Honeywell

BASIC LANGUAGE

SERIES 16

SUBJECT:

Description and Use of BASIC Language for Models 316 and 516, Including Stand-Alone and OP-16 Versions of BASIC Interpreter.

SPECIAL INSTRUCTIONS:

This manual, Order Number AB85, Rev. 1, supersedes the previous edition, Order Number AB85, Rev. 0 (Formerly M-449), Document Number 70130072543A, dated May 1971. Technical changes have been made in accordance with ECO-20051. Bars in the margins indicate revisions; asterisks indicate deletions.

DATE:

March 1972

ORDER NUMBER:

AB85, Rev. 1

DOCUMENT NUMBER:

70130072543B

PREFACE

This manual describes the structure and use of BASIC (Beginner's All-Purpose Symbolic Instruction Code), an engineering- and science-oriented programming language for Series 16 computers having as little as 4,096 words of memory. The manual has been designed for the computer novice. No actual experience in computer programming is assumed. If the new BASIC user has mastered elementary algebra, he can begin programming simple problems immediately.

After an introduction to the BASIC language, including a summary of its major constituents and the system commands available to the interactive user, the manual deals in succeeding sections with the syntax of the language, its five statement groups, and the programmatic interface between programs written in BASIC and programs written in the DAP-16 assembly language or in Fortran. A concluding section on general operational procedures associated with the BASIC Interpreter shows that it can be used as a stand-alone interactive program or as a subsystem of the OP-16 Operating System.

Applicable documents include OP-16 Operating System, Order Number BY06; DAP-16 and DAP-16 Mod 2 Assembly Language, Order Number BY09; and Series 16 Fortran IV, Order Number BX32.

©1971, 1972, Honeywell Information Systems Inc.

File No. 1A23

AB85

CONTENTS

ILLUSTRATIONS

TABLES

SECTION I
INTRODUCTION

LANGUAGE SUMMARY

BASIC is a programming system that was developed at Dartmouth College in the middle of the sixties. It has been implemented on numerous computers since then because it is an easy language to learn and has a simple algebraic notation familiar to a very large class of prospective users.

A BASIC program consists of a set of statements. Each statement in a stored program must be numbered. Unnumbered statements are executed immediately. The program is executed in the numerical sequence given, though the statements need not appear in any given sequence.

Table 1-1 lists the major constituents of the BASIC language. Note that the use of these statements is reasonably self-evident. The angular and square brackets in the table represent variables and optional elements, respectively.

LET and GO TO are the assignment and control transfer statements, respectively. GO SUB is a subroutine call. Functions are invoked by enclosing the parameter in parentheses following the function name. IF ... THEN allows a single relational operator between expressions, and control passes to the statement number following THEN on the true path. NEXT is used to terminate the range of the FOR statement and must use exactly the same variable as in the FOR statement. (If the third expression or STEP in a FOR statement is omitted, it is assumed to be one.)

The expression in the ON statement is evaluated and truncated to an integer. For expression =1, control is transferred to the first statement number in the list; for expression =2, control is transferred to the second statement number in the list; etc.

READ assigns to the listed variables the values obtained from a DATA statement. The latter is used to specify all the values needed for the variables. For output, the user can specify variable names or literals; the literals are enclosed within the quotation marks. Thus, the statement PRINT "THE SQUARE ROOT OF" X, "IS" SQR(X) might cause the following to be printed: THE SQUARE ROOT OF 625 IS 25. For normal printing purposes, the output line is divided into five zones of 13 squares each. The user can change the

width of these zones, however, through the use of commas, semicolons, and TABS. Commas are used to advance to the beginning of a new zone. A PRINT command without anything following it signals a new line.

Table 1-1. BASIC Language Summary

LET <variable> = <expression> or <variable> = <expression>

GO TO <statement number>

GO SUB <statement number>

RETURN

IF <expression><relation><expression> THEN <statement number>

FOR <unsubscripted variable> = <expression> TO <expression> STEP <expression>

NEXT <unsubscripted variable>

ON <expression> GO TO <statement number> [<statement number>] ...

READ <variable> , <variable> , ..., <variable>

PRINT <literal or expression> , <literal or expression> ,...

STOP

END

CALL (<subroutine number> , <argument list>)

DIM <variable> (<integer> [, <integer>])

DATA <number> , <number> , ...

RESTORE

INPUT

DEF FN <letter> (<unsubscripted variable>) = <expression>

REM <any string of characters>

The execution of an END statement terminates the program: the STOP acts like a GO TO where the statement number represents the END command.

The CALL statement is used to invoke one of up to ten subroutines written in FORTRAN IV and the DAP-16 assembly language. The entry points of the subroutines are inserted in a table maintained by the Interpreter. The subroutine number tells BASIC which table entry to use. Arguments can then be transferred to and from the CALLing BASIC program.

The DIMension statement is used to specify arrays which contain more than 10 subscripts. DATA specifies the input constants. RESTORE returns the DATA pointer to the first constant of the first DATA statement. An INPUT statement causes an exclamation point to be typed and the program waits for the user to type in the number of data items requested in the read list.

Functions are defined by the DEF statement; the function name consists of the letters FN followed by a letter. Any expression which fits on one line can be used to define a function, including another function. In addition to eight intrinsic functions (sine, cosine, exponentiation, etc.), the INT and RND functions are provided. The former is |X|, i.e., the greatest integer not greater than X; the latter produces a random number between 0 and 1.

REMark statements are nonexecutable and are used to enter comments and explanations in the program listing.

SYSTEM COMMAND SUMMARY

The interactive user is also provided with system commands for complete control of processing from the teletypewriter console. The use of these system command is explained in Table 1-2.

Table 1-2. System Commands

COMMAND	RESULT
JOB	Clears the program and data value array in memory.
CLEAR	Clears the data value array in memory.
PUNCH	Punches a paper tape.
LOAD	Loads a paper tape.
RUN	Begins to execute the program at the statement with the lowest line number. A RUN command CLEARS the data value array in memory.
RUN sn	Begins to execute the program at the statement number (sn) specified.
LIST	Lists the program on the teletypewriter.
LIST sn	Lists the program starting at the statement number (sn) specified.
LIST sn, sn	List the statements between the given statement numbers.
CONTINUE	Setting Sense Switch 1 (SS1) on the computer console causes the system to type the line number being executed and the word BREAK, and to return to the system command mode. The CONTINUE command resumes execution where the program stopped after SS1 was set. CONTINUE cannot be used if the program was modified following BREAK.
QUIT	Terminates the BASIC Interpreter in the OP-16 version of BASIC, and places the computer in the HALT state in the stand-alone version.

With the system in the command mode, the user is able to perform extensive line editing of his source program. For example, statement lines may be deleted by typing the number and omitting the statement. Lines may be added by typing them with line numbers which insert the statements between existing lines and, of course, lines may be replaced by typing the previous line number followed by a new statement line.

The program is executed sequentially by line number unless a control statement causes a transfer. A program can be saved for later use by "PUNCHing" it.

REQUIRED HARDWARE

BASIC requires a minimum of the following equipment for operation:

a. Model 316 or 516 Central Processor.

b. 4K of core memory (BASIC can expand its buffers to use up to a maximum of 16K).

c. AST-33 or -35 teletypewriter, Types 5303 and 5503 (for Model 316) and Types 5305 and 5505 (for Model 516).

No other processor, peripheral, or communications subsystem facilities are supported by the standard 4K BASIC Interpreter. However, input/output is handled in the software by an IOS (Input/Output Supervisor) package which will allow expansion to devices other than the teletypewriter in other versions of the Interpreter. An IOS is available which loads by using a High-Speed Paper-Tape Reader and punches by using a High-Speed Paper-Tape Punch.

REQUIRED SOFTWARE

BASIC is a stand-alone program requiring no other programs. The self-loading version contains its own loader, IOS package, and system command facilities. The Interpreter will accept and execute any program written in standard BASIC. Provision is made in the system for linking up to 10 subroutines written in FORTRAN or DAP-16 to the main BASIC program in core, contingent on the availability of core storage.

PERFORMANCE SPECIFICATIONS

The BASIC Interpreter provides an interactive environment in which a user can compose, edit, debug, and execute programs written in BASIC. The user may enter an entire program from the teletypewriter, call for its execution, or enter an individual language statement(s) for immediate execution. Statements are checked for syntactic and logical errors during execution. All errors are reported to the user when detected. All errors return control to the command mode.

Honeywell Series 16 BASIC represents an extension of the programming facilities available under the Dartmouth College system. Chief among these extensions are the ability to include multiple statements in a line, immediate execution of statements and commands, the availability of n-dimensional arrays with unrestricted subscript expressions, and the ability to call DAP-16 and FORTRAN subroutines for the BASIC program.

The stand-alone version is loaded from self-loading paper tape at the teletypewriter and executed starting at location 1000_8. All further communication is made via the console.

The OP-16 version of the Interpreter can be configured to execute under the RTX-16 Executive. BASIC is called from the console by using the RTX keyboard program. All further communication is directly with the BASIC Interpreter by use of the ASR console.

SECTION II

LANGUAGE

INTRODUCTION

A BASIC program is a self-contained computing procedure (specified by program statements) telling the computer what steps to perform to solve a problem and how to present answers for the data given. For example, consider a quadratic equation of the form

$$X^2 + 2X - 4 = 0$$

The algebraic representation for one of the two roots of the equation can be written

$$ROOT = \frac{-B + \sqrt{B^2 - 4AC}}{2A}$$

The four short but complete BASIC programs shown in Figure 2-1 (a-d) were written to solve the expression for A=1, B=2, and C=-4. Each was entered and run at the console exactly as shown, with the computer supplying the responses that are underscored.

a. Use of READ and DATA Statements for Variables

```
?JOB
?10 READ A, B, C
?20 LET X=(-B+(B↑2-4*A*C)↑0.5)/(2*A)
?30 DATA 1,2,-4
?40 PRINT "ROOT IS",X
?50 END
?RUN
ROOT IS             1.23607

50 EXIT
?
```

b. Use of Data Constants

```
?JOB
?10 LET X=(-2+(2↑2-4*1*(-4))↑.5)/(2*1)
?20 PRINT "ROOTIS",X@
?20 PRINT "ROOT IS",X
?30 END
?RUN
ROOT IS             1.23607

30 EXIT
?
```

Figure 2-1. BASIC Programming Example

c. Use of INPUT Statement

```
?JOB
?10 INPUT A,B,C
?20 LET X=(-B+(B↑2-4*A*C)↑.5)/(2*A)
?40 PRINT "ROOT IS", X
?50 END
?RUN
!1,2,-4
ROOT IS            1.23607

50 EXIT
?
```

d. Use of Arithmetic Calculator Loop

```
?JOB
?10 INPUT I:  PRINT I:  GO TO 10
?RUN
!(-2+(2↑2-4*1*(-4))↑.5)/(2*1)
    1.23607
!
```

Figure 2-1 (cont). BASIC Programming Example

With BASIC loaded and running in the computer, the Interpreter requests user input by typing a "?". The programmer may respond by typing in either a system command or a BASIC statement. If the statement is to be stored as part of a program, it must have a statement number. If there is no statement number, the statement is checked for errors and executed immediately. The Interpreter allows a user to assemble an entire program and run it or to transfer control to a segment of a program and start execution there.

In Figure 2-1a, JOB is entered to indicate that a new task is to be performed. Now the statements of the program can be entered. Statement 10, the READ statement, is used in conjunction with statement 30, the DATA statement. When the BASIC Interpreter scans a READ statement it causes the variables A, B, and C listed after READ to be given values according to the next available numbers in the DATA statement. In the example, A is assigned a value of 1, B a value of 2, and C a value of -4.

The LET statement in line 20 directs the computer to evaluate the expression or formula on the right side of the equals sign and set it equal to the variable X. Note that exponentiation and multiplication operations are represented in BASIC by the "↑" and "*" signs -- e.g., B↑2 for B^2, X↑0.5 for $X^{1/2}$ or $\sqrt{X}$, and 4*A*C for 4xAxC or 4AC. Line 40 tells the computer to PRINT the message "ROOT IS", to recover the value of the variable X (which was evaluated in line 20), and to PRINT that after the message.

Line 50 tells the computer that the entry of the program or JOB is complete.

Typing RUN requests the computer to execute the program just entered. If the program contains no syntactic or logical errors, the computer types out the message "ROOT IS" and the value of the variable X, the number of the last statement executed, and "EXIT".

Figure 2-1a illustrates the use of variables in an expression (in this case A, B, and C), but the same result can be obtained as shown in Figure 2-1b by inserting numerical values in the statement in place of the variables. The use of variables allows one to change his data very conveniently by merely entering a new DATA statement.

Often it is desirable to enter data during program execution. For instance, one person may write a program for which other people wish to supply data relevant to their own application. The INPUT statement of Figure 2-1c and d act like a READ statement but does not draw numbers from a DATA pool as was done in Figure 2-1a.

