to PD - usually means that GOSUBs were called recursively to too great a depth.
LG	An attempt was made to take the LOG of a number which was less than or equal to zero.
FN	A user function was called which was not defined in a DEF statement or interfaced through a PAUSE statement.
DA	The program tried to read more data than it had.
SS	A subscript on a variable was larger than the maximum specified for that variable in a DIM statement.
CH	An attempt was made to CHAIN to a non-existant program.
WR	So much data was written with a WRITE statement that the system is in danger of being overwritten.

APPENDIX B

A SAMPLE OF DEBUG OUTPUT

```
1ØØ  REM THIS PROGRAM COMPUTES PRIMES AND STORES THEM IN AN ARRAY
11Ø  INPUT L, J
12Ø  LET K=1
13Ø  IF L>7 THEN 16Ø
14Ø  PRINT 2;3;5;7;
15Ø  LET L=11
16Ø  FOR I=L TO U STEP 2
17Ø  FOR J=3 TO SQR(I) STEP 2
18Ø  IF I=J*INT(I/J) THEN 24Ø
19Ø  NEXT J
2ØØ  PRINT I;
21Ø  DIM A(4ØØ)
22Ø  LET A(K)=I
23Ø  LET K=K+1
24Ø  NEXT I
25Ø  PRINT
26Ø  PRINT K
27Ø  GO TO 11Ø
28Ø  END

DEBUG
READY
RUN


PRIMES   Ø9/15/69

11Ø              1Ø61    The code generated by statement 11Ø starts at
                         location 1Ø61 (Octal).
L                5175    The variable L is in locations 5175-5177 (Octal).
U                5172
K                5167
ØØØ334ØØØØØØ    1ØØ6    The literal 7.Ø, which has an internal representation
                         of ØØØ3 34ØØ ØØØØ, is in locations 1ØØ6-1Ø1Ø
16Ø              1111
ØØØ22ØØØØØØ     1Ø11
ØØØ23ØØØØØØ     1Ø14
ØØØ324ØØØØØ     1Ø17
ØØØ426ØØØØØ     1Ø22
I                5164
J                5161
24Ø              1163
A(4ØØ)           2676    The array A, dimensioned 4ØØ, starts at location 2676.
```

APPENDIX C

LOADING POLY BASIC ON A PDP-8, 8/I or 8/L COMPUTER

BASIC comes as 2 tapes - the BASIC LOADER and BASIC itself.

Follow the steps below to insure proper loading.

a) Make sure the binary loader is in locations 7600 - 7777 of field zero. Then LOAD ADDRESS 7777.

b) Place the tape marked BASIC LOADER in the low-(high) speed reader, and set the switches to 7777 (3777).

c) Press START - the loader should read in. When the reader stops the AC should be zero and the link 1. If not, go back to step a.

d) LOAD ADDRESS 600; make sure the Teletype is on; START.

e) The loader will type HIGH SPEED RDR?.

Type a Y if you have high-speed paper tape, type an N if you do not.

f) The loader will type DF32 DISK?.

First load the tape marked BASIC into the proper reader - then, if you have a DF32 Disk, Type Y and go to step j. Otherwise, type N.

g) The load will type TU55 DECTAPE?.

If you have a TU55 DECtape mount, certified tape, set the unit to unit #8 and WRITE ENABLE type Y, and go to step k. Otherwise type N.

h) The loader will type RFØ8 DISK?.

If you have an RFØ8 Disk type Y and go to step j. If not type N.

i The loader will type "PDP-12 TAPE?" If you have a PDP-12, put a tape which has been blocked at 129 words per record on drive Ø, WRITE ENABLED, type Y, and go to step k. Otherwise, type N and go back to step e.

j The loader will type HOW MANY DISKS?.

Set all disks to WRITE ENABLED (non-protected) state and type the number of disks you have, followed by a carriage-return. Go to step l .

k) The tape should move for a few seconds. This is o.k.

l) If you have a low-speed reader, push the lever to START.

The loader is now reading in the paper tape. If any errors occur it will print "TAPE ERROR" and halt. If this occurs, move the tape backwards in the reader until the first 1-inch blank space before the error is at the read station. This may mean backspacing up to 2 feet of tape. Then press CONTINUE. If error persists, the tape on the reader is probably bad.

m) At some point near the end of the tape the loader should print "FINISHED".
A short time after this the BASIC system will automatically load and print "READY". You may now run BASIC programs.

Note: Loading time on a DF32 or RFØ8 Disk takes approximately 1 minute from high-speed paper tape, 27 minutes from the Teletype. DECtape adds about 1 minute to these times.

The BASIC system has no known bugs which will cause it to crash, but the following tips are in order.

1) If the machine halts due to a PAUSE statement or an execution error under the DEBUG statement and you wish to abort execution, perform LOAD ADDRESS 7600 and START.

2) If someone else uses the machine with some other program, perform the following functions:

DF32 or RFØ8 DISK - enter the locations

775Ø	= 66Ø3	7772 = Ø	
7751	= 7577		
7752	= 5352		
7753	= 5352		

LOAD ADDRESS 7750, START

PDP-12 - Set Left Switches = 0700, Right Swithces = 0. SET mode to LINC. Press DO. Wait for tape to stop. Set Left Switches = 4160. Press START LS.

