SINGLE-USER
HP BASIC
USER MANUAL

Extract from
"A Pocket Guide to Hewlett-Packard Computers"
Hewlett-Packard Company, July 1970

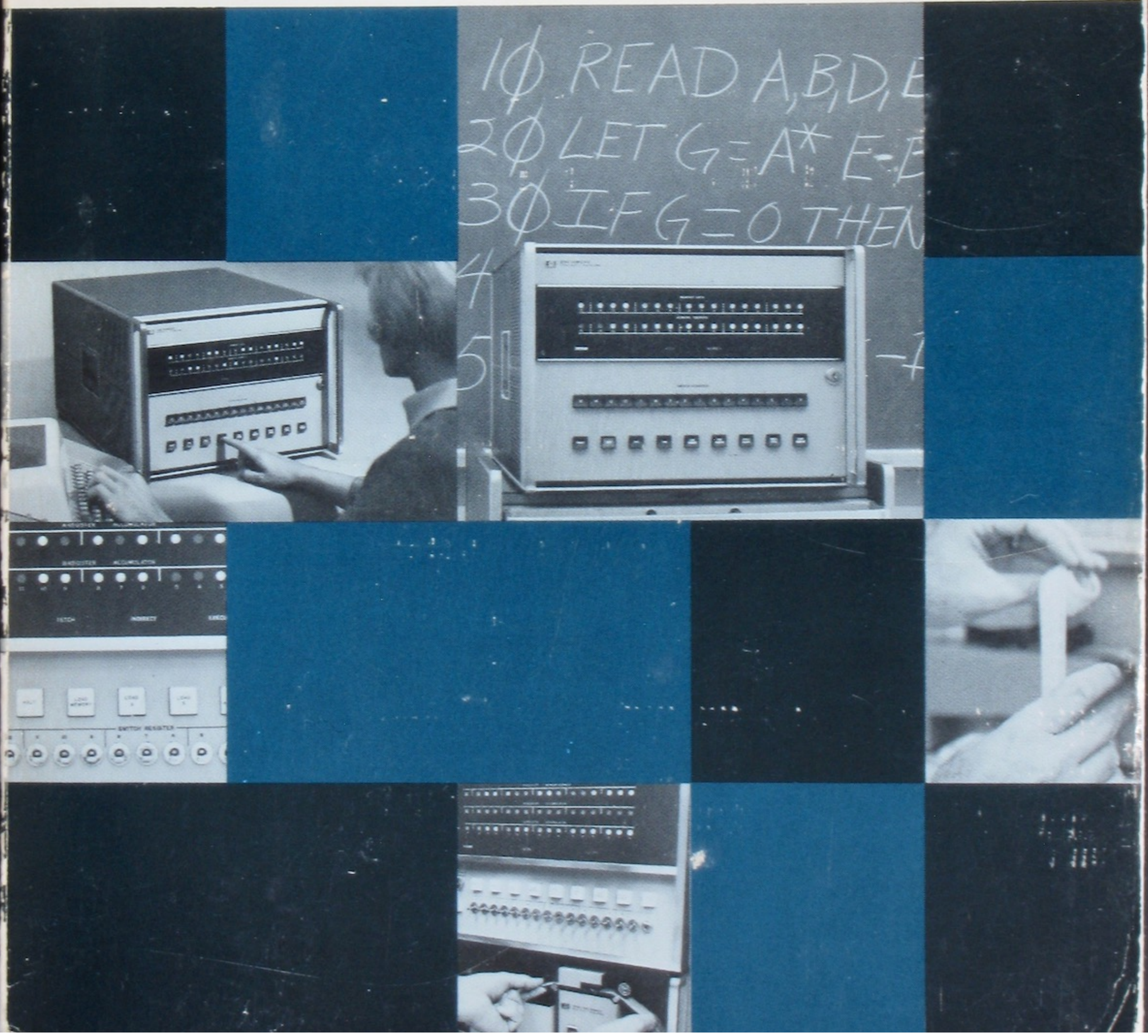
PDF TABLE OF CONTENTS

SECTION			PDF
INTRODUCTION			10
CHAPTER 1	A BASIC PRIMER		11
	1.1	AN EXAMPLE	11
	1.2	FORMULAS	14
	1.3	LOOPS	17
	1.4	ARRAYS	20
	1.5	USE OF THE SYSTEM	22
	1.6	ERRORS & DEBUGGING	23
	1.7	SUMMARY OF STATEMENTS	27
CHAPTER 2	ADVANCED BASIC		31
	2.1	MORE ABOUT PRINT	31
	2.2	FUNCTIONS & DEF	34
	2.3	GOSUB & RETURN	36
	2.4	INPUT	37
	2.5	SOME MISCELLANEOUS STATEMENTS	38
	2.6	MATRICES	40
	2.7	ERROR CODES	46
APPENDIX A	USING THE HP BASIC SYSTEM		49
APPENDIX B	CALL & WAIT STATEMENTS		51
APPENDIX C	LOGICAL OPERATORS		55
APPENDIX D	BASIC SYNTAX IN BNF		56
APPENDIX E	PREPARE-BASIC-SYSTEM PROGRAM (PBS)		60

bh / 2025-06-2

HEWLETT hp PACKARD

A Pocket Guide to Hewlett-Packard Computers



PREFACE

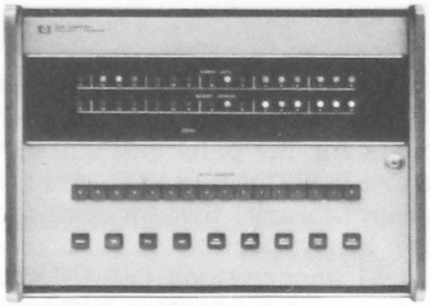
This manual combines in one convenient publication comprehensive hardware specifications for Hewlett-Packard 2100 Series Computers and programmer's reference manuals for the principal software systems. Software manuals included are FORTRAN, BASIC, Assembler, Basic Control System and Program Library. System designers and programmers will find this book a handy, permanent reference. Potential users will find the technical descriptions valuable for evaluating Hewlett-Packard computers and supporting software. Since Hewlett-Packard hardware and software specifications are subject to change, the information in this manual is intended to be used strictly as a guide and does not necessarily represent current policies and products supported by Hewlett-Packard.

Further information on Hewlett-Packard computer products is available from your local Hewlett-Packard field office; one of more than 130 Sales and Service Offices throughout the world.

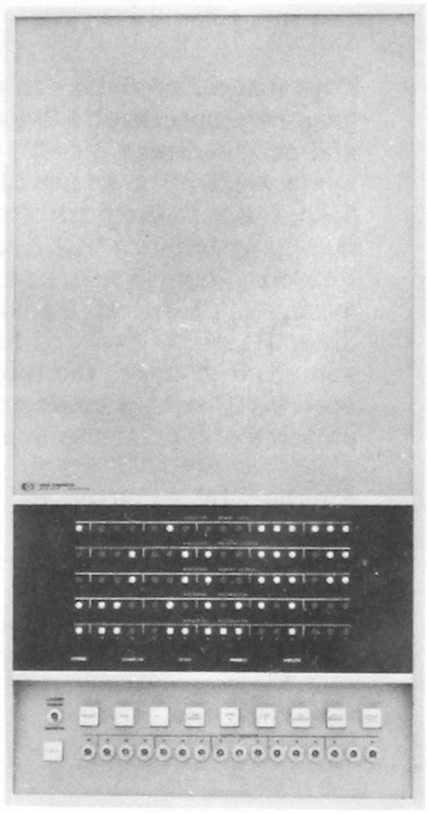
Or write Hewlett-Packard, 1501 Page Mill Road, Palo Alto, California 94304; Europe, 1217 Meyrin-Geneva, Switzerland.

Here is a family of computers with the power to solve problems for engineers and scientists — at a cost that makes them uniquely practical.

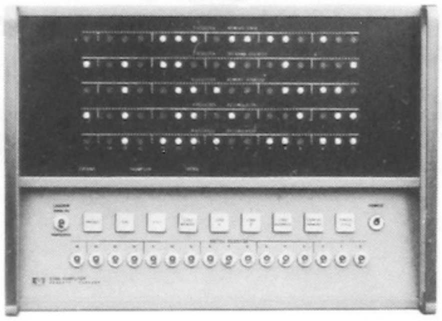
2114



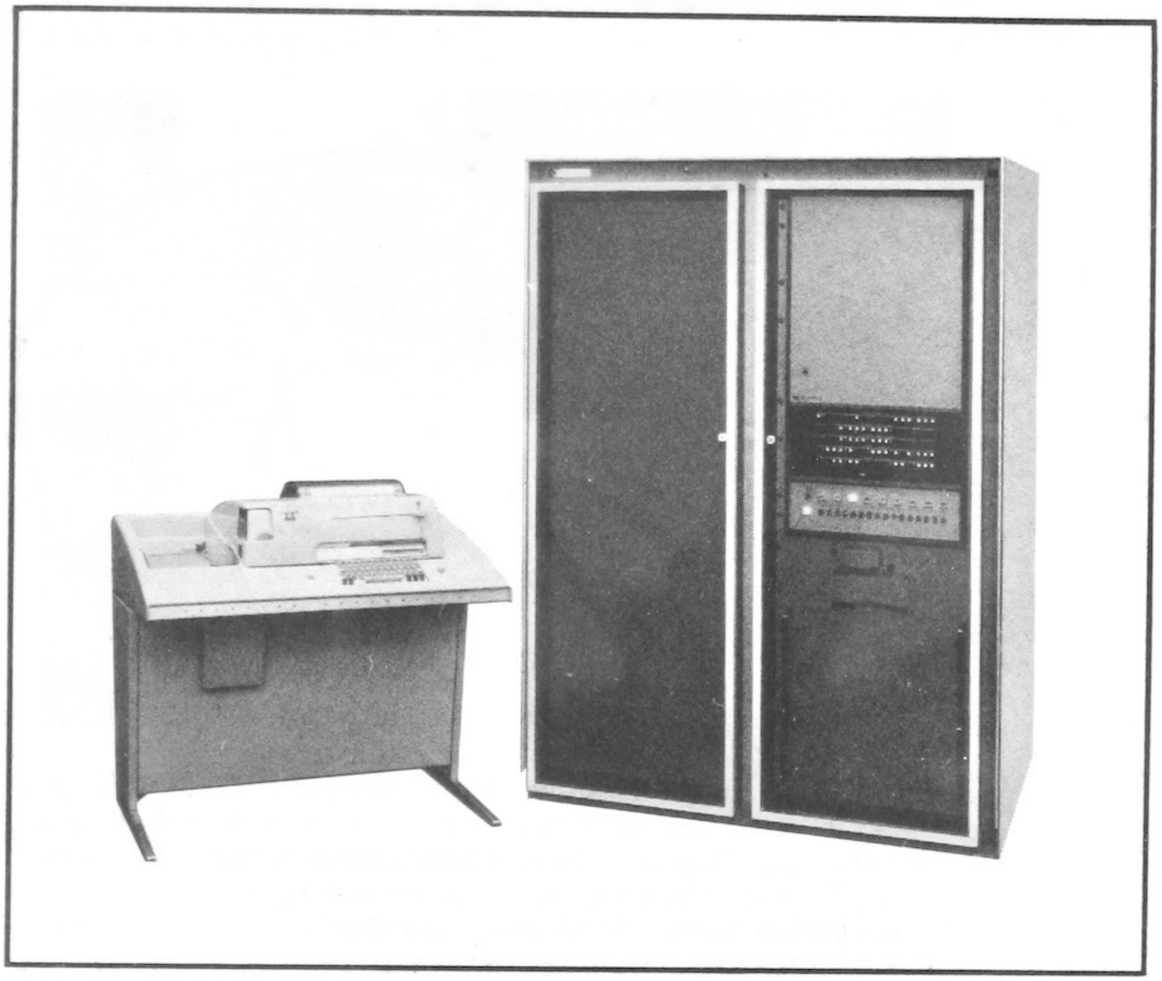
2116



2115



•••••••• Backed up by software which is compatible to all three computers.