To supply values for the A, B, and C variables to the sample program, we insert an INPUT statement before the first statement which uses any of these numbers. When the computer encounters this statement, it types an exclamation point and awaits user entry of data. The user then types in the three numbers (separated by commas) and presses the carriage return key; the computer finishes executing his program.

One of the interesting aspects of using a dedicated (one-user) computer is that a program such as we see in Figure 2-1d is possible. Here a very-high-speed calculator capable of evaluating arithmetic expressions up to 72 characters long is implemented by a simple, one-line multiple-statement program loop. We also see a very significant feature of Series 16 BASIC, the multiple-statement line delimiter (:).

BASIC editing features include the ability to delete lines or characters. For example, when inputting the statement of line 30 in Figure 2-1a, a backward arrow (←) is used to remove an improper character, the "-" instead of a ",". In line 20 of Figure 2-1b, the "at...each" sign (@) is used to delete the entire typed line. To replace a line, type another line with the same line number.

SYNTAX

Statement Line

Every statement line in a BASIC program consists of (1) a line number and (2) one allowable BASIC instruction, in that order. There are 72 printing positions across the terminal line for these two items. Instructions may not be continued from one line to the next. Except where noted, blanks may be used freely. More than one statement may be included in the line through the use of the colon, as explained later.

Line Number

Every line in a BASIC file must have a unique line number in the range of 1 to 9999, inclusive; 0 is not a valid line number. A line number may have leading 0s (as 005), but it may not contain any characters other than the digits 0 through 9 (no blanks and no commas).

Legal Numbers	Illegal Numbers
007	0.07
3245	3,245.
1	1X1
473	123456

Line numbers serve two purposes. First, they allow the system to order the program automatically according to line number. In this way, lines can be entered in a program in any order, but they are always processed in ascending numerical order. Line numbers do not need to start with 1 or be consecutive, but they must be unique. It is a common programming practice to number lines by using multiples of 10. Then if it is necessary to revise or correct the program by inserting a new line, it is given an intermediate number (11, 22, 34, etc.).

The second function of line numbers is to act as labels. Normally the lines in a program are executed one after another in order of line number. However, this order can be changed by certain instructions which transfer program control to a line number which is out of sequence. (Program control refers to the selection of the instruction to be executed next.) Control can be transferred forward (skipping instructions) or backward (repeating instructions). Line numbers are used as reference points when directing the computer to break the normal, serial execution of instructions.

Statement Operator

Immediately following the line number is the statement operator. The rest of the 72 characters allowed for one line contain a BASIC instruction. There are several types of statement operators. Each instruction in a program performs a specific duty and is in one of the general instruction forms. The statement operator is a string of characters which tells the BASIC compiler which statement form is being used. In line 10 of the sample program in Figure 2-2, the statement operator is the word LET. The other two statement operators used in the program are the words PRINT and END. The word LET informed the computer that the statement that followed would be an arithmetic calculation to compute a value and assign it to a variable. In the example, there was no computation necessary; the LET statement merely assigned the value 5 to the variable X.

Spaces may be used freely because the compiler ignores them except in something called an alphanumeric literal, which will be discussed later. The user may write text as shown in Figure 2-2. This is not as tidy as possible but good enough.

```
?10     LETX = 5
?20PRINTX↑2+2.2*X+4
?30 E N  D
?RUN
   40

30 EXIT
```

Figure 2-2. Statement Format Example

Use of Statement Delimiter (:)

BASIC statements are normally assigned one to the line. However, as shown in Figure 2-1c, it is possible to put as many statements on a line as there are print positions available. For example, line 104 in Figure 2-3 constitutes an entire program of five statements and illustrates the legal use of the colon as a statement delimiter.

```
?JOB
?104 FOR X=1TO5:PRINT X;SQR(X):NEXT X:END
?RUN
     1       1
     2       1.41421
     3       1.73205
     4       2
     5       2.23607

 104 EXIT
 ?
```

Figure 2-3. Use of Statement Delimiter

Notice that if more than one statement is to be included on a line, statement numbers are omitted after the initial statement number for that line. The major advantages accruing to the use of the delimiter are that it saves a little typing time on program entry and allows the Interpreter to use memory locations normally assigned to statement numbers (in their function as logical "labels") for programs and data.

*

REMarks

Often a programmer may wish to include comments in his program. Many times a few words at the beginning of a program, which explain what the program does, save time and trouble later. Remarks (REM statement) are also helpful to anyone other than the programmer who uses and revises the program. Remarks are treated like blanks; that is, they are ignored by BASIC, except when LISTing the program.

A remark has the form

```
ln REM STRING
```

where	ln	is a line number
	REM	is the statement operator
	STRING	is any string of legal characters

The REM statement, like all other statements, has a line number and a statement operator. Any characters which can be typed at the terminal (except ← and @) may follow the statement operator. In Figure 2-4, two remarks have been added at the beginning of the program file to explain what the program does.

```
?LIST

5  REM  THIS IS A PROGRAM TO EVALUATE
10  REM  THE FORMULA X↑2 +1.2X+4
15  LET X=5
20  PRINT X↑2+1.2*X+4
25  END

?RUN
   35

25 EXIT
?
```

Figure 2-4. Use of REMark Statements

The REM statement is an example of a nonexecutable specification statement. When it is encountered during execution, it is ignored; that is, it causes no action to be taken.

ELEMENTS OF BASIC

An intuitive understanding was all that was required for the constants and variables of the sample program. BASIC, however, does have rules for forming constants, variables, and expressions. These rules are discussed in the following paragraphs.

Constants

DESCRIPTION

A constant is simply a decimal number with an optional minus sign. The sample program in Figure 2-1 used the constants 1, 2, and -4. The absolute value of a constant must be greater than 10^{-38} and less than 10^{+38}. The constant may have a minus (-) sign preceding it. If the minus sign is not present, the number is assumed to be positive. The constant may also have a decimal point.

There are three external forms a constant may take in a program or in data for the program: integer, floating-point, or exponential.

All constants are stored internally as either one-word, fixed-point numbers or two-word, floating-point numbers. In the internal exponential notation, every number is expressed as a value between 0.1 and 1.0 times a power of 10. For instance, the constant 5.1 can be expressed as $0.51 \times 10^{+1}$. The computer stores only the mantissa (.51) and the exponent (+1). The value 5 would be stored as a one-word, fixed-point number no matter how it was entered. This is a very efficient way of storing the most significant part of a number. An integer between +32,767 and -32,767 is stored as one fixed-point word no matter how it is entered.

A constant will have an accuracy of approximately six digits. For instance, the number 123,456,789 would be stored internally as $.123456 \times 10^{+9}$. The last three digits would be lost.

INTEGER NUMBERS

An integer is a whole number which may have a sign but cannot have a decimal point.

FLOATING-POINT NUMBERS

A floating-point number has a decimal point. It may or may not have a whole number part, and it may or may not have a fractional part. Besides numeric digits, the only characters allowed in a floating number are a plus or minus sign and a decimal point.

EXPONENTIAL NUMBERS

The exponential format for numbers is much like the internal form the machine uses. This form is a floating-point or integer number mantissa with a power of 10 added. A user is not restricted to using a mantissa between 0.1 and 1.0, as is the machine. The numbers 5, 50×10^{-1}, $0.5 \times 10^{+1}$, and $5,000,000 \times 10^{-6}$ are all equivalent.

The letter E is used to denote the exponent instead of the number 10 with a superscript. The E separates the mantissa from the exponent. The numbers 5, 50E-1, .5E+1, and 5000000E-6 are all equivalent. The exponent may have been signed (+ or -). The sign can be omitted, in which case any missing sign is assumed to be positive. A one- or two-digit integer number may be used as an exponent as long as the resulting constant is neither greater than +38 nor less than -38.

A constant does not change its value during a run or from one run to the next. A 1 is always 1. Values which do change are given names and called variables.

Variables

A variable is a name which represents a value. A variable name may be one letter (A through Z) or one letter followed by one digit (0 through 9). There are thus 26 plus 26x10 = 286 variable names a BASIC programmer may use. The sample program used the variable X. A variable may have its value changed during the run or from one run to the next.

Legal Names	Illegal Names
A	1A (starts with a digit)
X	XY (second character not a digit)
F2	Q37 (too long)
Z0	ZO
N	

Simple variables are not initialized in BASIC. (Arrays are always initialized to zero.) Thus the appearance of a variable in an expression before it has been assigned a

value through a READ, INPUT, or LET statement will result in the following error message being printed at the console:

ERROR UV LINE XXXX

A variable may be equal to any value a constant may have. All variables are stored as standard floating-point numbers (two words per variable). Thus a variable must have a magnitude greater than 10^{-38} and less than 10^{+38} and will retain six digits of accuracy.

Expressions

An expression defines an arithmetic calculation. Expressions are composed of variables, constants, arithmetic and relational operators, intrinsic and programmer-defined functions, and parentheses. An expression may consist of as little as one constant and one variable.

ARITHMETIC OPERATORS

The arithmetic operators are:

↑	exponentiation
*	multiplication
/	division
+	addition
-	subtraction

Arithmetic operators should not be confused with statement operators (LET, PRINT, etc.). Statement operators apply to whole statements, while arithmetic operators apply only to constants and variables.

RELATIONAL OPERATORS

Six relational operators are used to describe the numeric relationship between two numbers:

Operator	Meaning
=	Equal to
< = or = <	Less than or equal to
<	Less than
>	Greater than
>= or = >	Greater than or equal to
<> or ><	Not equal to

Relational operators are used to compare two numerical quantities which may be expressions (containing arithmetic operators), variables, and/or constants. A relational expression has the form

	e_1 op e_2	
where	e_1, e_2	are expressions to be evaluated
	op	is one of the six relational operators

The relational expression is either true or false. Either the two expressions satisfy the relation stated, or they do not.

The double relational operators (<=, >=, < >) are true if either relation is true. In the expression A1> =A2, the relation is true either if A1>A2 or if A1=A2. Below are some further examples of relational expressions:

```
X > Y
A5<> 6
X*2+Y↑2+3*X*Y<= 5
```

The conditional transfer statements (see Section III, Control Statements) base the decision of where program control will go next on the result of a relational expression.

FORMING EXPRESSIONS

An expression closely resembles an algebraic formula. The formula for the sample program in Figure 2-1 is an expression. The right side of the equal sign in line 20 of the same example is also an expression. There are certain rules which BASIC expressions must follow but which algebraic formulas do not:

1. Two operators may not be adjacent. For instance, BASIC will not accept X↑ -2, but X↑ (-2) is perfectly legal.

2. Variables and constants may not be adjacent. For example, 1.2*X must be used for 1.2X. In algebra the lack of an operator is assumed to mean multiply, but BASIC makes no such assumptions.

3. Parentheses may enclose any legal expression. Thus, A + (B+) C is not accepted, since B+ is not a legal expression. A parenthesized expression is treated like a variable or constant; that is, it cannot appear adjacent to another variable, constant, or parenthesized expression without an operator in between. A(-B) is illegal, but A+(-B) is legal.

   As in algebra, parentheses are used to determine which operations are performed first. In the expression (A+B)*C, the addition is performed, then the multiplication. If the parentheses are changed so that the expression is A+(B*C), the multiplication is performed before the addition.

Parentheses may be "nested" or placed one pair inside another, as A*(B+C* (D+E)). Innermost parenthesized expressions are evaluated first. Any expression inside a pair of parentheses is evaluated before the expression containing the parentheses pair is evaluated.

4. A minus sign may precede any expression. The expressions A/(-B), (-A/B), and -(A/B) are equivalent. Care should be taken to guard against a minus sign appearing adjacent to another operator. A+-B is illegal, but A+(-B) and A-B are legal. Note also that while -3 is a legal constant, -3*-3 is not a legal expression.

EVALUATING EXPRESSIONS

In order to have consistent evaluation of expressions, a set of rules has been established. These rules form a hierarchy of arithmetic operations, that is, the order in which operations are to be performed. This is necessary because of the ambiguity which may arise in an unparenthesized expression.

For example, the formula

```
5  +  6.2 * 2↑3
```

has three operations. Any one of the operations could be performed first and the result used for the remaining operations. In total, there are six sequences in which the operations could be performed. Below, the six sequences are illustrated along with the resulting values. The answers range from less than 60 to over 11,000. The numbers above the expressions give the order of performance of the operations. Parentheses have been added to the expressions in each case.

```
   1      3   2
(5 + 6.2)* (2↑3) = 89.6

   1      2    3
((5 + 6.2)* 2) ↑3 = 11,239.424

   2       1     3
(5 + (6.2 * 2)) ↑3 = 5268.024

   3       1    2
5 + ((6.2 * 2) ↑3)= 1911.624

   2       3     1
(5 + 6.2) * (2  ↑3) = 89.6

   3       2     1
5 + (6.2 * (2 ↑3)) = 54.6
```

The four rules that govern the hierarchy of arithmetic operations are listed below. When these rules are applied to the expression above, the expression is evaluated as 54.6, the last possibility listed. The primary use of expressions is to define the value of a variable.