TU55 DECtape - enter the locations

7613	= 6774	7622	= Ø6ØØ
7614	= 1222	7623	= Ø22Ø
7615	= 6766	7754	= 7577
7616	= 6771	7755	= 7577
7617	= 5216	7772	= Ø
762Ø	= 1223		
7621	= 5215		

LOAD ADDRESS 7613, press START.

```
/                       APPENDIX D
/
/               A SAMPLE USER PROGRAM USING THE PAUSE STATEMENT,
/
/
/       WHEN THE MACHINE PAUSES AT EXECUTION TIME YOU HAVE THE FOLLOWING
/OPTIONS:
/       1       SET SR=0 AND PRESS CONTINUE - CONTINUES EXECUTION
/       2       /SET SR=7777 AND PRESS CONTINUE - LOADS ONE TAPE IN
/               BINARY LOADER FORMAT OFF OF THE LOW SPEED READER AND
/               HALTS AGAIN.
/       3       SET SR=3777 AND PRESS CONTINUE - DOES THE SAME FROM THE
/               HIGH SPEED READER.
/
/               IN THIS WAY YOU CAN LOAD MACHINE LANGUAGE PROGRAMS.  TO
/       INTERFACE THESE PROGRAMS WITH BASIC YOU INCLUDE AN "UNDEFINED
/       FUNCTION HANDLER" SUCH AS IS INCLUDED IN THIS EXAMPLE AND YOU
/       CAN THEN "CALL" YOUR MACHINE LANGUAGE ROUTINES BY USING A
/       FUNCTION WITHOUT DEFINING IT.
/ NOTE: THE FIRST USE OF THE FUNCTION IS MERELY TO INTERFACE IT AND
/       WILL NOT ACTUALLY TRANSFER TO THE MACHINE LANGUAGE.  THE SECOND
/       CALL OF THE FUNCTION WILL ACTUALLY EXECUTE THE MACHINE-LANGUAGE
/       CODED ROUTINE.  IT IS IMPORTANT TO SET UP THE FIRST CALLS
/       TO THE FUNCTIONS(IF THERE ARE MORE THAN ONE) SO THAT THE RIGHT
/       CORRESPONDENCES ARE MADE - I.E, IN THIS EXAMPLE, THE FIRST
/       FUNCTION CALLED WILL BE THE FUNCTION "FN1" AND THE SECOND WILL BE
/       "FN2" REGARDLESS OF THE NAMES THEY ARE GIVEN IN THE BASIC
/       PROGRAM.
/
/
/
```

```
                *5401   /LOCATIO  OF "UNDEFINED FUNCTIO " ERROR ROUTINE
5401  5402      JMP I 2 /TRA SFER TO FUNCTION DEFINER.
      0002      *2
0002  4600      SETUP
      4600      *4600   /THE MAI  PROGRAM SHOULD CO TAI  A "PAUSE 2432"(2432=4600 OCTAL)
4600  1013  SETUP, TAD FPX
4601  5207      DCA STMP        /GET POINTER TO FUNCTIO 'S SYMTAB ENTRY
4602  7240      CLA CMA
4603  1611      TAD I STABL     /GET NEXT MACHINE LA GUAGE FUNCTION TO BE DEFINED
4604  3207      DCA I STMP      /STORE ITS ADDRESS (-1) I  THE SYMBOL TABLE
4605  2210      ISZ STABL       /BUMP POINTER
4606  5244      JMP I RETURN    /RETURN FIRST TIME - FLOATING AC IS UNCHANGED.
4607  0000  STMP,  0
4610  4611  STABL, .+1
4611  4613      FN1             /FUNCTION TO READ A CHARACTER FROM THE HS READER
4612  4627      FN2             /FUNCTION TO PUNCH A CHARACTER ON THE HS PUNCH
4613  7402  FN1,  IEXIT  /EXIT FROM BASIC INTERPRETER
4614  2244      ISZ FIRSTA
4615  7410      SKP
4616  6414      6414    /FEED A CHARACTER FIRST TIME ONLY FOR OVERLAP
4617  3244      DCA FIRSTA      /RESET SWITCH
4620  6011      6011
4621  5220      JMP .-1
4622  6416      6416    /REA  CHAR A D FEED NEXT CHAR
```

```
4623  3027            DCA LORD
4624  3026            DCA HORD        /STORE CHAR AS DOUBLE PRECISION INTEGER IN
                                      /FLOATING AC
4625  4545            JMS I IFLOAT    /CALL ROUTINE TO CONVERT TO FLOATING POINT
4626  5544            JMP I RETURN    /RETURN TO CALLING PROGRAM AND BASIC INTERPRETER
4627  7404    FN2,    IEXIT   /EXIT FROM BASIC INTERPRETER
4630  2245            ISZ FIRSTB      /HAVE WE PUNCHED BEFORE?
4631  7412            SKP             /YES
4632  5235            JMP .+3         /NO - NO NEED TO WAIT,
4633  6021            6021
4634  5233            JMP .-1         /WAIT FOR PUNCH
4635  3245            DCA FIRSTB      /SET PUNCH-IN-USE FLAG
4636  4543            JMS I INT       /CONVERT FLOATING AC TO A DOUBLE PRECISION INTEGER
4637  1227            TAD LORD        /GET THE LOW-ORDER WORD
4640  6026            6026            /PUNCH IT
4641  7200            CLA
4642  4462            JMS I NORM      /ALWAYS LEAVE WITH A NORMALIZED FLOATING AC!
4643  5544            JMP I RETURN
4644  7777    FIRSTA, -1
4645  7777    FIRSTB, -1
        /LOCATIONS IN INTERPRETER USED BY THIS ROUTINE,
0013            FPX=13  /A TEMPORARY WHICH JUST HAPPENS TO CONTAIN THE RIGHT THING
7404            IEXIT=7404      /DONT ASK QUESTIONS - USE IT TO LEAVE
                                /THE BASIC INTERPRETER,
0143            INT=143         /INTEGERIZES THE FLOATING AC,
0144            RETURN=144      /RETURNS FROM FUNCTION TO CALLING PROGRAM
```