Illustrative of HP 2116 Computer power is its use in HP Series 2000 Time Sharing Systems. Here, up to 32 users may communicate with the computer simultaneously, in conversational BASIC language.

CONTENTS

Specifications and Basic Operation Manual

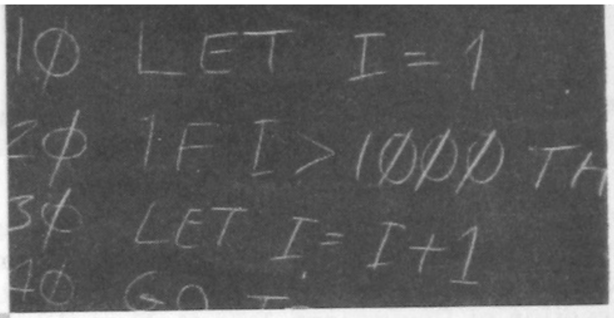
Assembler Reference Manual

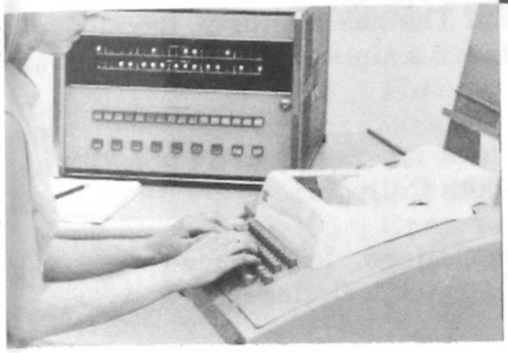
Basic Control System Reference Manual

FORTRAN Reference Manual

Program Library Reference Manual

BASIC Language Reference Manual


```
1Ø LET I=1
2Ø IF I>1ØØØ TH
3Ø LET I=I+1
4Ø GO T
```



BASIC Language Reference Manual

The development of the BASIC Language and the original version of this manual on BASIC were supported in part by the National Science Foundation under terms of Grant NSF GE 3864 to Dartmouth College. Under this grant, Dartmouth College, under the direction of Professors John G. Kemeny and Thomas E. Kurtz, developed the BASIC Language and a compiler for operation under a time-sharing system in use at Dartmouth.

CONTENTS

INTRODUCTION v

CHAPTER 1 A BASIC PRIMER 1-1

- 1.1 An Example 1-1
- 1.2 Formulas 1-4
- 1.3 Loops 1-7
- 1.4 Arrays 1-10
- 1.5 Use of The System 1-12
- 1.6 Errors and Debugging 1-13
- 1.7 Summary of Elementary BASIC Statements 1-17
  - 1.7.1 LET 1-17
  - 1.7.2 READ and DATA 1-17
  - 1.7.3 PRINT 1-18
  - 1.7.4 GO TO 1-19
  - 1.7.5 IF - THEN 1-19
  - 1.7.6 FOR and NEXT 1-19
  - 1.7.7 DIM 1-20
  - 1.7.8 END 1-20

CHAPTER 2 ADVANCED BASIC 2-1

- 2.1 More About PRINT 2-1
- 2.2 Functions and DEF 2-4
  - 2.2.1 INT 2-4
  - 2.2.2 RND 2-4
  - 2.2.3 SGN 2-6
  - 2.2.4 DEF 2-6
- 2.3 GOSUB AND RETURN 2-6
- 2.4 INPUT 2-7
- 2.5 Some Miscellaneous Statements 2-8
  - 2.5.1 STOP 2-8
  - 2.5.2 REM 2-8
  - 2.5.3 RESTORE 2-9
  - 2.5.4 COM 2-9
- 2.6 Matrices 2-10
- 2.7 Error Codes 2-16

APPENDIX A USING THE HP BASIC SYSTEM A-1

APPENDIX B CALL AND WAIT STATEMENTS B-1

APPENDIX C LOGICAL OPERATORS IN THE BASIC LANGUAGE C-1

APPENDIX D BASIC SYNTAX IN BACKUS NORMAL FORM D-1

APPENDIX E PREPARE BASIC SYSTEM PROGRAM (PBS) E-1

INTRODUCTION

This publication is a reference manual for using the HP BASIC System with HP computers. It includes both the elements of the language and the information required to operate the system on the computer. The minimum configuration on which the HP BASIC Compiler will run is:

HP 2116B, 2115A, or 2114A Computer with 8196 words of memory

HP 2752A Teleprinter

WHAT IS A PROGRAM?

A program is a set of directions, or a recipe, that is used to tell a computer how to provide an answer to some problem. It usually starts with the given data as the ingredients, contains a set of instructions to be performed or carried out in a certain order, and ends up with a set of answers as the cake. And, as with ordinary cakes, if you make a mistake in your program, you will end up with something else - perhaps hash!

Any program must fulfill two requirements before it can be carried out. The first is that it must be presented in a language that is understood by the "computer". If the program is a set of instructions for solving a system of linear equations and the "computer" is an English-speaking person, the program will be presented in some combination of mathematical notation and English. If the "computer" is a French-speaking person, the program must be in his language; and if the "computer" is a high-speed digital computer, the program must be presented in a language which the computer "understands".

The second requirement for all programs is that they must be completely and precisely stated. This requirement is crucial when dealing with a digital computer which has no ability to infer what you mean - it does what you tell it to do, not what you meant to tell it.

We are, of course, talking about programs which provide numerical answers to numerical problems. It is easy to program in the English language, but such a program poses great difficulties for the computer because English is rich with ambiguities and redundancies, those qualities which make poetry possible but computing impossible. Instead, you present your program in a language which resembles ordinary mathematical notation, which has a simple vocabulary and grammer, and which permits a complete and precise specification of your program. The language you will use is BASIC which is, at the same time, precise, simple and easy to understand.

A first introduction to writing a BASIC program is given in Chapter 1. This chapter includes all that you will need to know to write a wide variety of useful and interesting programs. Chapter 2 deals with more advanced computer techniques, and the Appendices contain a variety of reference materials.

A BASIC PRIMER

1.1 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 - and then read the commentary and explanation.

```
10     READ A, B, D, E
15     LET G = A * E-B * D
20     IF G = 0 THEN 65
30     READ C, F
37     LET X = (C*E - B*F)/G
42     LET Y = (A*F - C*D)/G
55     PRINT X, Y
60     GO TO 30
65     PRINT "NO UNIQUE SOLUTION"
70     DATA 1, 2, 4
80     DATA 2, -7, 5
85     DATA 1, 3, 4, -7
90     END
```

We immediately observe several things about this sample program. First, we see that the program uses only capital letters, since the teletypewriter has only capital letters.

A second observation is that each line of the program begins with a number. These numbers are called line numbers 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 the computer. Line numbers also serve to specify the order in which the statements are to be performed by the computer. This means that you may type your program in any order. Before the program is run, the computer 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 we shall explain later.)

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

A fourth observation, not at all obvious from the program, is that spaces have no significance in BASIC, except in messages enclosed in quotation marks which are to be printed out, as in line number 65 above. Thus, spaces may be used, or not used, at will to "pretty up" 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 preface, 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 the computer encounters a READ statement while executing your program, it will cause the variables listed after the READ to be given values according to the next available numbers in the DATA statements. In the example, we read A in statement 10 and assign the value 1 to it from statement 70 and, 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, and we pick up from it the number 2 to be assigned to E.

We next go to statement 15, which is a LET statement, and first encounter a formula to be evaluated. (The asterisk "*" is obviously used to denote multiplication.) In this statement we direct the computer to compute the value of AE - BD, and to call the result G. In general, a LET statement directs the computer to set a variable equal to the formula on the right side of the equals sign. We know that if G is equal to zero, the system has no unique solution. Therefore, we next ask, in line 20, if G is equal to zero. If the computer 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", and it then 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, the computer goes on to the next statement, in this case 30. (Thus, an IF-THEN tells the computer where to go if the "IF" condition is met, but to go on to the next statement if it is not met.) The computer is now directed to read the next two entries from the DATA statements, -7 and 5, (both are in statement 80) and to assign them to C and F respectively. The computer is now ready to solve the system

$$x + 2y = -7$$
$$4x + 2y = 5.$$

In statements 37 and 42, we direct the computer 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 parentheses, only BF would be divided by G and the computer would let $X = CE - \frac{BF}{G}$.

The computer is told to print the two values computed, that of X and that of Y, in line 55. Having done this, 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, the computer is told in line 30 to take the next one and assign it to C, and the one after that to F. Thus, the computer is now ready to solve the system

$$x + 2y = 1$$
$$4x + 2y = 3.$$

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 the computer reads two more values, 4 and -7, which it finds in line 85. It then proceeds to solve the system

$$x + 2y = 4$$
$$4x + 2y = -7$$

and to print out the solutions. It is directed back again to 30, but there are no more pairs of numbers available for C and F in the DATA statements. The computer then informs you that it is out of data, printing on the paper in your teletypewriter "ERROR 56 IN LINE 30"

For a moment, 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: the computer 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, it would not do it, and the solutions would be the computer's secret. What would have happened if we had left out line 20? In this problem just solved nothing would have happened. But, if G were equal to zero, we would have set the computer the impossible task of dividing by zero, and it would tell us so, printing "ERROR 69 IN LINE 37". Had we left out statement 60, the computer 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 which we want the machine 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. We put the numbers such a distance apart so that we can later insert additional statements if we find that we have forgotten them in writing the program originally. Thus, if we find that we have left out two statements between those numbered 40 and 50, we can give them any two numbers between 40 and 50 - say 44 and 46; and in the editing and sorting process, the computer 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 sample 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 fifth 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,

```
70 DATA 1, 2, 4, 2
75 DATA -7, 5
80 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 teletypewriter.

```
READY
10 READ A, B, D, E
15 LET G = A * E - B * D
20 IF G = 0 THEN 65
30 READ C, F
37 LET X = ( C * E - B * F ) / G
42 LET Y = ( A * F - C * D ) / G
55 PRINT X, Y
60 GO TO 30
65 PRINT "NO UNIQUE SOLUTION"
70 DATA 1, 2, 4
80 DATA 2, -7, 5
85 DATA 1, 3, 4, -7
90 END
RUN
4              -5.5
 .666667        .166667
-3.66667        3.83333

ERROR 56 IN LINE  30
```

After typing the program, we type RUN followed by a CARRIAGE RETURN. Up to this point the computer stores the program and checks the form of the statements. It is this command which directs the computer to execute your program. The message out-of-data error code here may be ignored. However, in some cases it indicates an error in the program: for more details see Sec. 1.7.2.

1.2 FORMULAS

The computer can perform a great many operations; it can add, subtract, multiply, divide, extract square roots, raise a number to a power, and find the sine of a number (on an angle measured in radians), etc. - and we shall now learn how to tell the computer to perform these various operations and to perform them in the order that we want them done.

The computer 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 operations can be used to write a formula, and these are listed in the following table:

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)
↑	X ↑ 2	Raise to the power (find X^2)

We must be careful with parentheses to make sure that we group together those things which we want together. We must also understand the order in which the computer does its work. For example, if we type A + B * C ↑ D, the computer 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 $A + B{*}C^D$. 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 their 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, the computer first raises the number to the power, 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 operations of the same priority, the operations are performed from left to right.

These rules are illustrated in the previous example. The rules also tell us that the computer, faced with A - B - C, will (as usual) subtract B from A and then C from their difference; faced with A/B/C, it will divide A by B and that quotient by C. Given A ↑ B ↑ C, the computer will raise the number A to the power B and take the resulting number and raise it to the power C. If there is any question in your mind about the priority, put in more parentheses to eliminate possible ambiguities.

In addition to these five arithmetic operations, the computer can evaluate several mathematical functions. These functions are given special 3-letter English 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	
TAN (X)	Find the tangent of X	
ATN (X)	Find the arctangent of X	
EXP (X)	Find e^X	
LOG (X)	Find the natural logarithm of X (ln X)	
ABS (X)	Find the absolute value of X ($\|X\|$)	
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 Chapter 2. In place of X, we may substitute any formula or any number in parentheses following any of these formulas. For example, we may ask the computer to find $\sqrt{4 + X^3}$ by writing SQR (4 + X↑3), or the arctangent of $3X - 2e^X + 8$ by writing ATN (3 * X - 2 * EXP (X) + 8).

If, sitting at the teletypwriter, you need the value of $(\frac{5}{6})^{17}$ you can write the two-line program