1. Hierarchy: This requires that certain operations be evaluated before others. In the absence of parentheses specifying the exact order of evaluation, the priority of operators from high (evaluated first) to low (evaluated last) is:

↑	exponentiation
* or /	multiplication and division (equal hierarchy)
+ or -	addition and subtraction

The same hierarchy applies to expressions within parentheses. This rule alone is enough to verify the evaluation of the above expression as:

```
5 + 6.2 * 2↑3 = 5 + (6.2 * (2↑3))
              = 5 + (6.2 * 8)
              = 5 + 49.6
              = 54.6
```

2. Equal Priority: When the priority of adjacent operators in an expression is the same, evaluation proceeds from left to right. For example, in the expression

```
A * B / C + D1 / D2 * D3
```

the * and / operators have the higher priority and so must be evaluated before the +. However, rule 1 does not state which is evaluated first, the * or /. Rule 2 says that operations are performed from left to right. So the expression is evaluated as:

```
((A * B) / C) + ((D1 / D2) * D3)
```

Notice that the addition is performed last, since rule 2 is applied only to adjacent operators of the same priority.

3. Parenthesized Expressions: Expressions within parentheses are evaluated separately, so parentheses may govern the hierarchy of operations. The expression inside a pair of parentheses is evaluated according to all the rules of expression evaluation. In the expression above, the addition was the last operation performed. Parentheses could alter this; for example:

```
A * B / (C + D1) / D2 * D3
```

4. Nesting: In a nest of parentheses (one pair of parentheses inside another pair), the expression within the innermost pair of parentheses is evaluated first. The expression within the next innermost pair is evaluated next, and so on down to the outermost pair, which is evaluated last. Note that -A1↑3=-(A1↑3).

SECTION III
LANGUAGE STATEMENTS

Series 16 BASIC provides for the following types of statement operators in defining the function of its language statements:

a. Arithmetic Assignment
b. Control
c. Output/Input
d. Specification
e. Subroutine/Function

The remainder of this section will define each statement type within these groups and illustrate their use in actual programs.

ARITHMETIC ASSIGNMENT STATEMENT

The arithmetic assignment statement, used to define a numerical calculation, closely resembles conventional arithmetic formulations. The LET statement commands the computer to perform the computations specified by an expression and to assign the value of that expression to a single variable. It has the form

	ln LET v = e	or	ln v = e
where	ln	is a line number	
	LET	is the statement operator (optional)	
	v	is any variable name	
	=	means "be replaced by"	
	e	is an arithmetic expression	

When the LET statement is executed, the expression e is evaluated and the result is substituted for the variable v. Variables and expressions must conform to the rules specified in Section II.

Notice that the equal sign (=) has the meaning "be replaced by" rather than "equals," because the LET statement is not an assertion of equality as is 5 + 2 = 7. Instead, a BASIC statement such as

```
5 LET D=D+1
```

causes the computer to add 1 to D and store the result in D. Each variable used in a BASIC program has a specified memory location assigned to it. When the value of a variable is changed, the new value replaces the old value, and the old value is destroyed. Hence the arithmetic assignment statement is not an equation but a command to replace a value.

An optional form of the LET statement permits several variables to be assigned a given value. It has the form

ln LET v_1, v_2, v_3, ... = *e*

where	ln	is a line number
	LET	is the statement operator
	v	is any variable name
	=	means "replaced by"
	e	is an arithmetic expression

Example: LET X, Y, Z = 10.

Result: X, Y, Z are all set to 10.

Line 20 of the sample program in Figure 3-1 contains an example of the BASIC arithmetic assignment statement. In this example, the expression to be evaluated determines the current drawn by an a-c circuit. The expression has the form

$$I = \frac{E}{\sqrt{R^2 + 2\pi FL - 1/(2\pi FL)^2}}$$

Line 20 assigns a value to the variable I (amperes) for fixed values of R (resistance), F (frequency), L (inductance), and C (capacitance).

```
10  READ E,R,F,L,C
20  LET I=E/(R↑2+(6.2832*F*L-1)/(6.2832*F*C)↑2)↑.5
30  DATA 120,200,60,.1,.1E-02
40  PRINT "THE CURRENT FOR THE STATIC CONDITION IS:  ",I," AMPERES."
50  END

?RUN
THE CURRENT FOR THE STATIC CONDITION IS:        .598073   AMPERES.

50 EXIT
?
```

Figure 3-1. Example of Arithmetic Assignment Statement

Other examples of legal arithmetic assignment statements are:

```
100 LET Z=(X1-Y↑3) * (Z-T/(T-S3))
110 LET Y = X↑4+3*X↑3-7./4*X↑2+13*X-8
120 S1 = S
130   A9 = A9 + A
```

CONTROL STATEMENTS

Normally statements are executed in sequence, starting at the lowest line number or statement number specified in a RUN statement. Control statements alter this sequence by changing the order of execution of a program. The function of control statements is to direct the program to continue execution at a different line number rather than at the next line number in sequence.

Control statements may be either conditional or unconditional, depending on whether a specific condition must be met before control is switched to another part of the program. When scanned, an unconditional control statement will immediately transfer control to the line number specified. Conditional control statements branch only if a specified condition is met.

Unconditional GO TO Statement

The GO TO statement is an unconditional transfer that transfers control to the specified line number.

	ln GO TO ln_1	
where	ln	is the line number of the statement
	GO TO	is the statement operator
	ln_1	is the line number of the next line to be executed

In the form shown above, ln_1 may be greater than or less than ln; that is, control may be transferred forward or backward.

Referring to the program in Figure 2-1, it is evident that much editing would be required to evaluate the formula for a large number of values of A, B, and C. Using the GO TO statement, the user could rewrite the program as shown in Figure 3-2 so that a series of incremented values would be "plugged into" this expression.

Rather than stopping after one evaluation of the formula's initial constants, the program has been rewritten so that the values of A, B, and C are incremented or decremented (lines 50-70) and control is transferred back to the PRINT statement of line 40. The PRINT statement evaluates the function for each new set of values and types out the answers at the terminal. Lines 40 through 80 form a "loop" which will be iterated as long as the program runs.

```
?LIST

10  LET A=1
20  LET B=2
30  LET C=-4
40  PRINT A,B,C,"ROOT IS "((-B+(B↑2-4*A*C)↑.5))/(2*A)
50  LET A=A+1
60  LET B=B+1
70  LET C=C+(-1)
80  GOTO 40
90  END

?RUN
   1          2          -4         ROOT IS      1.23607
   2          3          -5         ROOT IS      1
   3          4          -6         ROOT IS       .896805
   4          5          -7         ROOT IS       .838037
   5          6          -8         ROOT IS       .8
   6          7          -9         ROOT IS       .773235
   7          8         -10         ROOT IS       .753374
   8          9         -11         ROOT IS       .738041
   9         10         -12         ROOT IS       .72584
  10         11         -13         ROOT IS       .715898
  11         12         -14         ROOT IS       .707641

50 BREAK
```

Figure 3-2. Example of Unconditional Control Statement (GO TO)

The PRINT statement has also been modified to print out four values, the values A, B, and C and the formula evaluation for ROOT. Since more than one value is used for each dependent variable, this helps in reading the output. The carriage is returned after each execution of the PRINT statement so that four columns of numbers are printed out.

There is only one problem with the program as it stands: there is no instruction to stop the program from looping. A system interrupt (depressing the SS1 switch on the front panel of the computer) must be used to halt execution of the program and return the system to the command mode after BREAK is printed at the console. The conditional control statements to be discussed subsequently can be used to set up and test for certain conditions, instead of using the system interrupt.

Two-Branch IF Statement

The IF statement is a conditional control statement which causes a transfer of control only if the condition specified by the relational expression is met; otherwise, control continues normally.

When this "conditional GO TO" statement is executed, the two expressions are evaluated and compared. If the relationship stated (e_1 op e_2) is true, control is transferred to ln_1; otherwise, control goes on to the next statement as it normally would.

The IF statement has the following form:

ln IF e_1 op e_2 THEN ln_1	ln IF e_1 op e_2 GO TO ln_1
ln IF e_1 op e_2 THEN sn	

where	ln	is the line number of the statement
	IF	is the statement operator
	e_1, e_2	are any expressions which will be evaluated
	op	is any one of the six relational operators: =, <, >, <=, >=, <>
	THEN GO TO	are used to separate the relationship (e_1 op e_2) from the line number
	ln_1	is the line to which control is transferred if the stated relationship is satisfied
	sn	is the BASIC statement

In the program segment:

```
200 IF X>100 GO TO 500
210
.
.
.
.
```

the stated condition is whether X is greater than 100. If X is indeed greater than 100 (that is, equal to 100.001 or more), control passes to line number 500. If X is less than or equal to 100, control continues normally to line 210.

Figure 3-3 shows a program that evaluates the formula $X^2 + 1.2\ X + 4$ for X values of 5, 6, 7, 8, 9, and 10. The IF statement in line 40 causes the PRINT statement to be executed as long as X is less than or equal to 10. When X becomes 11, the IF statement transfers control to line 50, where the program exits.

```
?LIST

10  LET X=5
20  PRINT X,X↑2+1.2*X+4
30  LET X=X+1
40  IF X<=10 GOTO 20
50  END

?RUN
    5            35
    6            47.2
    7            61.4
    8            77.6
    9            95.8
   10           116

50 EXIT
```

Figure 3-3. IF Statement

Some further examples of two-branch IF statements are:

```
100 IF (X+Y) /2 = 5 THEN 80
110 IF (X1+X2+X3) /3<5.6/(D*D) GO TO 500
120 IF N5 <>N6 THEN 50
```

In an IF statement of the form

ln IF e_1 op e_2 THEN s

where s is a basic statement, two points are worth noticing:

1. Consider a line of the form

   ln IF e_1 op e_2 THEN s1: s2

If the condition is met, then s1 and s2 are executed. If the condition is not met, s1 and s2 are not executed, since control passes to the next line (as opposed to the next statement).

2. A statement of the form

   ln IF e_1 op e_2 THEN GO TO ln_1

is valid.

Three-Branch IF Statement

The three-branch IF statement branches to one of three possible line numbers. The line number which is used depends on how a specified expression compares with 0.

```
ln IF e, ln1, ln2, ln3
```

where	ln	is the line number of the statement
	IF	is the statement operator
	e	is the expression to be evaluated
	ln_1, ln_2, ln_3	are line numbers to which control may be transferred; line ln_1 is executed next if $e<0$, ln_2 if $e=0$, and ln_3 if $e>0$.

Control is transferred to one of the three line numbers, depending on the arithmetic value of the expression e. If e is negative, control is transferred to line number ln_1; if e is 0, control is transferred to line number ln_2; and if e is positive, control is transferred to line number ln_3. A statement like the general form above is equivalent to

```
IF e<0 GO TO ln1
IF e>0 GO TO ln3
GO TO ln2
```

As an example, the statement

```
100 IF X↑2-16, 10, 50, 600
```

will transfer control to line 10 if the absolute value of X is less than 4, to line 50 if X is equal to +4 or -4, and to line 600 if the absolute value of X is greater than 4.

The three line numbers need not all be different. To test only for X equal to +4 or -4, the above line could be rewritten

```
100 IF X↑2 - 16,  600, 50, 600
```

Control goes to line 600 unless X is equal to +4 or -4. Line 100 also shows that control can be transferred forward (600) or backward (50).

Additionally, the three-way branch can be converted to a two-branch IF statement by assigning the same statement number to two of the three statement numbers in the list, e.g.,

```
5 IF (I+46), 30, 30, 32
9 IF (R), 4, 12, 12
```

EXAMPLES OF USE

Figure 3-3 has illustrated how instructions can be repeated with different values of the variables involved. One of the computer's most powerful features is this ability to repeat the steps in a solution to a problem. As we have indicated, this is called iterating or looping.

Iterating has been accomplished in Figure 3-3 by use of an IF statement. The PRINT and LET statements are repeated a specified number of times. The general form of a loop involves two statements (an IF statement and a LET statement) plus a variable designated as a counter (X). The counter is used to decide when the proper number of repetitions have been made; in Figure 3-3, when the counter becomes greater than 10, the looping is complete. One statement (the IF statement) is used to test for completion of the number of required iterations. The second statement involved is a LET statement used to increment the counter.

The bases of the FOR-NEXT statements are the three elements of a loop: a counter, an end test, and an increment. When a block of instructions is to be executed repeatedly, the FOR statement precedes the block, and the NEXT statement is the first instruction following the block. A variable is chosen to be a counter, and its value is increased each time through the loop (i.e., the block of instructions is repeated). As long as the counter does not exceed a specified value, the instructions between the FOR and NEXT statement are re-executed.

When a FOR statement is scanned, expressions e_1, e_2 and e_3 are evaluated and their values saved. The simple variable, v, is then given the value of the first expression and control is transferred to the following statement line.

For example:

```
50 FOR I = 1 TO 10 STEP 2
60
        .
        .
100   NEXT I
```

is equivalent to

```
50 I = 1
60
        .
        .
100 I = I+2: IF I<=10 THEN GO TO 60
```

A FOR statement has the following form:

ln FOR v=e_1 TO e_2 STEP e_3	ln FOR v=e_1 TO e_2

where	ln	is the line number
	FOR	is the statement operator
	v	is the simple variable (variable without a subscript) which is used as a counter and takes on various values
	e_1	is an expression which is evaluated and assigned as the initial value of the variable (v)
	e_2	is an expression which is evaluated and is the terminal value of the variable (v)
	e_3	is an expression which is evaluated and is the value by which the variable (v) is increased with each repetition of the loop; this value is assumed to be 1 if it is omitted
	STEP, TO	are expression separators which may be replaced by commas

The FOR statement says that the statements following it, up to the corresponding NEXT statement, are to be repeated. The first time the statements are executed, the variable v will be equal to e_1; the second time, the variable will be equal to $e_1 + e_3$; the third time, the variable will be equal to $e_1 + 2e_3$; and so on. The last value of the variable is the greatest value the variable can reach without exceeding e_2.

Every FOR statement must have a corresponding NEXT statement. The NEXT statement must be executed after FOR is encountered. This example is perfectly legal:

```
 70 GO TO 100
 80 NEXT I
 90 GO TO 130
100 FOR I = 1,10
110 PRINT I
120 GO TO 80
130 ...
```

A NEXT statement takes the following form:

```
ln NEXT v
```

where	ln	is the line number
	NEXT	is the statement operator
	v	is the same variable named in the corresponding FOR statement

When a NEXT statement is encountered, the simple variable, v, is incremented by the saved value of the third expression, e_3 (or by 1 if e_3 was not specified). If the new value is within the inclusive range of the first two values saved for e_1 and e_2, control is transferred to the statement following the FOR statement. If not, the following NEXT statement is executed.

The examples in Figures 3-2 and 3-3 could have been written using the NEXT statements as illustrated in Figure 3-4.

```
?10 LET X=5
?20 PRINT"X IS",X,"F(X) IS ", X↑2+1.2*X +4
?30 LET X =X+1
?40 IF X<=10 GO TO 20
?50 PRINT
?60 PRINT EN←←"END OF RUN"
?70 END
?RUN

THIS PROGRAM EVALUATES THE FORMULA:
F(X)=X↑2+1.2X+4

X IS              5          F(X) IS          35
X IS              6          F(X) IS          47.2
X IS              7          F(X) IS          61.4
X IS              8          F(X) IS          77.6
X IS              9          F(X) IS          95.8
X IS             10          F(X) IS         116

END OF RUN

70 EXIT
?
```

Figure 3-4. FOR-NEXT Statement Usage

Since the STEP size was 1, statement 10 could have been written:

```
10 FOR X=5 TO 10
```

and a STEP size of 1 would have been assumed.

An abbreviated form of the FOR statement may be used:

```
ln FOR v=e1, e2, e3
        or
ln FOR v=e1, e2
```

The words TO and STEP have been replaced by commas. Again, if e_3 is not specified, it is assumed to be 1. The following statements are legal FOR statements, assuming there is a corresponding NEXT statement:

```
10 FOR A9 = Y↑2+3 TO 9*C+23.5 STEP .5
20 FOR C = 15.2,-7.8,-.2
30 FOR N = -8 TO 23
```

NEGATIVE STEP SIZE

Several comments can be made about FOR-NEXT loops. The expressions e_1, e_2, e_3 can take any value, positive or negative, whole number or fraction. As noted above, when a NEXT statement is encountered, the value is incremented or decremented (depending on e_3) and tested within the inclusive range of the first two values saved for e_1 and e_2. Thus the statement in the loop

```
FOR C=15.2,-7.8,.2
     :
NEXT C
```

would be executed once, since 15.2+.2 is out of inclusive range

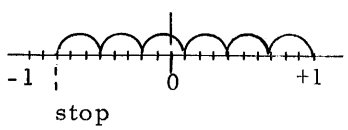
$$-7.8 \leq C \leq 15.2$$

When a negative step size is used, the last value the loop variable has is the least value not less than the stopping value (e_2). This is the opposite of the positive step size. When a positive step size is used, the last value the loop variable has is the greatest value not more than the stopping value (e_2). The easiest way to picture this is that the counter will not step past the stopping criteria no matter in which direction (+ or -) it is moving. For example, the loop

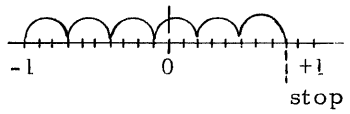
```
10 FOR S = +1 TO -1 STEP -.3
```

would be executed for values of S=1, .7, .4, .1, -.2, -.5, -.8. The variable S would not take the value -1.1, because this is a negative step size loop and -1.1 is less than the stopping value of -1. In summary:

```
FOR C =+1, -1, -.3
```



FOR C =-1, +1, +.3



NESTING LOOPS

FOR-NEXT loops may be placed inside other FOR-NEXT loops. When loops are being nested, each must appear entirely inside another. During execution, the inner loop is fully executed each time the outer loop is executed. FOR-NEXT iterative loops may be nested to any depth. Figure 3-5 shows legal and illegal uses of nested FOR loops.

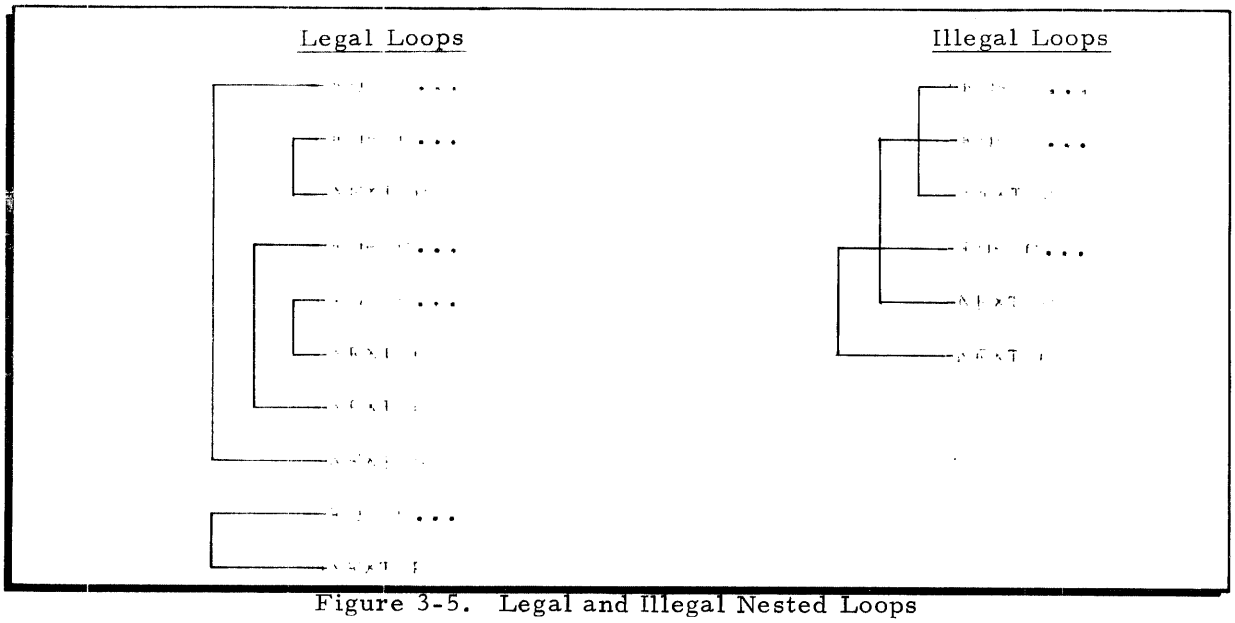


Figure 3-5. Legal and Illegal Nested Loops

The program in Figure 3-6 illustrates the result of nesting FOR-NEXT loops. The three loops begin at lines 10, 20, and 30 and end at lines 60, 50, and 40, respectively. The only purpose of each loop is to print out the loop counter and its current value for each iteration of the loop. With the use of TABs, the printout graphically illustrates the result of nesting FOR-NEXT loops. The inner loop is executed once for each iteration of the middle loop, and the middle loop is executed once for each iteration of the outer loop.

```
?LIST

10  FOR X=1 TO 3
15  PRINT " X=";X
20  FOR Y=8.4 TO 8.8 STEP .2
30  PRINT TAB(5);" Y=";Y
40  FOR Z=1 TO -1 STEP -1
50  PRINT TAB(10);" Z=";Z
60  NEXT Z
70  NEXT Y
80  NEXT X
90  STOP
100  END
```

Figure 3-6. Nested FOR-NEXT Loops

The outer loop calls for three iterations and is executed in its entirety once (a total of three print lines). The middle loop also calls for three iterations but is executed in its entirety nine times (a total of 27 print lines).

All loops illustrated are defined using constants; however, variables and expressions may also be used as loop parameters. For example, line 10 (10 FOR X = 1 TO 3) could be replaced with:

```
10 LET X1 = 1
13 LET X2 = 3
16 FOR X = X1 TO X2 STEP X2-2*X1
```

In the original program, the step size was omitted because it was 1.

The middle loop (line 20) illustrates a loop with fractional values. The terminal value for a loop need not be exact, because it will not be exceeded. For example, line 20 could be written as

```
20 FOR Y - 8.4 TO 8.9 STEP .2
```

with the results illustrated in Figure 3-7. The fourth value the counter would take in the loop would be 9.0. The terminal value, 8.9, would stop the loop before that value was used.

The inner loop (line 30) illustrates a negative step size and that the initial and final loop values need not be of the same sign.

Each loop in the example has three iterations. It is not necessary for inner loops to iterate the same number of times as outer loops.

```
?RUN
 X=     1
       Y=     8.4
              Z=     1
              Z=     0
              Z=    -1
       Y=     8.6
              Z=     1
              Z=     0
              Z=    -1
 X=     2
       Y=     8.4
              Z=     1
              Z=     0
              Z=    -1
       Y=     8.6
              Z=     1
              Z=     0
              Z=    -1
 X=     3
       Y=     8.4
              Z=     1
              Z=     0
              Z=    -1
       Y=     8.6
              Z=     1
              Z=     0
              Z=    -1

90 EXIT
?
```

Figure 3-7. Results of Nested FOR-NEXT Loops Example

ON Statement Conditional Control

The ON statement transfers control to one of a number of possible lines, depending on the value of an expression.

ln ON e GO TO ln_1, ln_2, ..., ln_n

where	ln	is the line number of the ON statement
	ON	is the statement operator
	e	is an expression to be evaluated and truncated to an integer; this value must be greater than or equal to 1 and less than n+1
	GO TO	separates the expression from the line numbers which follow
	ln_1, ln_2, ..., ln_n	is a list of n line numbers separated by commas

When this statement is encountered, the expression e is evaluated and truncated to the greatest integer less than or equal to the expression. If the value of this integer is 1, control is transferred to line ln_1; if the value is 2, control goes to line ln_2; and so on to a value of n which transfers control to ln_n. If the value of the expression is less than 0 or greater than n, the error message ON ERR is typed at the terminal, and the program returns to the command mode. The ON statement

```
50 ON 2+SGN (X) GO TO 100, 200, 300
```

is equivalent to

```
50 IF X, 100, 200, 300
```

An error, SN, also results if one of the line numbers in the list is not present in the program.

The ON statement in the example of Figure 3-8 will transfer control to line 35, as soon as the expression 2↑N yields a value greater than one after a single iteration in the loop. The statement detects the value 1.07177, truncates it to 1, and branches to the PRINT statement. WE GOT THERE is printed out at the teletype and the program exits.

```
?JOB
?10 FOR N=0.1 TO 1.0
?20 PRINT 2↑N
?30 ON 2↑N GO TO 50
?40 NEXT N
?50 PRINT "WE GOT THERE!"
?60 END
?RUN
     1.07177
 WE GOT THERE!
60 EXIT
?
```

Figure 3-8. Example of ON Control Statement

Whenever a STOP or END is encountered, the line number and the word EXIT are printed. If the program terminates by executing the highest numbered statement without encountering either a STOP or an END statement, line number 0000 and EXIT are printed.

END Statement

When executed, the END statement returns control to the command mode. At the same time, the line number of the statement, followed by the word EXIT, is typed at the terminal.

	ln END	
where	ln	is the line number of the statement
	END	is the statement operator

The END statement need not necessarily be executed, but if it is, control is returned to the command mode. If an END statement appears in the middle of a BASIC program, BASIC stops executing the program. When executed, the statement

```
600 END
```

causes the following terminal typeout:

```
600 EXIT
```

STOP Statement

The STOP control statement, when executed, has the same effect as the END statement. END and STOP are equivalent in Series 16 BASIC.

	ln STOP	
where	ln	is the line number
	STOP	is the statement operator

The STOP statement may appear anywhere in the program. The line

```
345 STOP
```

would cause the following typed output at the terminal:

```
345 EXIT
```

Figure 3-9 shows the STOP and END statements used in a BASIC program. In this example, the STOP instruction appears in the middle of the program and is the instruction which causes the program to halt execution. The END instruction is the last line of the program although it is never executed.