```
10      PRINT (5/6) ↑ 17
20      END
```

and the computer will find the decimal form of this number and print it out in less than it took you to type the program.

Since we have mentioned numbers and variables, we should be sure that we understand how to write numbers for the computer and what variables are allowed. A <u>number</u> may be positive or negative and it may contain up to approximately seven significant digits, but it must be expressed in decimal form. For example, all of the following are numbers in BASIC: 2, -3.675, 1234567, -.7654321, and 483.4156. The following are not numbers in BASIC: 14/3 and $\sqrt{7}$. We may ask the computer to find the decimal expansion of 14/3 or $\sqrt{7}$, and to do something with the resulting number, but we may not include either in a list of DATA. We gain further flexibility by use of the letter E, which stands for "times ten to the power". Thus, we may write .001234567 in a form acceptable to the computer in any of several forms: .1234567E-2 or 1234567E-9 or 1234.56789E-6. We may write ten million as 1E7 (or 1E + 7) and 1965 as 1.965E3 (or 1.965E + 3). We do not write E7 as a number, but must write 1E7 to indicate that it is 1 that is multiplied by 10^7. All numbers are represented in the computer by two 16-bit words. The exponent and its sign are eight bits; the fraction and its sign are twenty-four bits. They have a range in magnitude of approximately 10^{-38} to 10^{38} and may assume positive, negative or zero values. If a fraction is negative the number is stored in two's complement form. A zero value is stored as all zero bits. If a number is too large, the computer prints "ERROR 65 IN LINE nn" and replaces it with the largest representable number of the same sign. If a number is too small, the computer prints "ERROR 66 IN LINE nn" and replaces it with zero.

A numerical variable in BASIC is denoted by any letter, or by any letter followed by a single digit. Thus, the computer will interpret E7 as a variable, along with A, X, N5, I0, and O1. A variable in BASIC stands for a number, usually one that is not known to the programmer at the time the program was written. Variables are given or assigned values by LET, READ, or INPUT statements. The value so assigned will not change until the next time a LET, READ, or INPUT statement is encountered with a value for that variable. However, all variables are set to "undefined" before a RUN. Thus, it is always necessary to assign a value to a variable before using the variable in a computation. Failure to do so will cause the computer to print "ERROR 50 IN LINE nn".

Six other mathematical symbols are provided for in BASIC, symbols of relation, 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 sample program in Section 1.

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 (A is less than B)
<=	A <= B	Is less than or equal to (A is less than or equal to B)
>	A > B	Is greater than (A is greater than B)
>=	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)

1.3 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. In order to write the simplest program, the one in which this portion to be repeated is written just once, we use the programming device known as a loop.

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

```
10     PRINT 1, SQR (1)
20     PRINT 2, SQR (2)
30     PRINT 3, SQR (3)

       . . . . . . .
```

```
     990     PRINT 99,  SQR (99)
    1000     PRINT 100,  SQR (100)
    1010     END
```

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

```
    10     LET X = 1
    20     PRINT X,  SQR (X)
    30     LET X = X + 1
    40     IF X < = 100 THEN 20
    50     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 the computer back to line 20. Here it 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 $4 \leq 100$ 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. The computer now receives a negative answer to the question in line 40 (X is <u>greater than</u> 100, not less than or equal to it), does not return to 20 but moves on to line 50, and ends the program. All loops contain four characteristics; initialization (line 10), the body (line 20), modification (line 30), and 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:

```
    10     FOR X = 1  TO 100
    20     PRINT X,  SQR (X)
    30     NEXT X
    50     END
```

In line 10, X is set equal to 1, and a test is set up, like that of line 40 above. Line 30 carries out two tasks: X is increased by 1, and the test is carried out 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, we could specify it by writing

```
    10     FOR X = 1  TO  100 STEP 5
```

and the computer 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 last time. Another step of 5 would take X beyond 100, so 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

```
10      FOR X = 100 TO 1 STEP -1
```

In the absence of a STEP clause, 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

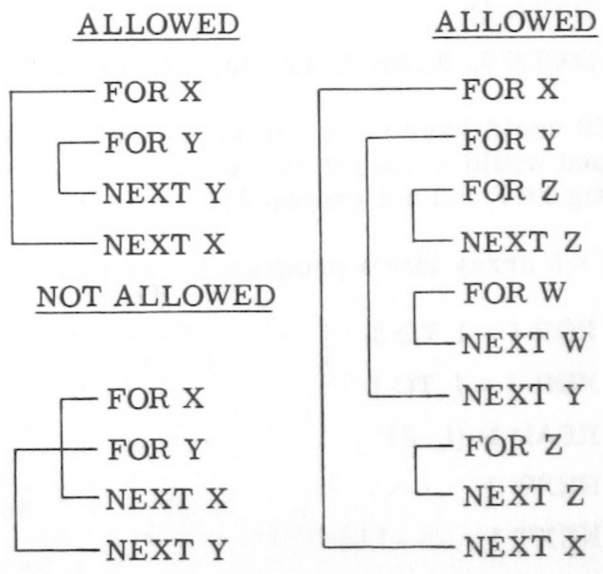
```
FOR X = N + 7 * Z TO (Z-N) /3 STEP (N-4 * Z) / 10
```

For a positive step-size, the loop continues as long as the control variable is algebraicly <u>less than or equal</u> to the final value. For a negative step-size, the loop continues as long as the control variable is <u>greater than or equal</u> 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 not be performed at all, but the computer will immediately pass to the statement following the NEXT. As an example, the following program for adding up the first n integers will give the correct result 0 when n is 0.

```
10      READ N
20      LET S = 0
30      FOR K = 1 TO N
40      LET S = S + K
50      NEXT K
60      PRINT S
70      GO TO 10
90      DATA 3, 10, 0
99      END
```

It is often useful to have loops within loops. These are called nested loops and can be expressed with FOR and NEXT statements. However, they must actually be nested and must not cross, as the following skeleton examples illustrate:



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
```

1.4 ARRAYS

In addition to the ordinary variables used by BASIC, there are variables which can be used to designate the elements of an array. These are used where we might ordinarily use a subscript or a double subscript, for example the coefficients of a polynomial $[a_0, a_1, a_2, \ldots]$ or the elements of a matrix $[b_{i,j}]$. The variables which we use in BASIC consist of a single letter, which we call the name of the array, followed by the subscripts in brackets or parentheses. Thus, we might write A [1], A [2], A [3], etc., for the coefficients of the polynomial and B [1, 1], B [1, 2], etc., for the elements of the matrix.

We can enter the array A(1), A(2), A(3), . . . A(10) into a program very simply by the lines:

```
10      FOR I = 1 TO 10
20      READ A (I)
30      NEXT I
40      DATA 2, 3,-5, 5,2.2, 4, -9, 123, 4, -4
```

We need no special instruction to the computer if no subscript greater than 10 occurs. However, if we want larger subscripts, we must use a DIM statement, to indicate to the computer that it has to save extra space for the array. When in doubt, indicate a larger dimension than you expect to use. The maximum value for a dimension is 255. For example, if we want a list of 15 numbers entered, we might write:

```
10      DIM A (25)
20      READ N
30      FOR I = 1 TO N
40      READ A (I)
50      NEXT I
60      DATA 15
70      DATA 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47
```

Statements 20 and 60 could have been eliminated by writing 30 as FOR I = 1 TO 15, but the form as typed would allow for the lengthening of the array by changing only statement 60, so long as it did not exceed 25.

We would enter a 3 × 5 array into a program by writing:

```
10      FOR I = 1 TO 3
20      FOR J = 1 TO 5
30      READ B (I, J)
40      NEXT J
50      NEXT I
60      DATA 2, 3, -5, -9, 2
```

```
70     DATA 4, -7, 3, 4, -2
80     DATA 3, -3, 5, 7, 8
```

Here again, we may enter an array with no dimension statement, and it will handle all the entries from B(1, 1) to B(10, 10). If you try to enter an array with a subscript greater than 10, without a DIM statement, you will get an error message telling you that you have a subscript error. This is easily rectified by entering the line:

```
5     DIM B (20, 30)
```

if, for instance, we need a 20-by-30 table.

The single letter denoting an array name may also be used to denote a simple variable without confusion. However, the same letter may not be used to denote both a singly subscripted and a doubly subscripted array in the same program. The form of the subscript is quite flexible, and you might have the array element B(I, K) or Q(A(3, 7), B - C).

Shown below is a list and run of a problem which uses both a singly and a doubly subscripted array. The program computes the total sales of each of five salesmen, all of whom sell the same three products. The array P gives the price/item of the three products and the array S tells how many items of each product each man sold. You can see from the program 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 array 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.

This sample program did not need a dimension statement, since the computer automatically saves enough space to allow all subscripts to run from 1 to 10. A DIM statement is normally used to save more space. But in a long program, requiring many small arrays, DIM may be used to save less space for arrays, in order to leave more for the program.

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

```
READY
10     FOR I = 1 TO 3
20        READ P(I)
30     NEXT I
40     FOR I = 1 TO 3
50        FOR J = 1 TO 5
60           READ S(I,J)
70        NEXT J
80     NEXT I
90     FOR J = 1 TO 5
100       LET S = 0
110       FOR I = 1 TO 3
120          LET S = S + P(I)*S(I,J)
130       NEXT I
140       PRINT "TOTAL SALES FOR SALESMAN "J, "$" S
```

```
150   NEXT J
900   DATA 1.25, 4.30, 2.50
910   DATA 40, 20, 37, 29, 42
920   DATA 10, 16, 3, 21, 8
930   DATA 35, 47, 29, 16, 33
999   END
RUN
TOTAL SALES FOR SALESMAN  1                    $ 180.5
TOTAL SALES FOR SALESMAN  2                    $ 211.3
TOTAL SALES FOR SALESMAN  3                    $ 131.65
TOTAL SALES FOR SALESMAN  4                    $ 166.55
TOTAL SALES FOR SALESMAN  5                    $ 169.4
```

1.5 USE OF THE SYSTEM

Now that we know something about writing a program in BASIC, how do we set about using a teletypewriter to type in our program and then to have the computer solve our problem?

First, load the BASIC system tape and initiate its execution. The computer then types READY and you should begin to type your program. Make sure that each line begins with a line number which contains no more than four digits and contains no non-digit characters. Be sure to press the CARRIAGE RETURN key at the completion of each line. Spaces may be inserted at any point in the line, including before the line numbers.

If, in the process of typing a statement, you make a typing error and notice it immediately, you can correct it by pressing the backward arrow (shift key above the letter "oh"). This will delete the character in the preceding space, and you can then type in the correct character. Pressing this key a number of times will erase from this line the characters in that number of preceding spaces. To delete all of the present line, press ALT MODE. † Programs or data may be annotated by typing the remark and then deleting the line (as far as the system is concerned) with an ALT MODE. BASIC types a slash to show that a line has been deleted.

After typing your complete program, you type RUN, press the CARRIAGE RETURN key, and hope. If the program is one which the computer can run, it will then run it and type out any results for which you have asked in your PRINT statements. This does not mean that your program is correct, but that it has no errors of the type known as "grammatical errors". If it had errors of this type, the computer would have typed an error code as soon as the error was detected during the typing of the program. Errors detected after RUN are structural (loop nesting, matching GOSUB and Return) or arithmetical errors. A list of the error codes is in Sect. 2.7 together with the interpretation of each.

If you are given an error message, you can correct the error by typing a new line with the correct statement. If you want to eliminate the statement on line 110 from your program, you can do this by typing 110 and then the CARRIAGE RETURN. If you want to insert a statement between those on lines 60 and 70, you can do this by giving it a line number between 60 and 70.

† Use ESC key if no ALT MODE key.

If it is obvious to you that you are getting the wrong answers to your problem, even while the computer is running, you can type STOP and the computation will cease. If the teletypewriter is actually typing, there is an express stop - just press any key. It will type STOP and then READY and you can start to make your corrections. If you are in serious trouble, use the express stop, and type SCRATCH. When the system is ready to accept a new program, READY will be typed.

A sample use of the system is shown below.