The stopping criterion (X+99=0) demonstrates the often used programming technique of "keying" on a special value (-99 for example). The data pool in the above program can contain any number of values as long as the last value and only the last value is -99

(the key value). Any value could have been set aside as a "key" to prevent a DATA exit. Whenever the READ instruction tries to get too much data from the data pool, the program exits to the command mode. This could be a nuisance if further processing of the data is desired. When the amount of data that will be in a data pool is unknown, the end of the data pool can be marked with a special value and the BASIC instructions can test for this value.

```
?LIST


3  PRINT
4  PRINT "THIS PROGRAM EVALUATES THE FORMULA:"
5  PRINT "F(X)=X↑2 +1.2X +4"
6  PRINT
10  DATA 72,18,15,-99
20  READ X
30  IF X+99,50,40,50
40  STOP
50  PRINT "X IS ";X;"F(X) IS ";X↑2+1.2*X+4
60  GOTO 20
70  END


?RUN

THIS PROGRAM EVALUATES THE FORMULA:
F(X)=X↑2 +1.2X +4

X IS     72  F(X) IS  5274.4
X IS     18  F(X) IS   349.6
X IS     15  F(X) IS   247

40 EXIT
```

Figure 3-9. STOP and END Statements

INPUT/OUTPUT STATEMENTS

The input/output statements allow the user to input data to a program, output the results of the computation, and restore the original data to its initial condition before the program was executed. There are five I/O statements:

a. DATA

b. READ

c. RESTORE

d. INPUT

e. PRINT

Of these, DATA is more properly classified as a specification statement but is described here because of its relevance to all the other I/O commands.

READ and DATA Statements

The READ statement is used in conjunction with the DATA statement to process a list of numeric items. The DATA statement defines a list of values (called a data pool) and the READ statement assigns values from the DATA list to specific program variables. Numeric items in a data pool have a specific order (determined by the DATA statement) which the READ statement uses during the execution of the program. DATA statements, on the other hand, are ignored during execution.

The DATA statements in a program form a pool of data items (in the form of constants) which may be scanned by READ statements.

```
ln DATA c1, c2, . . ., cn
```

where	ln	is the line number
	DATA	is the statement operator
	c_1, c_2, ..., c_3	is a list of numeric constants; these numeric constants may be in any of the three formats allowed for constants

When executed, a READ statement causes the variables in the READ statement to be assigned the next consecutive values from the data pool created by the DATA statement. The scanning of the DATA statement proceeds from left to right, and if no unused data items remain, an error message is printed and control is returned to the command mode.

```
ln READ v1, v2, . . ., vn
```

where	ln	is the line number
	READ	is the statement operator
	v_1, v_2, ..., v_n	is the list of variable names which are to be assigned values; there must be at least one variable in the list

Note that the list of a READ statement is composed entirely of variable names; it has no constants or expressions. The list of a DATA statement consists only of constants. Expressions and variables can be used in a data list. Expressions will be evaluated and variables are replaced by the value they have at time the READ statement is executed.

The items in the list of a DATA statement are used as required by READ statements. There may be more than one DATA statement in a program, with each consecutively numbered statement adding to the end of the data pool. The first constant in the lowest numbered DATA statement is the first constant in the data pool. The last constant of the highest numbered DATA statement is the last constant in the data pool.

The DATA statements themselves are processed sequentially, as are all BASIC statements. A DATA statement may appear anywhere in a program, before or after the READ statements which use the elements of the data pool. During execution, a DATA statement is ignored.

When a READ statement is encountered in a program, the next available elements from the data pool are assigned as the values of the variables in the READ statement. The values from the data pool are assigned as they are needed. When the end of the data pool is reached, the message DATA is typed to indicate that the data pool is exhausted, and the program exits.

The relevant BASIC 16 error message is

ERROR DA LINE XXXX

where XXXX is the line of the READ statement in which a read was attempted after DATA was exhausted. As after all other errors, control is returned to the command mode.

The program in Figure 3-10 demonstrates how READ and DATA statements are used in a BASIC program. Notice that the position of the DATA statements and the number of elements in each statement are unrelated to the READ statement which uses the data pool they create. Each time the READ statement is repeated, a new value of X is assigned, and the function is evaluated.

Reading the data from the data pool does not destroy it. It is possible to reuse the data pool during a BASIC run.

RESTORE Statement

Control over the use of the data pool is accomplished by use of the RESTORE statement. When executed the RESTORE statement returns a data pointer to the first constant in the first DATA statement. In other words, the data pool is "restored" to its original condition.

```
?LIST


3  PRINT
4  PRINT " THIS PROGRAM EVALUATES THE FORMULA:"
5  PRINT "F(X) =X↑2 +1.2X+4"
10  DATA 5.7,.12E06,5,8.9
22  DATA 1.234,.5E11
25  DATA 7,8,9
27  READ X
30  PRINT " X IS ",X," F(X) IS ";X↑2+1.2*X+4
50  GOTO 27
70  END


?RUN

 THIS PROGRAM EVALUATES THE FORMULA:
F(X) =X↑2 +1.2X+4
 X IS              5.7          F(X) IS     43.33
 X IS           .12E 06         F(X) IS   .144001E 11
 X IS              5            F(X) IS     35
 X IS              8.9          F(X) IS     93.89
 X IS              1.234        F(X) IS      7.00356
 X IS           .5E 11          F(X) IS   .25E 22
 X IS              7            F(X) IS     61.4
 X IS              8            F(X) IS     77.6
 X IS              9            F(X) IS     95.8

ERROR DA LINE 27
?
```

Figure 3-10. READ and DATA Statements

ln RESTORE

where ln is the line number of the statement

RESTORE is the statement operator

Note the following program segment:

```
100 DATA 50,67.3,85,-4E3
 .
 .
 .
200 READ C 1,X,Z 5
 .
 .
 .
300 READ Y 4,F,C
 .
 .
 .
400 RESTORE
 .
 .
 .
500 FOR I = 1 TO 8
501 READ A(I)
502 NEXT I
 .
 .
 .
600 DATA 1.5,5E2,-40.,6.E-3
```

The data pool will contain the constants 50, 67.3, 85, -4000, 1.5, 500, -40, and .006.

At line 200, the variables will have the following values after the READ:

C1 = 50

X = 67.3

Z5 = 85

At line 300, the variables will have the following values after the READ:

Y4 = -4000

F = 1.5

C = 500

A_1 = 50, A_2 = 67.3. . . A_7 =-40, A_8 = 6.E-3

INPUT Statement

The INPUT statement requests information from the terminal user.

ln INPUT v_1, v_2, ..., v_n

where	ln	is the line number of the instruction
	INPUT	is the statement operator
	$v_1, v_2, \ldots, v_n$	is a list of variables or array elements separated by commas

When executed, an INPUT statement causes an exclamation point to be typed and the program awaits the entry at the teletypewriter of those data items the user requested in his list. If more than one data item is typed in, the items must be separated by commas. If more items are typed than the items in the list, the extra items are lost and no error indication is given. If fewer are typed than needed, BASIC responds with an exclamation point until the list is completely satisfied.

The input data may be composed of constants or expressions separated by commas. An expression input as a data item may utilize variables defined in the program. The line of input is terminated by a carriage return. The @ and ← characters have their usual effect in a line of input; i.e., the @ allows a line of input to be retyped, and the ← allows a character to be corrected.

*

Figure 3-11 demonstrates the use of the INPUT instruction; each X is input by line 10. Line 20 is an end test; when a value of 0 is input, the program terminates. Each value is requested by an exclamation point. The second input line contains more than the one requested value. The first value is used and the rest are ignored. The fourth input line contains a nonnumeric character. BASIC interprets this as an error and re-executes the INPUT statement. The fifth input line terminates the program.

```
 LIST


3  PRINT
4  PRINT "THIS PROGRAM EVALUATES THE FORMULA:"
5  PRINT "F(X)=X↑2+1.2X+4"
6  PRINT
10  INPUT X
20  IF X,30,50,30
30  PRINT "X IS",X,"F(X) IS ",X↑2+1.2*X+4
40  GOTO 10
50  PRINT
60  PRINT "END OF RUN"
70  END


?RUN

THIS PROGRAM EVALUATES THE FORMULA:
F(X)=X↑2+1.2X+4

!5.7
X IS              5.7        F(X) IS          43.33
!5,6,7
X IS              5          F(X) IS          35
!1200000
X IS           .12E 07       F(X) IS        .144E 13
!X15

ERROR UV LINE 10
?RUN

THIS PROGRAM EVALUATES THE FORMULA:
F(X)=X↑2+1.2X+4

!0

END OF RUN

70 EXIT
?
```

Figure 3-11. Use of INPUT Statement

PRINT Statement

While the READ statement inputs values of item lists, the PRINT statement does the opposite by outputting the values of item lists.

Among the common uses of the PRINT statement are:

a. Printing out the result of a computation

b. Printing a message included in a program

c. Performing a combination of a and b

d. Skipping lines

ln PRINT a_1, a_2, . . . , a_n

where	ln	is a line number
	PRINT	is the statement operator
	a_1, a_2, ..., a_n	is a list of the items separated by commas or semicolons whose values are to be printed out

ITEMS OF PRINT LIST

Figure 3-3 has shown that variables and expressions may be items of a list to be printed out. Among the many types of items which comprise a printout are:

a. Constants — Constants may also be members of a print list, as

```
10 PRINT 5, 6, 7
```

The format or position of the numbers on the output line is determined by the machine. The output from the above line will appear as

```
RUN
     5             6             7

 0 EXIT

?
```

b. Variables — Figure 3-3 illustrates that a variable may be a member of a print list. The value of the variable X was printed as part of the output. Printing a variable does not destroy its value. In Figure 3-3, the variable X still retains its value after it has been printed.

c. Expressions — Figure 3-3 also illustrates an expression as part of a print list. The expression is evaluated and the value printed out. However, the value of the expression is not available for

further computation, because it has not been stored in a variable. If the value of the expression will be needed for further computation, the following lines could be used:

```
15 Y=X↑2+1.2*X+4
20 PRINT X, Y
```

d. Alphanumeric Literals — The PRINT statement can also be used to type out an alphanumeric literal. An alphanumeric literal is any string of characters the user chooses, enclosed in quotation marks. "THIS SENTENCE IS A GOOD EXAMPLE OF A LITERAL." The main use for literals is as a title for computer output. A literal may appear as any list element or as the only list element. (The program from Figure 3-3 may be rewritten as illustrated in Figure 3-12.) The ← (character delete) and @ (line delete) cannot be printed in a literal, because the system responds as usual to these characters. The blank is important in a literal. The blank is as much a part of a literal as any other character.

In Figure 3-12, the first PRINT statement in the program has no list of items following, so it causes a blank line to be typed. Next an explanation of the function of the program is listed, with the output data. If only the data were present, there would be no indication of how it was obtained. The PRINT statement in line 20 illustrates a list of elements, including a variable, an expression, and two literals. The literals are typed exactly as quoted. The output from lines 4 and 5 begins in column 1, the output from line 20 in column 2, because literal X IS begins with a blank (column 1).

SEPARATING PRINT LIST ITEMS WITH COMMA

A comma following a literal or variable in a PRINT statement moves the print head to the next position (column 15, 29, 43, or 57). A comma at the end of a PRINT statement suppresses a carriage return/line feed to the next line. When the comma is used in a PRINT statement, the Interpreter looks at one line on the terminal as composed of five zones of 14 spaces each. Each element of a PRINT statement list is put in a zone (or zones, if necessary with a long literal). The next element of the list is put in the next unused zone. If more than the five zones of one line are needed for a PRINT statement, a new line is started. In Figure 3-12, each of the four elements of the print list of line 20 required one zone each. The literals were each less than 14 characters, so only one zone was used for each of them.

SEPARATING PRINT LIST ITEMS WITH SEMICOLON

The items of a print list may also be separated with semicolons. All items must be followed by a punctuation mark, except the last. Commas and semicolons may be intermixed in the same line. A semicolon following a literal or variable in a PRINT statement does not move the print head at all when that literal or variable is printed. A semicolon at the end of a PRINT statement suppresses a carriage return/line feed to the next line.

The semicolon results in what is called packed output. In packed notation, zones are for the most part ignored. Each element of the list is printed out with only enough spaces before and after to make it readable and distinguishable from the other list items. Figure 3-13 illustrates how the semicolon packed the output data from Figure 3-12.

```
?LIST


3  PRINT
4  PRINT "THIS PROGRAM EVALUATES THE FORMULA:"
5  PRINT "F(X)=X↑2+1.2X+4"
6  PRINT
10  LET X=5
20  PRINT "X IS",X,"F(X) IS ",X↑2+1.2*X+4
30  LET X=X+1
40  IF X<=10 GOTO 20
50  PRINT
60  PRINT "END OF RUN"
70  END


?RUN

THIS PROGRAM EVALUATES THE FORMULA:
F(X)=X↑2+1.2X+4

X IS              5          F(X) IS           35
X IS              6          F(X) IS           47.2
X IS              7          F(X) IS           61.4
X IS              8          F(X) IS           77.6
X IS              9          F(X) IS           95.8
X IS             10          F(X) IS          116

END OF RUN

70 EXIT
?
```

Figure 3-12. PRINT Statement

```
?20 PRINT " X IS "; X; " F(X) IS "; X↑2+1.2*X+4
?RUN

THIS PROGRAM EVALUATES THE FORMULA:
F(X)=X↑2+1.2X+4

 X IS      5   F(X) IS     35
 X IS      6   F(X) IS     47.2
 X IS      7   F(X) IS     61.4
 X IS      8   F(X) IS     77.6
 X IS      9   F(X) IS     95.8
 X IS     10   F(X) IS    116

END OF RUN

70 EXIT
?
```

Figure 3-13. Semicolon in PRINT Statement

A literal is typed exactly as it appears in the PRINT statement, whether it is used with commas or semicolons. If a literal is followed by a semicolon, any remaining part of the zone is used for the next element in the print list. If a comma is used, the rest of the zone is left blank.

NUMERICAL OUTPUT FORMAT IN LISTS

All numbers printed out of program lists are printed to take up as little room as possible and yet maintain reasonable accuracies. Numbers with absolute values between 0.1 and 9999 are printed in the format XXXX.YYYYYY or -XXXX.YYYYYY , with leading and trailing zeros suppressed. The decimal point is suppressed for integral values. Numbers outside the above range are printed in the following format:

.XXXXXXE±ZZ or -.XXXXXXE±ZZ

TABBING

Another item that may be included in a print list is a term to permit spacing across the line.

TAB (e)

where e is an expression evaluated and truncated with a value from 0 to 70 representing a print position along the line

The TAB causes the carriage to space over to the position designated by e. If TAB (50) is an element in a print list, the next zone or list element would start at print position 50 on the terminal. If position 50 has already been passed, the TAB is ignored. Printing continues normally after the TAB. A zone begins at the tab point. The print positions across the line are numbered from 0 to 70.

The TAB may be used to position literals used as headings and columns of numbers, to ensure that all values appear in the correct column.

Figure 3-14 illustrates the three types of constants and the output that results when a semicolon, or TAB is used. Although the semicolon allows more numbers on a line, the comma keeps the decimal points lined up in the columns of numbers. In the output from lines 130 to 230, BASIC automatically supplies carriage returns when they are needed.

```
?LIST

10  LET X=123
20  LET Y=1.23
30  LET Z=.123E11
40  PRINT
50  PRINT X,
60  PRINT Y,
70  PRINT Z
100  PRINT
110  PRINT " USE OF THE COMMA"
120  PRINT
130  PRINT X,Y,Z,X,Y,Z,X,Y,Z
200  PRINT
210  PRINT "USE OF THE SEMICOLON"
220  PRINT
230  PRINT X;Y;Z;X;Y;Z;X;Y;Z
240  PRINT
300  PRINT " USE OF THE TAB"
310  PRINT
320  PRINT TAB(10),X,TAB(20),Y,TAB(40),Z
400  END

?RUN

  123              1.23         .123E 11

 USE OF THE COMMA

  123              1.23         .123E 11         123              1.23
 .123E 11         123               1.23         .123E 11

USE OF THE SEMICOLON

  123       1.23   .123E 11    123       1.23   .123E 11    123       1.23
 .123E 11

 USE OF THE TAB

             123                    1.23         .123E 11

400 EXIT
?
```

Figure 3-14. Uses of PRINT Statement

SPECIFICATION STATEMENTS

Statement Types

Specification statements are the only noncomputational elements of a BASIC program. There are two types: a comment or remarks statement for appending explanatory text to the program; and a dimensioning statement which reserves locations in memory for program data. When a specification statement is executed, no computations are performed and control is passed to the next line.

DIMension Statement

APPLICATION

The DIM (short for dimension) statement is used to declare variables as tables or matrices and to reserve space for them (i.e., declare the size or dimensions they will have). In Series 16 BASIC, the DIM statement may appear anywhere in the program.

ln DIM v_1 ($list_1$), v_2 ($list_2$), ..., v_n ($list_n$)

where	ln	is a line number
	DIM	is the statement operator
	v_1, v_2, ..., v_n	are single letter names
	$list_1$, $list_2$, ...	are each integer constants (no decimal point) which represent the largest value that a particular subscript will ever reach.

The constants in the DIM statement must be integer (no decimal point) numbers.

If a variable, say X, is dimensioned as X(101), then 102 spaces are reserved for X. The elements of X are X(0), X(1), X(2), ..., X(101). Note that array subscripts start at zero. If Y is dimensioned Y(5,5), then 6 x 6 = 36 spaces are reserved for Y. In general, the number of spaces reserved for a singly dimensioned array, V(n), is (n+1); the number of spaces reserved for a doubly dimensioned array, V(n,m), is (n+1)x(m+1); the number reserved for a triply dimensional array, V(n,m,o), is (n+1)x(m+1)x(0+1); etc.

The DIM statement is nonexecutable. When the BASIC program is executed (RUN), the information in the statement is used by the compiler to assign memory locations. The DIM statement need appear only before the dimensioned variables.

A DIM statement is not always necessary. If a single letter is used as a subscripted variable name with no preceding DIM statement, BASIC will automatically set aside memory space for the name. If the letter is used as a table, BASIC automatically sets aside 11 locations (i.e., as if it has been dimensioned 10). The same name cannot be used as both a singly and a doubly subscripted array; however, the same name can be used as an array (table or matrix) and a simple variable.

Care should be taken in the use of the automatic dimensioning feature. If either of the dimensions is to be greater than 10, the name must be included in a DIM statement. If a table or matrix is to be much smaller than the automatic specifications, it may still be included in a DIM statement to save space. For example, a matrix A(3, 3) requires room for 16 values. If the automatic specifications are used, the matrix is dimensioned A(10, 10). This requires 121 values, more than seven times the necessary space.

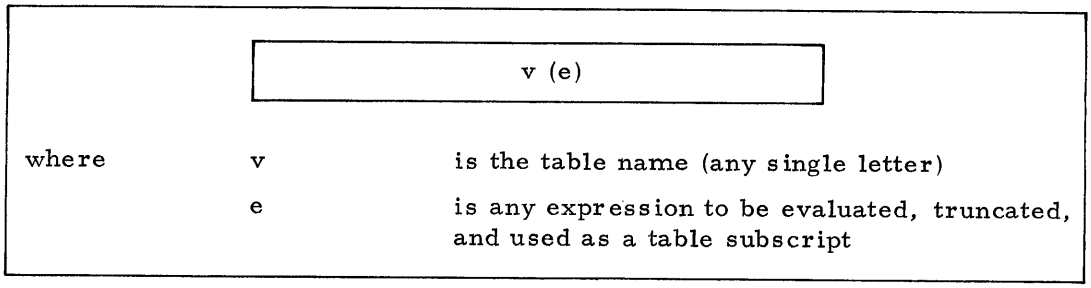
SINGLY SUBSCRIPTED ARRAYS

Any single letter may be a table name. This means there may be up to 26 tables in one BASIC program. Table names such as P2 or XX are illegal. Tables are also called one-dimensional, or singly subscripted, arrays. These names arise because of the way elements are retrieved from a table. Note that a variable can have the same letter as a table; e.g.,

```
10  A = 10
11  A(1) = A+10
```

is legal in the same program.

In order to pick out a particular item in a table, each element is given a number. The first element of a table is numbered 0, the second element is 1, the third element is 2, and so on. When a table element is to be referenced, the name of the table is given followed by the number or subscript of the desired element enclosed in parentheses.



A subscripted variable may be used wherever a variable can be used, e.g., in expression, PRINT statements, etc. Only integers may be used as subscripts, since one specific item is being referred to. A subscript expression is evaluated and truncated (the fractional part dropped); it is not rounded to an integer. Thus a subscript of 1, 99999 refers to v(1) not v(2).

Suppose the user wished to set up the table described above: a table of 12 values, each of which represents the number of television sets a salesman sold in one month of a year. The name of the table will be S. For ease of numbering, the element S(0) of the table is not used in this example. (All the elements in a table need not be used.) The

element S(1) will be the number of sets the salesman sold in January; S(2), the number of sets sold in February; and so on to S(12), the number of sets sold in December. The following BASIC statements contain legal references to table S.

```
10 LET S(3)=12
130 LET Q1=S(1) + S(2) + S(3)
252 PRINT "SETS SOLD IN OCT. =": S(10)
345 IF S(3) - 10, 200, 350, 600
```

In this example, all the subscripts are constants. However, any expression may be used as a subscript.

The convenience of tables is demonstrated in the program segment in Figure 3-15. The coding fills the previously discussed sales table with the appropriate values and computes the sum of all the elements to determine the number of sets the salesman sold in one year. The loop in lines 30 to 60 is used to do this. As each element is read (line 40) from the data pool, it is added to the variable S1 (line 50). After the loop is complete, the total is printed out (line 70).

```
 LIST


10  DATA 10,7,12,10,15,19,18,12,14,5,10,18
15  DIM S(12)
20  LET S1=0
30  FOR C=1,12
40  READ S(C)
50  S1=S1+S(C)
60  NEXT C
70  PRINT " YEARLY SALES TOTAL",S1
80  END


?RUN
 YEARLY SALES TOTAL            150

80 EXIT
?
```

Figure 3-15. Subscripted Variables

The DIM statement (line 15) is used to reserve space for a table in the computer's memory. This type of statement will be discussed more thoroughly later. A subscript expression value must always be less than or equal to the space reserved for the array and greater than or equal to 0.

Without the facility of tables, the instructions to accomplish the task performed in Figure 3-15 would have been considerably longer. For example, the READ statement would have contained 12 variables rather than the "one" it contains. The arithmetic statement to compute the sum would have contained 12 variables with 11 plus signs. If any amount of computation had to be done, a tremendous amount of typing would be necessary.

DOUBLY SUBSCRIPTED ARRAYS

Suppose there is more than one salesman selling television sets. When it is desired to sort data by two categories (e.g., salesman and month), a double subscripted, or two-dimensional, array may be used.

A matrix, in BASIC, is a set of elements, each of which is referenced by two subscripts. Figure 3-16 illustrates matrix A with dimensions m x n. In BASIC, 0 is a subscript, so the matrix illustrated contains (m+1)x(n+1) elements.

	column 1	column 2	column 3		column n+1
row 1	$a_{0,0}$	$a_{0,1}$	$a_{0,2}$	. . .	$a_{0,n}$
row 2	$a_{1,0}$	$a_{1,1}$	$a_{1,2}$	. . .	$a_{1,n}$
row 3	$a_{2,0}$	$a_{2,1}$	$a_{2,2}$	. . .	$a_{2,n}$
⋮	⋮	⋮	⋮		⋮
row m+1	$a_{m,0}$	$a_{m,1}$	$a_{m,2}$	. . .	$a_{m,n}$

Figure 3-16. Matrix A (m, n)

The first subscript defines the row the element is in and the second gives the column.

A matrix is used in the same way a singly subscripted array (or table) is used.

$v(e_1, e_2 \ldots e_n)$

where	v	is the matrix name (any single letter)
	e_1,	are expressions to be evaluated, truncated, and used as subscripts
	e_2,	
	e_3...	
	e_n	

Again, a matrix reference may be used anywhere a variable or table can be used. The subscript expressions e_1 and e_2 follow all the rules for subscripts of tables. They must be greater than or equal to 0 and less than or equal to the matrix dimensions.

Suppose there are three salesman selling television sets and their sales records are as summarized in Table 3-1. The program illustrated in Figure 3-17 was written to evaluate the performance of each salesman, using the information in Table 3-1 as input.

The output from the program includes the total sets each salesman sold and the average number of sets sold per month by each salesman. The chart following this output is just the input written neatly in columns. The numbers marked with an asterisk in Figure 3-18 denote values which are below the salesmen's monthly quota of 13 sets.

Table 3-1. Sales Totals

Month	Salesman 1	Salesman 2	Salesman 3
January	10	10	12
February	7	5	9
March	12	10	12
April	10	15	11
May	15	17	12
June	19	22	17
July	18	20	15
August	12	15	14
September	14	10	14
October	5	5	10
November	10	7	12
December	18	20	16

The program begins with three DATA statements; each one contains the sales record of one salesman. Although BASIC does not care how many data items are contained in each DATA statement (as long as no more than one line is used), the data has been divided so each statement contains 12 values, the sales record for one salesman.