```
READY
10 FOR N = 1 TO 7
20 PRINT N, SQR(N)
30 NEXT N
40 PRINT "DONE"
50 END
RUN
 1              1
 2              1.41421
 3              1.73205
 4              2
 5              2.23607
 6              2.44949
 7              2.64575
DONE
```

1.6 ERRORS AND DEBUGGING

It may occasionally happen that the first run of a new problem will be free of errors and give the correct answers. 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 grammatical errors) which prevent the running of the program; and logical errors in the program which cause the computer to produce wrong answers or no answers at all.

Errors of form will cause error codes to be printed, and the various error codes are listed and explained in Sec. 2.7. 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 CARRIAGE 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 - whenever you notice them - either before or after a run. Since the computer sorts lines out and arranges them in order, a line may be retyped out of sequence. Simply retype the offending line with its original line number.

As with most problems in computing, we can best illustrate the process of finding the errors (or "bugs") in a program, and correcting (or "debugging") it, by an example. Let us consider the problem of finding that value of X between 0 and 3 for which the sine of X is a maximum, and ask the machine to print out this value of X and the value of its sine. If you have studied trigonometry, you know that $\pi/2$ is the correct value; but we shall use the computer to test successive values of X from 0 to 3, first using intervals of .1, then of .01, and finally of .001. Thus, we shall ask the computer to find the sine of 0, of .1, of .2, of .3..., of 2.8, of 2.9, and of 3, and to determine which of these 31 values is the largest. It will do it by testing SIN(0) and SIN(.1) to see which is larger, and calling the larger of these two numbers M. Then it will pick the larger of M and SIN(.2) and call it M. This number will be checked against SIN(.3), and so on down the line. Each time a larger value of M is found, the value of X is "remembered" in X0. When it finishes, M will have been assigned to the largest value. It will then repeat the search, this time checking the 301 numbers 0, .01, .02, .03,..., 2.98, 2.99, and 3, finding the sine of each and checking to see which has the largest sine. At the end of each of these three searches, we want the computer to print three numbers: the value X0 which has the largest sine, the sine of that number, and the interval of search.

Before going to the teletypewriter, we write a program and let us assume that it is the following:

```
 10    READ D
 20    LET X0 = 0
 30    FOR X = 0 TO 3 STEP D
 40    IF SIN (X) < = M THEN 100
 50    LET X0 = X
 60    LET M = SIN (X0)
 70    PRINT X0, X, D
 80    NEXT X0
 90    GO TO 20
100    DATA .1, .01..001
110    END
```

We shall list the entire sequence on the teletypewriter and make explanatory comments.

```
READY
1  REM PROGRAM NAME:  MAXSIN
10 READ D
20 LWR X0= 0
ERROR 4 IN LINE  20

20 LET X0= 0
30 FOR X = 0 TO 3 STEP D
40 IF SINE←(X) <= M THEN 100
50 LET X0=X
60 LET M = SIN(X)
70 PRINT XO, X, D
ERROR 21 IN LINE  70

70 PRINT X0, X, D
80 NEXT Z←X0
90 GO TO 20
100 DATA .1, .01, .001
```

```
110 END
RUN

ERROR 41 IN LINE  80
```

A message indicates that LET was mistyped in line 20, so we retype it, this time correctly.

Notice the use of the backwards arrow to erase a character in line 40, which should have started IF SIN (X) etc., and in line 80.

After receiving the second error message, we find that we used XO for a variable instead of X0 in line 70. The next error message indicates a FOR statement without a NEXT. Upon checking we see that the variable in the FOR and NEXT are different, so we correct statement 80. In looking over the program, we also notice that the IF-THEN statement in 40 directed the computer to a DATA statement and not to line 80 where it should go.

```
80 NEXT X
40 IF SIN(X) <= M THEN 80
RUN

ERROR 50 IN LINE  40
```

The message indicates that M has never been assigned an initial value. We decide to give it a value less than the maximum value of the sine, say -1.

```
20 LET M= -1
 RUN
 0                  0                   .1
 .1                 .1                  .1
 .2                 .2                  .1
 .3                 .3                  .1
 .4      STOP

READY
```

This is incorrect. We are having every value of X0, X, and the interval size printed, so we direct the machine to cease operations by typing S even while it is running. Note that the 'S' does not print, but the word STOP is printed.

We fix this by moving the PRINT statement outside the loop. Typing 70 deleted that line, and line 85 is outside of the loop. We also realize that we want M printed and not X.

```
70
85 PRINT X0, M, D
RUN
 1.6               .999574              .1
 1.6               .999574              .1
 1.6               .999574              .1
 1.6        STOP

READY
```

Of course, line 90 sent us back to line 20 to repeat the operation and not back to line 10 to pick up a new value for D. We also decide to put in headings for our columns by a PRINT statement.

```
90 GO TO 10
5  PRINT "X VALUE", "SIN", RESOLUTION"
ERROR 21 IN LINE  5
```

There is an error in our PRINT statement: no left quotation mark for the third item

Retype line 5, with all of the required quotation marks.

```
5  PRINT "X VALUE", "SIN", RESOLUT\
5  PRINT "X VALUE", "SIN", "RESOLUTION"
RUN
X VALUE          SIN              RESOLUTION
 1.6             .999574           .1
 1.57            1.                1.00000E-02
 1.57098         1                 1.00000E-03

ERROR 56 IN LINE  10
```

Exactly the desired results. Of the 31 numbers (0, .1, .2, .3, . . . , 2.8, 2.9, 3) it is 1.6 which has the largest sine, namely .999574. Similarly for finer subdivisions.

Having changed so many parts of the program, we ask for a list of the corrected program. Listing the corrected program, from time to time, is an important part of debugging.

```
LIST
1  REM PROGRAM NAME:  MAXSIN
5  PRINT "X VALUE","SIN","RESOLUTION"
10  READ D
20  LET M=-1
30  FOR X= 0 TO  3 STEP D
40  IF SIN(X) <= M THEN 80
50  LET X0=X
60  LET M=SIN(X)
80  NEXT X
85  PRINT X0,M,D
90  GOTO 10
100  DATA  .1         , 1.00000E-02, 1.00000E-03
110  END

PLIST
```

The program is saved for later use by punching it on the Tape Punch.

In solving this problem, there is a common device which we did not use, namely the insertion of a PRINT statement when we wonder if the machine is computing what we think we asked it to compute. For example, if we wondered about M, we could have inserted 65 PRINT M, and we would have seen the values.

1.7 SUMMARY OF ELEMENTARY BASIC STATEMENTS

In this section we shall give a short and concise description of each of the types of BASIC statements discussed earlier in this chapter and add one statement to our list. In each form, we shall assume a line number, and shall use brackets to denote a general type. Thus, [variable] refers to any variable, which is a single letter, possibly followed by a single digit.

1.7.1 LET

This statement is not a statement of algebraic equality, but rather a command to the computer to perform certain computations and to assign the answer to a certain variable. Each LET statement is of the form: LET [variable] = [formula]. More generally several variables may be assigned the same value by a single LET statement.

Examples: (of the first type):

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

(of the second type):

```
50 LET X = Y3 = A(3, 1) = 1
90 LET W = Z = 3*X -4* X ↑ 2
```

1.7.2 READ and DATA

We use a READ statement to assign to the listed variables values obtained from a DATA statement. Neither statement is used without one of 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, the computer 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 we get an out-of-data error code.

Since we have to read in data before we can work with it, 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 of the form:
DATA [sequence of numbers]

Examples:
```
150 READ X, Y, Z, X1, Y2, Q9
330 DATA 4, 2, 1.7
340 DATA 6.734E-3, -174.321, 3.14159265

234 READ B (K)
263 DATA 2, 3, 5, 7, 9, 11, 10, 8, 6, 4
```

```
10 READ R (I, J)
440 DATA -3, 5, -9, 2.37, 2.9876, -437.234E-5
450 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.

1.7.3 PRINT

The PRINT statement has a number of different uses and is discussed in more detail in Chapter 2. The common uses are (a) to print out the result of some computations, (b) to print out verbatim a message included in the program, (c) a combination of the two, and (d) to skip a line. We have seen examples of only the first two in our sample programs. Each type is slightly different in form, but all start with PRINT after the line number.

Examples of type (a):

```
100 PRINT X, SQR (X)

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

The first will print X and then, a few spaces to the right of that number, its square root. The second will print five different numbers:

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

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

Examples of type (b):

```
100 PRINT "NO UNIQUE SOLUTION"
430 PRINT "X VALUE","SINE","RESOLUTION"
```

Both have been encountered in the sample programs. The first prints that simple statement; the second prints the three labels with spaces between them. The labels in 430 automatically line up with three numbers called for in a PRINT statement (as long as the labels do not exceed 14 characters) as seen in MAXSIN.

Examples of type (c):

```
150 PRINT "THE VALUE OF X IS" X
 30 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.

Examples of type (d):

```
250 PRINT
```

The computer will advance the paper one line when it encounters this command.

1.7.4 GO TO

There are times in a program when you do not want all commands executed in the program. An example of this occurs in the MAXSIN problem where the computer has computed X0, M, and D and printed them out in line 85. We did not want the program to go to the END statement yet, but to go through the same process for a different value of D. So we directed the computer to go back to line 10 with a GO TO statement. Each is in the form of GO TO [line number]. (It is possible to go to a non-executable statement; control passes to the next sequential executable statement.)

Example: 150 GO TO 75

1.7.5 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. Such a statement occurred at line 40 of MAXSIN. The more common form of the statement is:

IF [formula] [relation] [formula] THEN [line number]

Examples:

```
40 IF SIN   (X) < = M THEN 80

20 IF G = 0 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, the computer will go to the next line of the program.

In the general sense, the IF-THEN statement may have the form:.

IF [formula] THEN [line number]

Examples:

```
17 IF G = 0 THEN 77

35 IF A THEN 83

85 IF A + B - 5 THEN 302

90 IF -2 THEN 200
```

The formula is evaluated and the result interpreted as non-zero (true) or zero (false). Thus, in the first case, if G had a value of 2, the formula G = 0 is evaluated as zero or false. If A would have a value of 2 (in the second case), the formula is evaluated as non-zero or true. In the third case, if A equalled 2 and B equalled 3, the formula would yield a value of zero (false). The fourth example would always be true.

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

1.7.6 FOR and NEXT

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

Any simple (not subscripted) variable may be used as the FOR variable. 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].

```
Examples:     30 FOR X = 0 TO 3 STEP D
              80 NEXT X
              120 FOR X4 = (17 + COS(Z))/3 TO 3*SQR(10) STEP 1/4
              235 NEXT X4
              240 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 the next example, the successive values of X will be 8, 7, 6, 5, 4, 3. In the last example, on successive trips through the loop, J will take on values -3, -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 not be performed and the computer will proceed to the statement immediately following the corresponding NEXT statement.

1.7.7 DIM

Whenever we want to enter an array with a subscript greater than 10, we must use a DIM statement to inform the computer to save us sufficient room.

```
Examples:     20 DIM H(35)
              35 DIM Q(5, 25)
```

The first would enable us to enter an array of 35 items, and the latter a 5 × 25 array.

1.7.8 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.

```
Example:      999 END
```

2.1 MORE ABOUT PRINT

The uses of the PRINT statement were described in 1.7.3, but we shall give more detail here. 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 teletypewriter line is divided into five zones starting at positions 0, 15, 30, 45, and 60. 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.

More compact output can be obtained by use of the semi-colon; it inhibits spacing between print zones on a line, acting only to separate quantities to be printed (e.g., A + B; C/D) or suppress a CARRIAGE RETURN at the end of a PRINT statement.

Spacing within a print zone depends on the value and type of the number being printed. A number is always printed in a zone larger than it needs and is left-justified in the zone. The size is determined as follows:

Value of Number	Type of Number	Format of Zone
$-999 \leq n \leq +999$	Integer	△xxx∧∧† only 6 long
$-32768 \leq n \leq -1000$ $+1000 \leq n \leq +32767$	Integer	△xxxxx∧∧∧
$.1 \leq n \leq 999999.5$	Large Integer or Real	△xxxxxxx∧∧∧∧ only 6 digits and decimal point. (Decimal point printed as one of x's; trailing zeros suppressed.)
$n < .1$ $999999.5 < n$	Large Integer or Real	△x.xxxxxE±ee∧∧∧

For Example, if you were to type the program

```
10 FOR I = 1 TO 15
20 PRINT I
30 NEXT I
40 END
RUN
```

†The carot symbol, ∧, represents a space typed on the teletypewriter.

the teletypewriter 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 changing line 20 to read

```
    20 PRINT I,
     RUN
```

you would have the numbers printed in the zones, reading

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

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

```
        20 PRINT I;
        RUN
```

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 semi-colon 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, you will remember that

250 PRINT

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

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

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

The instructions

```
50 PRINT " HP BASIC ";
51 PRINT "LANGUAGE COMPILER"
90 END
RUN
```

will result in the printing of

```
HP BASIC LANGUAGE COMPILER
```

Formatting of output can be controlled even further by use of the function TAB.

Insertion of TAB(17) will cause the teletypewriter to move to column 17, just as if a tab had been set there. For this purpose the positions on a line are numbered from 0 through 71.

More precisely, TAB may contain any formula as its argument. The value of the formula is computed, and its integer part is taken. (If the result is greater than 71, the teletypewriter is moved to position zero of the next line.) The teletypewriter is then moved forward to this position – unless it has already passed this position, in which case the TAB is ignored.

For example, inserting the following line in a loop:

PRINT X; TAB(12); Y; TAB(27); Z

will cause the X-value to start in column 0, the Y-value in column 12 and the Z-value in column 27.

A comma following a TAB clause has no effect on positioning the teletypewriter. For example, the statement "PRINT TAB (7), A+B" causes the value of A+B to be printed starting at position 7, while the statement "PRINT Z, A+B" causes the value to be printed starting at position 15.

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

1. If a number is an integer with a value between -32768 and +32767, inclusive, the decimal point is not printed.

2. If the number is an integer out of the above range (see INT function), or if the number is real and has an absolute value between .1 and 999999.5, the number is rounded to six digits and printed with a decimal point. Zeros trailing the decimal point are suppressed.

3. If a number is either greater than 999999.5 or less than .1 in magnitude, it is rounded to six places; the teletypewriter then prints a space (if positive) or minus sign (if negative), the first digit, the decimal point, the next five digits, the letter E (indicating exponent), the sign of the exponent, and the exponent. For example, it will take 32, 437, 580, 259 and write it as 3.24376E + 10.

The following program, in which we print out powers of 2, shows how numbers are printed.

```
READY
1Ø FOR N = -5 TO 3Ø
2Ø PRINT 2↑N;
3Ø NEXT N
4Ø END
RUN
 3.125ØØE-Ø2    6.25ØØØE-Ø2     .125          .25          .5           1
2      4     8     16    32    64    128   256   512   1Ø24     2Ø48
 4Ø96     8192     16384    32768.      65536.      131Ø72.     262144.
524288.      1.Ø4858E+Ø6    2.Ø9715E+Ø6     4.1943ØE+Ø6    8.38861E+Ø6
1.67772E+Ø7     3.35544E+Ø7    6.71Ø89E+Ø7     1.34218E+Ø8    2.68435E+Ø8
 5.36871E+Ø8    1.Ø7374E+Ø9
```

2.2 FUNCTIONS AND DEF

Three functions were listed in Section 1.2 but not described: INT, RND and SGN.

2.2.1 INT

The INT function is the function which frequently appears in algebraic computation as [x], and it gives the greatest integer not greater than x for $-32768 \leq x < 32768$. Thus INT (2.35) = 2, INT (-2.35) = -3, and INT (12) = 12. INT (x) = 32767 for $x \geq 32768$ and INT (x) = -32768 for $x \leq -32768$.

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 (3.4) = 3. You should convince yourself that this will indeed do the rounding guaranteed for it (it will round a number midway between two integers up to the larger of the integers).

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

2.2.2 RND

The function RND produces a random number between 0 and 1. The form of RND requires an argument although the argument has no significance, and so we write RND(X) or RND(Ø). (The argument may be a constant or a previously defined variable.)

If we want the first twenty random numbers, we write the program below and we get twenty six-digit decimals. This is illustrated in the following program.

```
READY
10 FOR L = 1 TO 20
20 PRINT RND(0),
30 NEXT L
40 END
RUN
 1.52602E-05    .500092        .500412        1.64799E-03    6.17992E-03
 .522248        .577867        .266971        .901025        .503415
 .411261        .436832        .419642        8.63642E-02    .241408
 .171168        .35434         8.55276E-02    .824103        .674869
READY
RUN
 1.52602E-05    .500092        .500412        1.64799E-03    6.17992E-03
 .522248
```

Note that the second RUN was giving exactly the same "random" numbers as the first RUN. This greatly facilitates the debugging of programs that use the random number generator.

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

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

and we would then obtain

```
0              5              5              0              0
5              5              2              9              5
4              4              4              0              2
1              3              0              8              6
```

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

```
20 PRINT INT(9*RND(0) + 1);
RUN
 1    5    5    1    1    5    6    3    9    5    4    4
 4    1    3    2    4    1    8    7
```

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

```
20 PRINT INT(20*RND(0) + 5);
RUN
 5    15   15   5    5    15   16   10   23   15   13   13
 13   6    9    8    12   6    21   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(0) + B).

2.2.3 SGN

The SGN function is one which 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.

2.2.4 DEF

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 are FN. Hence, you may define up to 26 functions, e. g., FNA, FNB, etc.

The handiness of such a function can be seen in a program where you frequently need the function $e^{-x^2} + 5$. You would introduce the function by the line

```
30 DEF FNE(X) = EXP(-X ↑ 2 + 5)
```

and later on call for various values of the function by FNE(. 1), FNE(3. 45), FNE (A+2), etc. Such definition can be a great time-saver when you want values of some function for a number of different values of the variable.

The DEF statement may occur anywhere in the program, and the expression to the right of the equal sign may be any formula which can be fitted onto one line. It may include any combination of other functions, including ones defined by different DEF statements, and it can involve other variables besides the ones denoting the argument of the function. Assuming FNR is defined by

```
70 DEF FNR(X) = SQR(2 + LOG(X) - EXP(Y*Z)*(X + SIN(2*Z)))
```

and you have previously assigned values to Y and Z, you can ask for FNR(2. 7). You can give new values to Y and Z before the next use of FNR.

The use of DEF is limited to those functions whose value may be computed within a single BASIC statement. However, there are often much more complicated functions or portions of a program which must be calculated at several points within the program. For these operations, the GOSUB statement may be used.

2.3 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,

```
90 GOSUB 210
```

directs the computer to jump to line 210, the first line of the subroutine. The logical end of the subroutine should be a return command directing the computer to return to the earlier part of the program. For example,

```
350 RETURN
```

will tell the computer 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 and their 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". GOSUBS may be nested nine deep. 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 if a subroutine will not work properly. You may have several RETURNs in the subroutine so long as exactly one of them will be used.

```
READY
10 PRINT " A", " B", " C", "GCD"
20 READ A, B, C
30 LET X = A
40 LET Y = B
50 GOSUB 200
60 LET X = G
70 LET Y = C
80 GOSUB 200
90 PRINT A, B, C, G
100 GO TO 20
110 DATA 60, 90, 120
120 DATA 38456, 64872, 98765
130 DATA 32, 384, 72
200 LET Q = INT(X/Y)
210 LET R = X - Q*Y
220 IF R = 0 THEN 300
230 LET X = Y
240 LET Y = R
250 GO TO 200
300 LET G = Y
310 RETURN
 320 END
RUN
 A              B              C              GCD
 60             90             120             30
 38456.         64872.         98765.          1
 32             384            72              8

ERROR 56 IN LINE  20
```

2.4 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 machine'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

40 INPUT X, Y

before the first statement which is to use either of these numbers. When it encounters this statement, the computer will type a question mark and wait. The user types two numbers, separated by a comma, presses the CARRIAGE RETURN key and the computer 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

```
20 PRINT "YOUR VALUES OF X, Y, AND Z ARE";
30 INPUT X, Y, Z
```

and the machine 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. Furthermore, it may take a long time to enter a large amount of data using INPUT. Therefore, INPUT should be used only when small amounts of data are to be entered, or when it is necessary to enter data during the running of the program such as with game-playing programs.

2.5 SOME MISCELLANEOUS STATEMENTS

Several other BASIC statements that may be useful from time to time are STOP, REM and RESTORE. The COM statement, a special feature of HP BASIC, permits transfer of data between programs.

2.5.1 STOP

STOP is entirely equivalent to GO TO xxxx, where xxxx 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 exactly equivalent.

```
250 GO TO 999        250 STOP
    .........            ....
340 GO TO 999        340 STOP
    .........            ....
999 END              999 END
```

2.5.2 REM

REM provides a means for inserting explanatory remarks in a program. The computer 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 GO TO or IF-THEN statement.

```
100 REM INSERT DATA IN LINES 900-998.  THE FIRST
110 REM NUMBER IS N, THE NUMBER OF POINTS.  THEN
120 REM THE DATA POINTS THEMSELVES ARE ENTERED, BY