The DIM statement sets aside space for three subscripted variables. The first, the S matrix, contains the number of sets sold by the salesmen. The first subscript of the matrix refers to the particular salesman, the second to the month. Thus subscript 2, 5 refers to the sales of the second salesman in May, while 1, 12 refers to the sales of the first salesman in December. The program never uses the 0 subscript of matrix S. Table T has four spaces reserved for it (again the 0 position is not used). This table contains the yearly total sales for each of the salesmen. The last table, A, also with four spaces reserved for it, is used to store the monthly average sales by each of the three salesmen.

The first action of the program is a FOR loop to set the elements of the T table equal to 0. The initial value of 0 is required in T, because line 130 assumes T begins as 0. In the run, it was not necessary to set T initially to 0, because BASIC sets array values to 0 at the beginning of a run. However, if this program were "used" by another program (see the section following on subroutines), the T table might not be 0.

```
LIST

10  DATA 10,7,12,10,15,19,18,12,14,5,10,18
20  DATA 10,5,10,15,17,22,20,15,10,5,7,20
30  DATA 12,9,12,11,12,17,15,14,14,10,12,16
40  DIM S(3,12),T(3),A(3)
50  REM  ZERO OUT TOTAL ARRAY
60  FOR X=1,3
70  T(X)=0
80  NEXT X
90  REM  READ SALES DATA AND CALCULATE TOTAL SALES PER MAN
100  FOR S=1,3
110  FOR M=1,12
120  READ S(S,M)
130  LET T(S)=T(S)+S(S,M)
140  NEXT M
150  NEXT S
160  REM  CALCULATE AVERAGE MONTHLY SALES
200  FOR S=1,3
210  LET A(S)=T(S)/12
220  NEXT S
230  REM  PRINT OUTPUT
240  PRINT
250  PRINT "YEARLY SALES PER MAN"
260  PRINT "     SALESMAN ONE   ",T(1)
270  PRINT "     SALESMAN TWO   ",T(2)
280  PRINT "     SALESMAN THREE",T(3)
290  PRINT
291  PRINT "AVERAGE MONTHLY SALES"
292  PRINT "     SALESMAN ONE   ",A(1)
293  PRINT "     SALESMAN TWO   ",A(2)
294  PRINT "     SALESMAN THREE",A(3)
295  PRINT
296  PRINT "  SALESMAN ONE       SALESMAN TWO      SALESMAN THREE"
297  REM  FOR EACH MONTH (M), PRINT ONE LINE OF DATA
300  FOR M=1,12
310  PRINT TAB(4);S(1,M);
320  IF S(1,M)-13,330,340,340
330  PRINT "*";
340  PRINT TAB(24);S(2,M);
350  IF S(2,M)-13,360,370,370
360  PRINT "*";
370  PRINT TAB(44);S(3,M);
380  IF S(3,M)-13,390,400,400
390  PRINT "*";
400  PRINT
410  NEXT M
413  PRINT
417  PRINT "    ASTERISK (*) IF UNDER QUOTA OF 13 MONTHLY SALES"
420  END

?
```

Figure 3-17. Sales Evaluation Program

```
?RUN

YEARLY SALES PER MAN
     SALESMAN ONE              150
     SALESMAN TWO              156
     SALESMAN THREE            154

AVERAGE MONTHLY SALES
     SALESMAN ONE               12.5
     SALESMAN TWO               13
     SALESMAN THREE             12.8333

  SALESMAN ONE       SALESMAN TWO       SALESMAN THREE
       10  *              10  *               12  *
        7  *               5  *                9  *
       12  *              10  *               12  *
       10  *              15                  11  *
       15                 17                  12  *
       19                 22                  17
       18                 20                  15
       12  *              15                  14
       14                 10  *               14
        5  *               5  *               10  *
       10  *               7  *               12  *
       18                 20                  16

    ASTERISK (*) IF UNDER QUOTA OF 13 MONTHLY SALES

420 EXIT
?
```

Figure 3-18. Sales Evaluation Program Results

The instructions in lines 100 and 150 illustrate the use of nested FOR loops. The outer loop (the S, or salesman, loop) iterates on the subscript of the salesmen. The inner loop (the M, or month, loop) counts through the months. The third parameter in the FOR statements has been omitted because it is 1. The inner loop is executed once each time the outer loop is executed. This means the statements enclosed in the inner loop (lines 120 and 130) are executed a total of 3 x 12 or 36 times. Notice that S has been used as both a single variable and a matrix. In BASIC, a name may represent a subscripted variable and a single variable. The value of the single variable is stored in the machine separately from the values of the subscripted variable. The way the variable name is used (subscripted or not) in an instruction decides what value is being referred to.

The M and S loops read the sales record data and add the appropriate values to get the yearly totals for three salesmen. The inner loop is executed the first time with S equal to 1 (salesman one). The first 12 values are read from the data pool, and since S is always equal to 1, the 12 numbers are added together to get T(1). When the inner loop is complete, the value of S is increased to 2 in the outer loop, and the inner loop is repeated. The next 12 values in the data pool are read from the sales record of salesman

two and added to produce the sum T(2). Again, when the inner loop is satisfied, S is increased to 3, and the inner loop is repeated for salesman three. After this point the outer loop is also satisfied so control continues at line 200 (REM lines are ignored by BASIC).

Lines 200 and 220 are a FOR loop to calculate the average monthly sales for the three salesmen. The total sales, T(S), for each man are divided by the number of months (12) and stored in A(S), the corresponding element of A.

Lines 230 through 290 are all PRINT statements which print out the headings for the output, the average (A), and the total (T) table.

Lines 300 and 410 contain a PRINT routine which takes full advantage of the output control characteristics of BASIC. The lines contain a FOR loop. Each execution of the loop produces one line of output for a month of the year. In addition to printing out the sales per month of each salesman, an asterisk (*) follows any sales values which are below the established quota of 13. TABs are used to position the data so the numbers are in vertical columns. The semicolon is used instead of the comma to make sure the asterisk immediately follows the number that is printed out.

The loop tests each salesman's sales for month M to see whether it is less than 13. If the sales total is less than 13, an extra PRINT line to insert an asterisk is executed. If the sales total is greater than or equal to 13, this PRINT line is skipped, and the next sales value is printed at the position determined by the TAB setting. The extra PRINT statement at line 400 has the effect of a carriage return. The last item of a line is either printed by line 370 or line 390. Each of these lines ends with a semicolon so the carriage is not returned to start a new line. Line 400 causes the carriage to be thrown and a new line to be started.

The program did not use elements such as A(0), S(5, 0), and S(0, 0). Not all the elements of a table or matrix need be used.

RULES FOR DIMENSIONING VARIABLES

There are many uses for dimensioned arrays. The number of subscripts any element can have is unlimited. If any variable is declared as having a specified number of subscripts, every reference to that variable name in the BASIC program must have the specified number of subscripts unless it is used as a simple variable. A name cannot refer to a singly and doubly subscripted array in the same program.

There are two rules to remember when forming subscript expressions (for either tables or matrices):

1. The subscript expression will be evaluated and truncated. If the expression is evaluated as 1.99999, this will be truncated to 1.
2. The value of the subscript expression must not be greater than the corresponding constant in the DIM statement and must not be negative. If it is, BASIC will print

   ERROR AS LINE XXXX

   where XXXX is the line number in which the array was referenced with the subscript out of bounds.

Arrays are stored columnwise as in FORTRAN, with the first subscripted varying most quickly. Thus A(1, 1, 1) is stored as

A(0, 0, 0)
A(1, 0, 0)
A(0, 1, 0)
A(1, 1, 0)
A(0, 0, 1)
A(1, 0, 1)
A(0, 1, 1)
A(1, 1, 1)

This is useful when transferring arrays with FORTRAN subroutines.

SUBROUTINES AND FUNCTIONS

Subroutines and functions are among the most powerful and flexible computational tools available in the BASIC Language. Both are provided to minimize the necessity of rewriting a sequence of code when the sequence is logically required several times in a program. But whereas the function is confined to the calculation of an arithmetic expression one line in length, the subroutine can do more than a mere calculation and can contain a large number of statements. Additionally, a function is named and tested like a variable. A subroutine is not named and may not be used in arithmetic expressions.

Two types of functions are used, those which are intrinsic to or "built" into the language (sine, cosine, exponentiation, etc.) and those which are defined by the user himself.

Series 16 BASIC also contains a provision for calling one of up to ten DAP-16 and FORTRAN IV subroutines through its CALL statement.

Subroutines Within Main Program

Subroutines may occur inside the main program or they may be stored elsewhere in core.

A subroutine is not used in the same manner as a function; it is "executed" by the proper BASIC instruction. The instruction which executes a subroutine is a control statement.

	ln GO SUB ln_1	
where	ln	is the line number of the instruction
	GO SUB	is the statement operator
	ln_1	is the line number of the first statement of the subroutine

When the GO SUB instruction is encountered, control is transferred to the indicated line number ln_1, and the line number of the GO SUB statement is saved for use with a later RETURN statement. When the subroutine has been executed, control returns to the instruction immediately following the GO SUB instruction. The result is that between the execution of lines ln-1 and ln+1, the block of coding beginning at line ln_1 has been executed. The statement of line ln_1 can be <u>any</u> BASIC statement.

Go SUBs can be executed as many times as desired from any point in the program file. BASIC remembers where to return after the subroutine has been executed.

A subroutine knows the variables and functions of the main or calling program; variables and functions need not be redefined in subroutines. Conversely, any variables defined in the subroutine are also known in the main program after execution of the subroutine. A subroutine does not have an argument, whereas a function does.

A special statement causes a subroutine to cease execution and return control to the statement following the last executed GO SUB statement. If no GO SUB statement has been executed, an error return is made to the command mode.

ln RETURN		
where	ln	is the line number of the statement
	RETURN	is the statement operator

A subroutine may have more than one RETURN statement, but every subroutine must have at least one. A RETURN statement does not have to be the last statement of a subroutine. The RETURN statement is the only statement which can return control from a subroutine; a GO TO, IF, or any other control statement cannot be used.

The file in Figure 3-19 contains a main program and a subroutine. The subroutine in lines 1000 to 1090 converts a number from radians to an angle in degrees, minutes, and seconds, and prints out the result. The main program in lines 10 to 70 uses the subroutine three times. Before each call to the subroutine, the value of R is set and used as input to the subroutine in the same way a function argument is used. This can be done because a subroutine is always executed by use of the same variable names.

Generally, as many variables as necessary are set aside as inputs to the subroutine and then set to desired values before each subroutine call. Conversely, variables may be set aside and used as outputs from the subroutine. In Figure 3-19, the variables D (degrees), M(minutes), and S(seconds) could be considered outputs from the subroutine. The subroutine calculates these values; thus they are available to be used in the main program after the subroutine has exited.

Program execution is terminated by the STOP in line 70, while the file is terminated with an END statement in line 2000.

The input R to the subroutine is assumed to be positive. In line 1000, R is first converted to degrees. Lines 1010 to 1030 form a loop to normalize the angle to a value between 0 and 360 degrees. The function INT is used to extract the integer portion for the degree value. Line 1050 takes only the fractional portion of the degree value and multiplies by 60 to get the minutes value. Line 1070 extracts the fractional portion of the minutes value and converts it to a whole number of seconds. Line 1080 prints one line of output. The seconds value is rounded.

```
LIST


10  LET R=.174533E-01
20  GOSUB 1000
30  LET R=3
40  GOSUB 1000
50  LET R=1.5708
60  GOSUB 1000
65  STOP
70  REM  AND SO FORTH THROUGH THE REMAINDER OF CALLING PROGRAM
1000  LET D1=R*57.2958
1010  IF D1<=360 THEN 1040
1020  LET D1=D1-360
1030  GOTO 1010
1040  LET D=INT(D1)
1050  LET M1=(D1-D)*60
1060  LET M=INT(M1)
1070  LET S=INT((M1-M)*60+.5)
1080  PRINT "ANGLE=";D;"DEGREES";M;"MINUTES";S;"SECONDS"
1090  RETURN
2000  END


?RUN
ANGLE=     1  DEGREES     0  MINUTES     0  SECONDS
ANGLE=    98  DEGREES    21  MINUTES    59  SECONDS
ANGLE=    90  DEGREES     0  MINUTES     1  SECONDS

65 EXIT
?
```

Figure 3-19. Subroutine in File with Main Program

A STOP or END statement, when it is executed in a subroutine, functions as it does in the main program; execution terminates and control returns to the system command mode.

Subroutine calls may be made from subroutines also. This is called nested GO SUBs. GO SUBs may be nested up to eight deep; eight GO SUBs can be executed before a RETURN is executed. BASIC remembers where to continue control for each RETURN executed. The first RETURN returns control to the statement following the last GO SUB executed; the second RETURN returns control to the statement following the next to the last GO SUB executed; and so on. If more than eight nested GO SUBs exist, BASIC will print

ERROR GS LINE XXXX

DAP-16 and FORTRAN IV Subroutine CALL Statement

The CALL control statement to execute subroutines written in the DAP-16 assembly language and the FORTRAN IV compiler language differs from the GO SUB statement to execute subroutines written in BASIC.

	ln CALL (sn, $a_1, a_2, \ldots a_n$)	
where	ln	is the line number of the statement
	CALL	is the statement operator
	sn	is a subroutine number from 1 to 10
	a_1 to a_n	are arguments to be passed to the subroutine called

When the CALL statement is executed, control is transferred to the first line of the identified subroutine. When a FORTRAN RETURN or DAP-16 JMP* instruction is executed, control is returned to the statement line following the CALL statement in the BASIC program. For example:

```
 10 CALL (1, A, B)
 80 CALL (4, 10, 20)
175 CALL (9, SIN(X), ABS(X), FNA(X))
```

A complete description and examples of the use of the CALL statement for FORTRAN and DAP programs are given in Section IV, Interface Conventions.