200 REM THIS IS A SUBROUTINE FOR SOLVING EQUATIONS
    .....
300 RETURN
    .....
520 GOSUB 200
```

2.5.3 RESTORE

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, the computer restores the data block pointer to the first number. A subsequent READ statement will then start reading the data all over again. A word of 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 block 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.

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

2.5.4 COM

The COM statement allows one program to store information in memory for retrieval by a subsequent program (only one BASIC program can exist in the system at a given time). The form of the statement is the same as for the DIM statement with the word COM replacing DIM, thus the common area information is accessible only as an array. Care must be taken not to dimension the same array in both a COM and DIM statement. The COM statement must be the first entered and lowest numbered statement of the program.

The common area is a block of contiguous storage elements; two computer words per element. The elements are allotted in the order that the arrays appear in the COM statement; the elements within an array are stored row by row. It is the programmer's responsibility to see that the portions of the common area are accessed properly. Assuming that one program starts with the statement "1 COM A(10), B(3, 3)" and another with "1 COM C(5), D(5), F(3, 3)", the common area storage elements would be assigned as follows:

Element Position	First Program Reference	Second Program Reference
1	A(1)	C(1)
2	A(2)	C(2)
3	A(3)	C(3)
4	A(4)	C(4)
5	A(5)	C(5)
6	A(6)	D(1)
7	A(7)	D(2)
8	A(8)	D(3)
9	A(9)	D(4)
10	A(10)	D(5)
11	B(1, 1)	F(1, 1)
12	B(1, 2)	F(1, 2)
13	B(1, 3)	F(1, 3)
14	B(2, 1)	F(2, 1)
15	B(2, 2)	F(2, 2)
16	B(2, 3)	F(2, 3)
17	B(3, 1)	F(3, 1)
18	B(3, 2)	F(3, 2)
19	B(3, 3)	F(3, 3)

A reference in the first program to B(1, 1) would access the same element as a reference to F(1, 1) in the second program. If A contained only 9 elements, however, the B(1, 1) and F(1, 1) references would access different elements.

The length of the common area may vary between programs, but for any two programs, information may be transferred only via the portion which is common to both.

If the first program declares "1 COM A(10), B(5, 5)" and the succeeding program contains "1 COM D(10), E(5, 5), F(10)", the values of F would be unpredictable. If the second program contained "1 COM D(10)" only, the contents of B would be destroyed.

2.6 MATRICES

A common and convenient interpretation of doubly subscripted arrays is as matrices. Although you can work out for yourself programs which involve matrix computations, there is a special set of twelve instructions for such computations. They are identified by the fact that each instruction must start with the word 'MAT'. They are

MAT READ A, B, C	Read the three matrices, their dimensions having been previously specified. Data is stored in the matrix row by row.
MAT C = ZER	Fill out C with zeros.
MAT C = CON	Fill out C with ones.
MAT C = IDN	Set up C as an identity matrix.
MAT PRINT A, B;C	Print the three matrices, with A and C in the regular format, but B closely packed.
MAT B = A	Set the matrix B equal to the matrix A.
MAT C = A + B	Add the two matrices A and B.
MAT C = A - B	Subtract the matrix B from the matrix A.
MAT C = A*B	Multiply the matrix A by the matrix B.
MAT C = TRN(A)	Transpose the matrix A.
MAT C = (K)*A	Multiply the matrix A by K. K, which must be in parentheses, may be a formula.
MAT C = INV(A)	Invert the matrix A.

These twelve statements, with the addition of DIM, make matrix computations easier, and in combination with the ordinary BASIC instructions make the language much more powerful. However, the user has to be careful to keep (and to understand) the conventions "built into" the language. We will discuss, below, the individual MAT statements.

The following convention has been adopted for MAT: If in a MAT instruction we have a matrix of dimension M-by-N, the rows are numbered 1, 2, . . . , M, and the columns 1, 2, . . . , N.

The DIM statement may simply indicate what the maximum dimension is to be. Thus, if we write

```
DIM M(20, 35)
```

then M may have up to 20 rows and up to 35 columns. This statement is to save enough space for the matrix, and hence, the only care at this point is that the dimensions declared are large enough to accommodate the matrix. (The assumed dimensions of 10 rows by 10 columns do not apply for a matrix involved in matrix computations.)

The actual dimensions of a matrix are established either when it is set up (by a DIM statement) or when they are defined in one of four MAT statements: MAT READ, MAT – ZER, MAT – CON, or MAT – IDN.
Thus,

```
10 DIM M(20, 7)
----
50 MAT READ M
```

will read a 20-by-7 matrix for M, while

```
50 MAT READ M(17, 30)
```

will read 17-by-30 matrix for M, provided sufficient space has been saved for it by writing, for example,

```
10 DIM M(20, 35).
```

The elements of a matrix are stored by row in ascending locations in memory; two computer words are used for each element. A matrix declared as DIM A(3, 3) will be structured as:

	Columns		
	A(1, 1)	A(1, 2)	A(1, 3)
Rows	A(2, 1)	A(2, 2)	A(2, 3)
	A(3, 1)	A(3, 2)	A(3, 3)

The elements would be stored in the following order:

Element Position	Memory Location	Element
1	m	A(1, 1)
2	m+2	A(1, 2)
3	m+4	A(1, 3)
4	m+6	A(2, 1)
5	m+8	A(2, 2)
6	m+10	A(2, 3)
7	m+12	A(3, 1)
8	m+14	A(3, 2)
9	m+16	A(3, 3)

Given a statement DIM A(M, N), the location of element A(i, j) with respect to the first element, A(1, 1), of the matrix is given by the equation:

$$\text{location } A(1, 1) + 2(M(i-1) + (j-1))$$

The three instructions:

```
MAT M = ZER
MAT M = CON
MAT M = IDN
```

which set up a matrix M with all components zero, all components equal to one, and as an identity matrix, respectively, act like MAT READ as far as the dimension of the resulting matrix is concerned. For example,

```
        MAT M = CON(7, 3)
```

sets up a 7-by-3 matrix with 1 in every component, while

```
        MAT M = CON
```

sets up a matrix, with ones in every component, according to previously specified dimensions. Thus,

```
        10 DIM M(20, 7)
        20 MAT READ M(7, 3)
        .......
        35 MAT M = CON
        .......
        70 MAT M = ZER(15, 7)
        .......
        90 MAT M = ZER(16, 10)
```

will first read in a 7-by-3 matrix for M. Then it will set up a 7-by-3 matrix of all ones for M (the actual dimension having been set up as 7-by-3 in line 20.) Next it will set up M as a 15-by- 7 all zero matrix. (Note that although this is larger than the previous M, it is within the limits set in 10.) But it will result in an error code in line 90. The limit set in line 10 is 140 components, and in 90 we are calling for 160 components. The original dimensions may be exceeded, providing the total number of components is within the proper limit;

```
        90 MAT M = ZER(25, 5)
```

would not yield an error message.

The MAT PRINT statement causes the array elements to be printed row by row across the page. The spacing between row elements is controlled by the use of , and ; in the same manner as for the PRINT statement. Rows containing more elements than can be printed on a line are continued on consecutive lines. Each row is started on a new line and is speared from the previous row by a blank line. Thus the instruction

```
        MAT PRINT A, B; C
```

will cause the three matrices to be printed A and C with five components to a line and B with up to twelve.

Singly subscripted arrays may be interpreted as column nestors. Vectors may be used in place of matrices, as long as the above rules are observed. Since a vector like V(I) is treated as a column vector by BASIC, a row vector has to be introduced as a matrix that has only one row, namely row 1. Thus

```
        DIM X(7),  Y (1, 5)
```

introduces a 7-component column vector and a 5-component row vector.

A column vector will be printed one element to the line with double spacing between lines. A row vector will be printed in the manner indicated by the form of the statement. For example: if V is a row vector then, "MAT PRINT V" will print the vector V a single element to the line; "MAT PRINT V, " will print V as a row vector, five numbers to the line, while

```
MAT PRINT V;
```

will print V as a row vector with up to twelve numbers to the line

```
MAT B = A
```

This sets B up to be the same as A provided the dimensions previously assigned to B are the same as those of A.

```
MAT C = A + B,  MAT C = A - B
```

For these to be legal A, B, and C must have the same dimensions. Instructions MAT A = A±B are legal – the indicated operation is performed and the answer stored in A. Only a single arithmetic operation is allowed so MAT D = A + B - C is illegal but may be achieved with two MAT instructions.

```
MAT C = A * B
```

For this to be legal it is necessary that the number of columns in A be equal to the number of rows in B, the number of rows in C be equal to the number of rows in A, and the number of columns in C be equal to the number of columns in B. For example, if A has dimension L-by-M and B has dimension M-by-N then C = A * B must have dimension L-by-N. It should be noted that while MAT A = A + B may be legal, MAT A = A * B will result in nonsense. There is good reason for this. In adding two matrices we may immediately store the answer in one of the matrices; but if we attempt to do this in multiplying two matrices, we will destroy components which would be needed to complete the computation. MAT B = A * A is, of course, legal provided A is a 'square' matrix.

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

This lets C be the transpose of the matrix A. If A is an M-by-N matrix C must be an N-by-M matrix.

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

The matrix C is the matrix A multiplied by the value of K (i.e., each component of A is multiplied by K to form the components of C). K, which must be in parentheses, may be a number or a formula. MAT A = (K) * A is legal.

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

The number C is the inverse of A. (A must, of course, be a 'square' matrix.) A matrix must not be inverted or transposed into itself.

We close this section with two illustrations of matrix programs. The first one reads in A and B in line 30 and in so doing sets up the correct dimensions. Dimensions for C are set up in line 35. Then, in line 40, A + A is computed and the answer is called C. Note that the data in line 90 results in A being 2-by-3 and B being 3-by-3. Both MAT PRINT formats are illustrated, and one method of labeling a matrix print is shown.

```
READY
10 DIM A(15,15), B(15,15), C(15,15)
20 READ M,N
30 MAT READ A(M,N), B(N,N)
35 MAT C = ZER(M,N)
40 MAT C = A + A
50 MAT PRINT C;
60 MAT C = A*B
70 PRINT
75 PRINT "A*B ="
80 MAT PRINT C,
90 DATA 2, 3
91 DATA 1, 2, 3
92 DATA 4, 5, 6
93 DATA 1, 0, -1
94 DATA 0, -1, -1
95 DATA -1, 0, 0
99 END
RUN
 2     4     6

 8     10    12


A*B =
-2                -2                -3

-2                -5                -9
```

The second example inverts an N-by-N Hilbert Matrix

$$\begin{array}{cccccc}
1 & 1/2 & 1/3 & . & . & . & 1/N \\
1/2 & 1/3 & 1/4 & . & . & . & 1/N+1 \\
1/3 & 1/4 & 1/5 & . & . & . & 1/N+2 \\
. & . & . & . & . & . & \\
. & . & . & . & . & . & \\
. & . & . & . & . & . & \\
1/N & 1/N+1 & 1/N+2 & & & & 1/2N-1
\end{array}$$

Ordinary BASIC instructions are used to set up the matrix in lines 50 to 90. Note that this occurs after correct dimensions have been declared. Then a single instruction results in the computation of the inverse, and one more instruction prints it. In this example we have supplied 4 for N in the DATA statement and have made a run for this case.

```
READY
5   REM THIS PROGRAM INVERTS AN N-BY-N MATRIX
10  DIM A(20,20), B(20,20)
20  READ N
30  MAT A = CON(N,N)
40  MAT B = CON(N,N)
50  FOR I = 1 TO N
60  FOR J = 1 TO N
70  LET A(I,J) = 1/(I+J-1)
80  NEXT J
90  NEXT I
100 MAT B = INV(A)
110 PRINT
115 PRINT "INV(A) ="
120 PRINT
125 MAT PRINT B;
190 DATA 4
199 END
RUN

INV(A) =

 16.0019     -120.021     240.049    -140.031

-120.02       1200.22    -2700.52     1680.33

 240.048     -2700.51     6481.2     -4200.77

-140.03       1680.33    -4200.77     2800.49
```

The reader is warned that beyond N = 7 the Hilbert matrix cannot be inverted because of severe round-off errors.

2.7 ERROR CODES

An error message is printed as soon as the error condition is detected. The format is as follows:

ERROR xx IN LINE nn

Error Code	Meaning
1	Statement ends unexpectedly.
2	Input exceeds 71 characters.
3	System command not recognized. (May be missing statement number.)
4	Missing or incorrect statement type.
5	Exponent of number is missing power.
6	Symbol following MAT not recognized.
7	LET statement has no store.
8	Multiple or misplaced COM statement.

9	Missing or incorrect function identifier in DEF.
10	Missing parameter in DEF statement.
11	Missing assignment operator.
12	Missing THEN.
13	Missing or incorrect for-variable.
14	Missing TO.
15	Incorrect STEP in FOR statement.
16	Called routine does not exist.
17	Wrong number of parameters in CALL statement.
18	Missing or incorrect constant in DATA statement.
19	Missing or incorrect variable in READ statement.
20	No closing quote for PRINT string.
21	Missing print delimiter or bad PRINT quantity.
22	Illegal word follows MAT.
23	Missing delimiter.
24	Improper matrix function.
25	No subscript where expected.
26	May not invert or transpose matrix into self.
27	Missing multiplication operator.
28	Improper matrix operator.
29	Matrix may not be both operand and result of matrix multiplication.
30	Missing left parenthesis.
31	Missing right parenthesis.
32	Operand not recognized.
33	Defined array missing subscript part.
34	Missing array identifier.
35	Missing or bad integer.
36	Non -blank characters following statement's logical end.
37	Out of storage during syntax phase.
38	Punched Tape Reader not ready.
39	Doubly defined function.
40	FOR statement has no matching NEXT statement.
41	NEXT statement has no matching FOR statement.
42	Out of storage for symbol table.
43	Array appears with inconsistent dimensions.

44	Last statement is not END.
45	Array doubly dimensioned.
46	Number of dimensions not obvious.
47	Array too large.
48	Out of storage during array allocation.
49	Subscript exceeds bound.
50	Accessed operand has undefined value.
51	Non-integer power of negative number.
52	Zero to zero power.
53	Missing statement.
54	Gosubs nested 1∅ deep.
55	RETURN finds no address.
56	Out of data.
57	Out of storage during execution.
58	Dynamic array exceeds allocated storage.
59	Dimensions not compatible.
60	Matrix operand contains undefined element.
61	Singular or nearly singular matrix.
62	Trigonometric function argument is too large.
63	Attempted square root of negative argument.
64	Attempted log of negative argument.

The following errors are warning only, execution continues.

65	Numerical overflow, result taken to be + or -infinity.
66	Numerical underflow, result taken to be zero.
67	Log of zero taken to be -infinity.
68	EXP overflows, result taken to be +infinity.
69.	Division by zero, result taken to be + or -infinity.
7∅	Zero raised to negative power, result taken to be +infinity.

USING THE HP BASIC SYSTEM — A

To use the BASIC system, you load the tape into the computer, start execution of the system, and begin typing your program when the system types "READY." If desired, the program may also be entered from the Punched Tape Reader.

OPERATING INSTRUCTIONS

Load BASIC using the Basic Binary Loader.

1. Place BASIC binary tape in the Standard Input Unit (e. g., Punched Tape Reader).
2. Set Switch Register to starting address of Basic Binary Loader: 017700
3. Press LOAD ADDRESS.
4. Set Loader switch to ENABLED. (On 2114A: LOADER ENABLE to ON.)
5. Press PRESET.
6. Press RUN.
7. When the computer halts with T-Register containing 102077, the BASIC tape is loaded. Set Loader switch to PROTECTED. (2114A: LOADER ENABLE to NORMAL.)

Set Switch Register to starting address of BASIC:

000100

Press LOAD ADDRESS and Run.

When the system types READY, begin typing program. Terminate each line with a CARRIAGE RETURN. (When the system is ready to accept a new line, it issues a LINE FEED.)

If errors are made while typing the program, they can be corrected by one of the following:

1. To delete one or more characters that have just been typed, press ← for each character, then type in the correct characters.
2. To delete the current line, press ALT MODE or ESC, and type the correct line.
3. To correct a previously typed line, retype the entire line starting with the line number.
4. To delete a previously typed line, type the line number followed by a CARRIAGE RETURN.

CONTROL COMMANDS

There are several commands that may be given to the computer by typing the command at the start of a new line (no line number) and following the command with a CARRIAGE RETURN.

STOP	Stops all operations at once. When the teletypewriter is typing, depressing any key will execute the STOP command. When the system is ready to accept further input, it types "READY".
RUN	Begins the computation of a program.
SCRATCH	Destroys the problem currently being worked on; it gives the user a "clean sheet" to work on. When the system is ready to accept a new program, it types "READY."
LIST	Causes an up-to-date listing of the program to be typed out. †
LIST XXXX	Causes an up-to-date listing of the program to be typed out beginning at line number XXXX and continuing to the end. †
TAPE	Informs the system that the program will be entered through the teletype reader. As the program tape will have line feeds following carriage returns, the feedback of a line feed after each statement will be suppressed to avoid double spacing between lines of the program.
PLIST	Causes an up-to-date copy of the program with leader, trailer to be punched on the High Speed Tape Punch. This tape may be saved and reloaded to execute the program at a later date. ①
PTAPE	Causes the system to read in the program from the Punched Tape Photoreader. ②

① If this command is given to a system which does not have a High Speed Tape Punch, the program is listed on the teletype preceded and followed by blank characters. Therefore to obtain a listing for debugging purposes type LIST, to obtain a copy of the program on paper tape for future use turn on the punch device appropriate to the system and type PLIST.

② If this command is given to a system which does not have a Photoreader, the system will type $\begin{array}{c}\text{STOP}\\ \text{READY}\end{array}$ and await further input from the teletype.

† The last few characters of statements whose listing exceeds 72 characters may not show on the listing but they are not lost internally.

HP BASIC provides two additional statements for use in a program.

WAIT

The WAIT statement extends BASIC to allow the introduction of delays into a program. Execution of WAIT (formula) causes the program to wait for the number of milliseconds, up to 32767, specified by the value of the formula. This is not precise because it does not take into account the time required to evaluate the formula.

CALL

The CALL statement is a provision of HP BASIC for interfacing absolute assembly language subroutines, typically input-output drivers, which can be added to the standard system to create specialized configurations. When executing a CALL statement, BASIC transfers control to an absolute assembly language subroutine which has been appended to the standard system. Once this transfer has taken place, the subroutine is in complete control of the computer and is at liberty to alter any part of core including the system itself. To minimize the danger of destroying the system by such an alteration, only those programmers who are proficient in absolute assembly language programming should attempt to append CALL statement subroutines to the standard system.

Subroutines which have been appended to BASIC can be accessed in a BASIC program through a statement of the form:

```
CALL (<subroutine number>, <parameter list>).
```

The subroutine number is a positive integer specifying the desired subroutine, if no such numbered subroutine exists the statement will be rejected by the syntax analyzer. The parameters, separated by commas, may be any formulas and their number is dependent upon the subroutine called. As an example, suppose that a subroutine designated by 5 has been appended to the system. Its function is to take readings from an A to D subsystem and store them in an array. The parameters specify the array into which the values are to be inserted, the channel number of the first point to be measured, the setup for the A to D converter and the number of points to be measured. A representative call might be as follows:

```
20 CALL (5,  A[1],  1,  1188,  10)
         ↑   ↑      ↑    ↑     ↑
         |   |      |    |     Number of points
         |   |      |    |
         |   |      |    A to D setup
         |   |      |
         |   |      Starting channel number
         |   |
         |   First element of data array
         |
         Subroutine number
```

In using such statements, it is very important that correct parameters be supplied. Interchanging the first and second parameters could result in the destruction of the core-resident BASIC system, unless special precautions have been taken by the writer of the called subroutine.

PREPARATION OF SPECIAL SYSTEMS

Since subroutines accessed by CALL statements are a part of the BASIC system rather than a part of the user's BASIC program, they may be included on the composite tape produced by PBS. This can be accomplished by following the instructions given in Appendix E except that the absolute assembly binary tape of the subroutines is loaded between steps 2. and 3. The binary tape from step 8. replaces the normal BASIC system tape.

BASIC accesses called subroutines through a table containing linkage information. Entries in the table, one per subroutine, are two words in length. Bits 5-0 of the first word contain the number identifying the subroutine (chosen freely from 1 to 77_8 inclusive) and bits 15-8 contain the number of parameters passed to the subroutine (CALL statements with an incorrect number of parameters will be rejected by the syntax analyzer). The second word contains the absolute address of the entry point of the subroutine (control is transferred via a JSB). Although subroutine numbers need not be assigned in any particular order, all entries in the table must be contiguous. An acceptable auxiliary tape contains the following:

1) An ORG statement to origin the program at an address greater than that of the last word of the BASIC system. The address of this last word + 1 is contained in location 110_8 of the standard BASIC system. Hence, a suitable lower limit for the origin address can be determined by loading BASIC and examining location 110_8.

2) The subroutine linkage table described above.

3) The assembly language subroutines.

4) Code to set the following linkage addresses:

   a) In location 110_8 put the address of the last word + 1 used in the auxiliary tape.

   b) In location 121_8 put the address of the first word of the subroutine linkage table.

   c) In location 122_8 put the address of the last word + 1 of the subroutine linkage table.

Assume that location 110_8 of the standard BASIC system contains 13405_8. An acceptable auxiliary tape could be assembled from the following code:

```
         ORG 13405B
SBTBL    OCT 2406    Subroutine 6 has 5 parameters
         DEF SB6
         OCT 1421    Subroutine 17 has 3 parameters
         DEF SB17
```

```
ENDTB    EQU * + 1
SB6      NOP
         .
         .        Subroutine #6 body
         .
         JMP SB6, I
SB17     NOP
         .
         .        Subroutine #17 body
         .
         JMP SB17, I
LSTWD    EQU * + 1
         ORG 110B
         DEF LSTWD
         ORG 121B
         DEF SBTBL
         DEF ENDTB
         END
```

Acceptable calls to subroutines SB6 and SB17 might be

```
CALL (6, A, B, 1, N*3, SIN(X + Y))
CALL (17, A[1], 5, N)
```

WARNING

Location 111_8 of the standard BASIC system contains the address of the last word of available memory. It is not possible to create a system which requires more space than that existing between the addresses in locations 110_8 and 111_8. Systems using all or most of this space leave very little space for the user of the system.

INTERNAL DESIGN CONSIDERATIONS FOR CALL SUBROUTINES

The parameters of a CALL statement provide the dynamic link between BASIC and the called subroutine. Prior to transferring control to the subroutine, BASIC evaluates the parameters and stacks the addresses of the results. Upon entering the subroutine, the A-register contains the address of this stack (i. e. , the address of the addresses of the values of the parameters). The A-register initially points to the address of the first parameter; successively decrementing the A-register causes it to point to successive parameter addresses. A typical situation follows:

17300	3rd parameter address	Parameter address list	17302	A-register upon entry to called subroutine.
17301	2nd parameter address			
17302	1st parameter address			

The parameter addresses passed by BASIC give the subroutine access to values in the BASIC program. The only way a called subroutine can transmit its results to a BASIC program is to store through the address of a parameter.

Transmittal of quantities of data between a BASIC program and a called subroutine is most conveniently handled through arrays. Since only addresses are passed to a subroutine, an array parameter must be an element of the array (in general this would be the first element of the array). When using arrays in this manner, it is important to remember that arrays are stored by rows, and that each element is a floating point number occupying two words. Hence, if an array A has M columns per row, the address of A[I , J] is (address A[1, 1] + 2(M(I-1) + (J-1)).

It may be necessary or desirable for a called subroutine to output on the teletype. This may be done by loading a buffer address into the B-register, a character count into the A-register, and executing a JSB 102B, I. The referenced block of core will be interpreted as an ASCII string and output accordingly, followed by a carriage return and line feed.

SYSTEM PROTECTION

Whenever data is transferred from a called subroutine through the address of a parameter, there is a danger that the BASIC system or users program might be destroyed. This situation can arise when parameters are specified incorrectly or insufficient space is allocated in a data array. For example, constants (e. g. 2 or -1. 1) in a BASIC program are stored in the program as they appear, therefore storing through the address of a constant parameter will change the actual constant in the CALL statement. A subsequent execution of that statement may lead to strange results. A parameter that is an expression (e. g. A & B or NOT A AND B) will be evaluated and the result placed in a temporary location. Since the parameter address references this temporary, storing into it will not harm the BASIC system or program. However, the value stored there is lost to the BASIC program. If a called subroutine stores more values in an array than the array can hold, the resulting overflow of data may destroy the BASIC system or program. To provide some protection against these dangers, users of CALL statements should be cautioned against the danger of using unsuitable parameters in CALL statements (especially against using a simple variable or a constant where an array element is expected). Also, when using arrays as parameters it is good practice to include the dimensions of the array as additional parameters to allow a means of checking within the subroutine.

A really effective measure of protection is available at the cost of additional programming effort. BASIC contains sets of pointers delimiting the areas of memory within which different types of parameters exist. By checking parameter addresses against these bounds, the subroutine can verify that they are of the expected type. If X represents the parameter address, the following applies:

a) Constant parameter (112B) < X < (113B)

b) Simple variable parameter (116B) < X < (117B)

c) Array parameter 1) In common storage (110B) < X < (112B)
2) Not in common storage (113B) < X < (115B)

d) Expression parameter (115B) < X < (1120B)

where (112B) etc. means the contents of location number octal 112.

LOGICAL OPERATORS IN THE BASIC LANGUAGE C

HP BASIC extends the concept of the formula to include logical operators. Included are the familiar relational operators =, # (not equal), <, >, <= (less than or equal), and >= (greater than or equal) and the Boolean operators AND, OR, and NOT. With the exception of NOT, which takes the following operand as its single argument, all of these are binary operators. In a formula the binary operators have a lower priority than any of the arithmetic operators ↑, *, /, +, and -. Among the binary operators the priority in ascending order is OR, AND, and the relational operators (all of equal priority). NOT has the same priority as the unary + and unary -. Thus

NOT A + B = C OR SGN(K) AND SGN(J)

is equivalent to

(((NOT A) + B) = C) OR (SGN(K) AND SGN(J)).

The relational operators take algebraic numbers as arguments and return 0 (false) or 1 (true) according to the relationship existing between their arguments. Thus 3< 2 evaluates to 0 and A # 0 evaluates to 1 if A has a non-zero value. The Boolean operators consider their arguments as zero (false) or non-zero (true) and return 0 or 1 as follows:

AND			OR			NOT	
Arg 1	Arg 2	Result	Arg 1	Arg 2	Result	Arg	Result
non-zero	non-zero	1	non-zero	non-zero	1	non-zero	0
non-zero	zero	0	non-zero	zero	1	zero	1
zero	non-zero	0	zero	non-zero	1		
zero	zero	0	zero	zero	0		

Thus 3 and 1 evaluates to 1 while NOT -3 AND 0 evaluates to 0. It is important to realize that A < B < C is not equivalent to A < B AND B < C. If A = -3, B = -2, and C = 1/2 the former evaluates as (-3 < -2) < 1/2 or 0 (false) whereas the latter evaluates as (-3 < -2) AND (-2 < 1/2) or 1 (true).

BASIC SYNTAX IN BACKUS NORMAL FORM D

The syntax of HP BASIC is described in Backus Normal Form through the use of several metalinguistic symbols. Quantities within < and > are metalinguistic variables representing a syntactic class. The symbol "::" means "is defined as" and connects the metalinguistic variable on the left with its definition on the right. Multiple definitions of a metalinguistic variable are separated by the symbol "|", meaning "or". All capital letters and symbols not enclosed in < and > are actual characters as they appear in the language (e.g., RESTORE or <integer>.<integer>). Uncapitalized letters represent English language expositions such as carriage return. Juxtaposition of quantities in a definition implies juxtaposition in the language; all BASIC punctuation appears explicitly.

Example: <integer>::=<digit>| <integer><digit> explains that an <integer> is either a <digit> or an <integer> immediately followed by a <digit>. (Remember that blanks in BASIC only count within a character string, so that 1 23 is the same as 123.) It is easy to see that an integer begins with a digit and absorbs digits to its right one by one until it finds a non-digit character.

```
<basic program>::= <program statement>| <basic program><program statement>(1)
<program statement>::= <sequence number><basic statement> carriage return
<sequence number>::= <integer>(2)
<basic statement>::= <let statement>| <dim statement>|<com statement>|
            <def statement>|<rem statement>|<go to statement>|
            <if statement>|<for statement>|<next statement>|<gosub statement>|
            <return statement>|<end statement>|<stop statement>|
            <wait statement>|<call statement>|<data statement>|
            <read statement>|<restore statement>|<input statement>|
            <print statement>|<mat statement>
<let statement>::= <let head> <formula>
<let head>::=  LET <variable> = |<let head><variable> =
<formula>::= <conjunction>|<formula> OR <conjunction>
<conjunction>::= <boolean primary>|<conjunction> AND <boolean primary>
<boolean primary>::= <arithmetic expression>|
            <boolean primary> <relational operator> <arithmetic expression>
```

```
<arithmetic expression>::= <term>|<arithmetic expression> + <term>|
            <arithmetic expression> - <term>
<term>::= <factor>|<term> * <factor>|<term>/<factor>
<factor>::= <primary>|<sign><primary>|NOT <primary>
<primary>::= <operand>|<primary>    <operands>
<relational operator>::= >|<|>=|<=|=|#
<operand>::= < variable>|<unsigned number>|<system function>|
            <function>|<formula operand>
<variable>::= <simple variable>|<subscripted variable>
<simple variable>::= <letter>|<letter><digit>
<subscripted variable>::= <array identifier><subscript head>
            <subscript><right bracket>
<array identifier>::= <letter>
<subscript head>::= <left bracket>|<left bracket><subscript>,
<subscript>::= <formula>
<letter>::= A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z
<digit>::= 0|1|2|3|4|5|6|7|8|9
<left bracket>::= (|[
<right bracket>::= )|]
<sign>::= +|-
<unsigned number>::= <decimal part>|<decimal part><exponent>
<decimal part>::= <integer>|<integer>.|<integer>.<integer>|. <integer>
<integer>::= <digit>|<integer><digit>
<exponent>::= E<integer>|E<sign><integer>
<system function>::= <system function name><parameter part>
<system function name>::= SIN|COS|TAN|ATN|EXP|LOG|ABS|SQR|INT|RND|SGN
<parameter part>::= <left bracket><actual parameter><right bracket>
<actual parameter>::= <formula>
<function>::= FN<letter><parameter part>
<formula operand>::= <left bracket><formula><right bracket>
<dim statement>::= DIM <formal array list>
<formal array list>::= <formal array>|<formal array list>, <formal array>
<formal array>::= <array identifier><formal bound head><formal bound>
            <right bracket>
<formal bound head>::= <left bracket>|<left bracket><formal bound>,
<formal bound>::= <integer>(3)
<com statement>::= COM<formal array list>
<def statement>::= DEF FN<letter><left bracket><formal parameter>
           <right bracket>=<formula>
<formal parameter>::= <simple variable>
<rem statement>::= REM<character string>
<character string>::= <character>|<character string><character>
<character>::= any teletype character except carriage return, alt mode, escape,
           rubout, or line feed
<go to statement>::= GO TO<sequence number>
<if statement>::= IF<formula> THEN <sequence number>
<for statement>::= <for head>|<for head>STEP <step size>
<for head>::= FOR<for variable>=<initial value>TO<limit value>
<for variable>::= <simple variable>
<initial value>::= <formula>
<limit value>::= <formula>
<step size>::= <formula>
<next statement>::= NEXT<for variable>
<gosub statement>::= GOSUB <sequence number>
<return statement>::= RETURN
<end statement>::= END
<stop statement>::= STOP
<wait statement>::= WAIT <parameter part>
<call statement>::= CALL<call head><right bracket>
<call head>::= <left bracket><driver number>|<call head>, <actual parameter>
<data statement>::= DATA <constant>|<data statement>, <constant>
<constant>::= <unsigned number>|<sign><unsigned number>
<read statement>::= READ<variable list>
<variable list>::= <variable>|<variable list>, <variable>
<restore statement>::= RESTORE
<input statement>::= INPUT<variable list>
<print statement>::= <print head>|<print head><print formula>
<print head>::= PRINT|<print head><print part>
<print part>::= <string>|<string><delimiter>|<print formula><delimiter>|
           <print formula><string>|<print formula><string><delimiter>
```

```
<string>::= "<character string>"(4)
<delimiter>::=  , | ;
<print formula>::= <formula> | TAB <parameter part>
<mat statement>::= MAT <mat body>
<mat body>::= <mat read> | <mat print> | <mat replacement>
<mat read>::= READ <actual array> | <mat read>, <actual array>
<actual array>::= <array identifier> | <array identifier><bound part>
<bound part>::= <actual bound head> <actual bound> <right bracket>
<actual bound head>::= <left bracket> | <left bracket> <actual bound>,
<actual bound>::= <formula>
<mat print>::= PRINT <mat print part> | PRINT <mat print part> <delimiter>
<mat print part>::= <array identifier> | <mat print part> <delimiter> <array identifier
<mat replacement>: := <array identifier>= <mat formula>
<mat formula>::= <array identifier> | <mat function> | <array identifier> <mat operator
            <array identifier> | <formula operand>* <array identifier>
<mat function>::= <mat initialization> | <mat initialization> <bound part> |
            INV <array parameter> | TRN <array parameter>
<mat initialization>::= ZER | CON | IDN (5)
<array parameter>::= <left bracket> <array identifier> <right bracket>(6)
<mat operator>::=  + | - | * (7)
```

(1) The <com statement>, if any exists, must be the first statement presented and have the lowest sequence number.
(2) A sequence number may not exceed 9999 and must be non-zero.
(3) A formal bound may not exceed 255 and must be non-zero.
(4) Strings may not contain the quote character (").
(5) A <bound part> for an IDN must be doubly subscripted.
(6) An array may not be inverted or transposed into itself.
(7) An array may not be replaced by itself multiplied by another array.

PREPARE BASIC SYSTEM PROGRAM (PBS)

The Prepare BASIC System Program prepares a composite binary tape containing configured system input/output drivers, the significant locations in the system linkage area and, optionally, the BASIC compiler.

PBS is an absolute program containing unconfigured teletype, photo reader and punch drivers. The BASIC compiler will be included on the composite tape if it is loaded along with PBS by the Binary Loader.

When initiated, PBS requests I/O assignments for the teletype, photo reader and punch, configures the drivers in accordance with these assignments, and punches a binary tape containing the configured drivers and the BASIC compiler, if loaded. Multiple copies of this tape may be obtained.

OPERATING PROCEDURES FOR PBS

The system on which the composite tape is prepared must have a teletype ASR 33 or 35 in the same I/O channels as that of the system on which the compiler is to run. The tape is prepared in the following manner:

1. Load PBS using the Binary Loader.
2. Load the BASIC compiler, if desired.
3. Set the switch register to 000002. Press Load Address.
4. Set switches 5-0 of the switch register to the lower numbered channel assigned to the teletype.
5. Press Run.
6. PBS outputs the following request on the teletype:

   PHOTO READER I/O ADDRESS?

   Respond by inputting via the teletype keyboard the I/O channel number assigned to the photoreader followed by a carriage return (CR) . If there is no photoreader, type a (CR) only.

7. PBS outputs the following request on the teletype:

   PUNCH I/O ADDRESS?

   Respond by inputting via the teletype keyboard the I/O channel number assigned to the punch followed by a (CR) . If there is no punch, type a (CR) only.

8. Preparatory to punching the composite tape, PBS inquires if there is a high speed punch on the system, which is being used to prepare the tape, by outputting the request:

   SYSTEM DUMP I/O ADDRESS?

   a) If there is a high speed punch on the system, input its I/O channel number on the keyboard following by a (CR) . PBS then proceeds to punch the composite tape on the high speed punch.

      [Note if the preparation is performed on the computer on which BASIC is to run, the punch address given in 8 will be the same as that given in 7.]

   b) If there is no high speed punch on the system, input a (CR) only. PBS prints:

      TURN ON TTY PUNCH, PRESS RUN

      and halts the computer with 102011 showing on the T-register.

   Turn on the TTY punch as instructed. When Run is pressed, PBS will proceed to punch the composite tape on the TTY punch.

9. When punching is complete, PBS will halt the computer with 102077 showing on the T-register. Further copies of the tape may be obtained by pressing Run after such a halt.

   The following error conditions are detected by PBS:

   1. If, in step 4, switches 5-0 of the switch register are set to a value less than 10_8, PSB will halt the computer with 102055 showing on the T-register. To recover, set the switch register to the correct number and press Run.

   2. If in steps 6, 7, 8 a number less than 10_8 or greater than 77_8 is input PSB will output the message:

      INVALID I/O ADDRESS

      To recover, input the correct number via the keyboard.

   3. A channel number may consist only of two digits, no preceding or trailing blanks or other characters.

$3

5950-8313

HEWLETT hp PACKARD

A Pocket Guide to Hewlett-Packard Computers

HEWLETT-PACKARD COMPANY 11000 WOLFE ROAD, CUPERTINO, CALIFORNIA 95014

PRINTED IN U.S.A. 7/70