Built-In Functions

Certain commonly used functions have been built in as part of the BASIC language. Functions are used in expressions as are constants and variables. The name of the function is given followed by an expression enclosed in parentheses. This expression is called the function argument. When BASIC encounters a function name in an expression, it evaluates the argument in the parentheses and uses this value to compute the requested function. The functional value is then used to evaluate the remainder of the expression.

Table 3-2 lists the built-in functions of the BASIC compiler. These functions may be used anywhere a variable or constant may legally appear. A function reference in an expression looks much like a table reference.

	nom (e)	
where	nom	is the three-letter name of the desired function
	e	is an expression to be evaluated and used as the function argument

Table 3-2. Built-In BASIC Functions

Function Reference	Mathematical Equivalent
SIN(X)	sin(x) The trigonometric sine of the argument x, where x is in radians.
COS(X)	cos(x) The trigonometric cosine of the argument x, where x is in radians.
ATN(X)	$\tan^{-1}(x)$ The arctangent in radians of the argument x.
EXP(X)	e^{x} The exponential function.
LOG(X)	$\log_e(x)$ The logarithm to the base e of the argument.
ABS(X)	$\lvert x \rvert$ The absolute value or magnitude of the argument x.
SQR(X)	$\sqrt{x},\ x \geq 0$ The positive square root of the argument. The argument must be greater than or equal to zero.
INT(X)	The largest integer less than or equal to the argument x. Only magnitude is considered. INT(+1.5)=1 INT(-1.5)=-2
SGN(X)	The sign of the argument x. x<0 SGN(X) =-1 x=0 SGN(X) =0 x>0 SGN(X) =1
TAN(X)	tan(x) The tangent of the argument x, assuming x is in radians.
RND(X)	A pseudo-random number generates: 1. If X>0, the argument is returned as the random number. 2. If X=0, RND will supply a random number such that $0 \geq / \text{RND}(0) \leq 1$. 3. If X<0, a "fixed" random number is supplied such that $0 \leq \text{RND}(X) \leq 1$. The same random number is generated for each use of RND(X) with the same negative argument X. Options 1 or 3 would probably be used to initiate a sequence of random numbers, after which option 2 would be used repeatedly.

A function reference may be

```
10 LET V = 1-SQR(X)
```

Using a variable as the argument of a function does not destroy the value of the variable. In the line above, X still has its original value. The value of V becomes $1-\sqrt{X}$.

The trigonometric functions (SIN, COS, and TAN) require arguments in radians. The ATN (arctangent) function result in an angle in radians. This angle is between - /2 and 0 if the argument is negative and between 0 and + /2 if the argument is positive.

The SQR (square root) function must have a positive argument. The value of the function is also positive. If the argument is negative, the message ERROR SQ LINE XXXX is typed at the terminal, and control returns to the system command mode.

The result of the function INT is the largest integer less than or equal to the argument:

```
INT ( 1.99999) =1
INT (-1.99999) =-2    (not -1)
```

The RND (random number generator) function generates a random number. If the argument is zero on the first call, the generator is initialized with a computer-defined number. If the argument is nonzero, the generator is initialized on that argument. Whenever the random number generator is initialized on the same number, the same list of random numbers is generated when RND is used. To get a random number after initialization, use an argument of zero. Any time RND (0) is used, a new random number is generated. The random number is from a uniform distribution of numbers greater than or equal to 0 and less than or equal to 1.

All function names consist of three letters. A function argument can be any expression, but there must be one and only one argument. There is only one result for any of the functions described in Table 3-2.

Functions have a higher priority than exponentiation in the operator hierarchy. Functions are evaluated before any other arithmetic operator in the expressions in which they appear. This means the expression used as an argument of a function has a higher priority than the expression of which the function is a part.

A function may be used anywhere an expression can be used — in IF statements, in PRINT statements, and even as an argument of another function. The following statements are examples of how functions may be used:

```
10 LET Z=ATN (X/SQR(1-X↑2))
112 I9=INT(X+.5)
280 IF SQR (ABS(A↑2-B↑2)), 110, 120, 130
600 PRINT "N=", N, "LOG OF N =", LOG (N)
```

If the LOG function is called with a negative or zero argument, the following message will be printed on the teletypewriter:

```
ERROR LG LINE XXXX
```

Programmer-Defined Functions

Besides the functions which are part of the BASIC language, the user is allowed to define his own, one-variable functions. Functions may be defined and redefined throughout a program, the last executed definition taking precedence.

A programmer-defined function reference can appear in any expression where a variable may be used. The value of the function is determined before the other arithmetic operations are performed; therefore, the expression used as the function argument is evaluated first. For example, suppose a program is written containing the following related statements:

```
30 DEF FNE(X)=2*EXP(1-X)
   .
   .
   .
520 LET Y = FNE (A/B)
```

Then, after line 520 is executed, the variable Y will contain the result of the $2e^{(1-A/B)}$ operation.

SECTION IV
INTERFACE CONVENTIONS

INTRODUCTION

BASIC differs from compiler and assembler languages such as FORTRAN and DAP-16 in that its Interpreter does not produce reusable object text from the user's source program. Object text is the machine-language binary representation of the source program text that the user entered into the computer for compilation or assembly. Once compiled or assembled, this object text can be loaded into core memory by a loader or linkage editor, and executed whenever needed.

Further, the compiler or assembler need not be in core with the object text. BASIC, on the other hand, requires its Interpreter's presence in core to reinterpret the source text each time the program is run. However, Series 16 BASIC does provide a subroutine CALL statement that causes a DAP-16 assembly language CALL through a table, CLST, maintained by the BASIC Interpreter. The addresses of up to 10 DAP-16 or FORTRAN IV object text subroutines to be called are entered in this table by the user.

A subroutine number in the parameter list of the BASIC CALL statement (see the discussion of Built-In Function in Section III) tells BASIC which table entry to use. Arguments are transferred to the calling subroutine by an assembly language subroutine called F$AT.

Entry points in a DAP-16 subroutine are declared by use of the ENT or SUBR psuedo-operations. Entries to FORTRAN subroutines are made by the SUBROUTINE statement. Linkages are made by the DAP psuedo-operation, CALL, and by the corresponding CALL statement in FORTRAN. Control is returned to the calling program by an assembly language JMP*instruction or by the FORTRAN statement RETURN.

ARGUMENT TRANSFER SUBROUTINE F$AT

The compiler inserts a call to this subroutine at the beginning of FORTRAN-coded subroutines. F$AT transfers pointers (DACs) to the variables being communicated between the calling program and the subroutine. No call to F$AT is made for subroutines which need no arguments.

Calling a Subroutine

The following sequence is used to call a subroutine which transfers arguments via F$AT. The variables are listed in the same order as in a FORTRAN CALL statement. If there is only one argument, the terminal zero must be omitted:

```
(L  )          CALL    subroutine name
(L+1)          DAC     <first variable>
(L+2)          DAC     <second variable>
                .
                .
(L+n)          DAC     <nth variable>
(L+n+1)        OCT     0              Zero must be omitted for n=1
(L+n+2)                               Return point
```

The DACs to the variables can be indirect pointers; F$AT tracks down the indirect links and transfers a direct pointer. Note that variables themselves are never transferred. The reason for this is that the length of the variable is not known (it could be any length, since arrays are acceptable variables).

Calling F$AT

By convention, the first action of a subroutine is to call F$AT. Therefore, the location preceding the call points to the first argument to be transferred. F$AT transfers the arguments associated with the words following the call to F$AT. Then, F$AT increments the pointer to the calling program so that it now points to the conventional return point (following the zero). For example:

```
(L  )   <name>  DAC     **             Subroutine entry point
(L+1)           CALL    F$AT           Must immediately follow entry
(L+2)           DEC     <number of arguments, n>
(L+3)   <name>  DAC     **             First argument address goes here
                .
                .
(L+n+2) <name>  DAC     **             nth argument address goes here
(L+n+3)                                Return point for F$AT
```

The subroutine call may include extraneous arguments following those used by the called subroutine. Although only the number of arguments specified in L+2 of the call to F$AT are transferred, the return pointer is incremented until it points to the word following the zero in the subroutine call.

EXAMPLES OF SUBROUTINE LINKAGE AND USE

Figures 4-1 and 4-2 are examples which typify the use of FORTRAN IV and DAP-16 subroutines that can be called from a BASIC program. Both subroutines were written for the same purpose, i.e., to list the elements of an array on a line printer, five elements to the line.

The CALL from the BASIC program would be of the form

```
CALL (S, A(I), N)
```

where	S	corresponds to the CLST Table entry containing the address of this subroutine
	A(I)	is the first array element to be printed
	N	is the number of elements to be printed

For example, executing the BASIC statement

```
10 CALL (1, B(0), 10)
```

will cause elements B(0) to B(9) to be printed at the line printer.

```
      SUBROUTINE SUB1 (X, Y)
      DIMENSION X(1)
      CONVERT SECOND ARGUMENT TO INTEGER
      I=Y
      PRINT THE ELEMENTS
      WRITE (4, 1) (X(J), J=1.I)
      FORMAT (5G14.7)
      RETURN
      END
```

Figure 4-1. Example of FORTRAN Subroutine for BASIC Program

```
0001                                        SUBR   SUB1
0002                                        REL
0003 00000    0 000000     SUB1  DAC    **
0004 00001    0 10 00000         CALL   F$AT
0005 00002    000002             OCT    2
0006 00003    0 000000     X     DAC    **
0007 00004    0 000000     Y     DAC    **
0008 00005    0 10 00000         CALL   L$22
0009 00006   -0 000004           DAC*   Y
0010 00007    0 10 00000         CALL   C$21
0011 00010    0 04 00035         STA    I
0012 00011    0 10 00000         CALL   F$W4
0013 00012    0 000037           DAC    F1
0014 00013    0 02 77777         LDA    -1
0015 00014    0 04 00036   LOOP  STA    J
0016 00015    0415 77            ALS    1
0017 00016    0 06 00003         ADD    X
0018 00017    0 07 00043         SUB    =2
0019 00020    0 04 00034         STA    TEMP
0020 00021    0 10 00000         CALL   F$L2
0021 00022    000002             OCT    2
0022 00023   -0 000034           DAC*   TEMP
0023 00024    0 02 00036         LDA    J
0024 00025    141206             AOA
0025 00026    0 11 00035         CAS    I
0026 00027    0 01 00032         JMP    *+3
0027 00030    0 01 00014         JMP    LOOP
0028 00031    0 01 00014         JMP    LOOP
0029 00032    0 10 00000         CALL   F$CB
0030 00033   -0 01 00000         JMP*   SUB1
0031 00034    000000       TEMP  BSZ    1
0032 00035    000000       I     BSZ    1
0033 00036    000000       J     BSZ    1
0034 00037    124265       F1    BCI    4,(5G14.7)
     00040    143661
     00041    132256
     00042    133651
0035 00043    000002             END

NO ERRORS IN ABOVE ASSEMBLY.
DAP-16 REV. E                  09-10-6
```

Figure 4-2. Example of DAP-16 Subroutine for BASIC Program

SECTION V
OPERATING PROCEDURES

INTRODUCTION

This section presents detailed operating procedures for loading and executing the BASIC Interpreter and for entering and running user programs and subroutines.

STAND-ALONE VERSION

Loading Interpreter

The BASIC System is provided on a self-loading paper tape, which is loaded as follows:

1. Refer to Figure 5-1, which depicts the control console of a Series 16 general purpose computer. Set the MA/SI/RUN switch to SI and depress the MASTER CLEAR pushbutton.
2. Select the P register by depressing the P/Y switch and load 000001_8 in the P register by depressing illuminated pushbutton 16.
3. Insert the self-loading tape into the appropriate input device.
4. Set the MA/SI/RUN switch to RUN and push START. (When loading with an ASR-33 teletype unit, the manual START switch on the device must be activated. When loading with an ASR-35 teletype unit, the MODE switch must be set to KT.)

Executing Interpreter

After the BASIC Interpreter has been loaded, execute it by proceeding as follows:

1. Set MA/SI/RUN to SI.
2. Press MASTER CLEAR.
3. Select the P Register by depressing the P/Y switch.
4. Select location 1000_8 by depressing the illuminated pushbutton marked "7" above and "A" below.
5. Set MA/SI/RUN to RUN.
6. Press START.

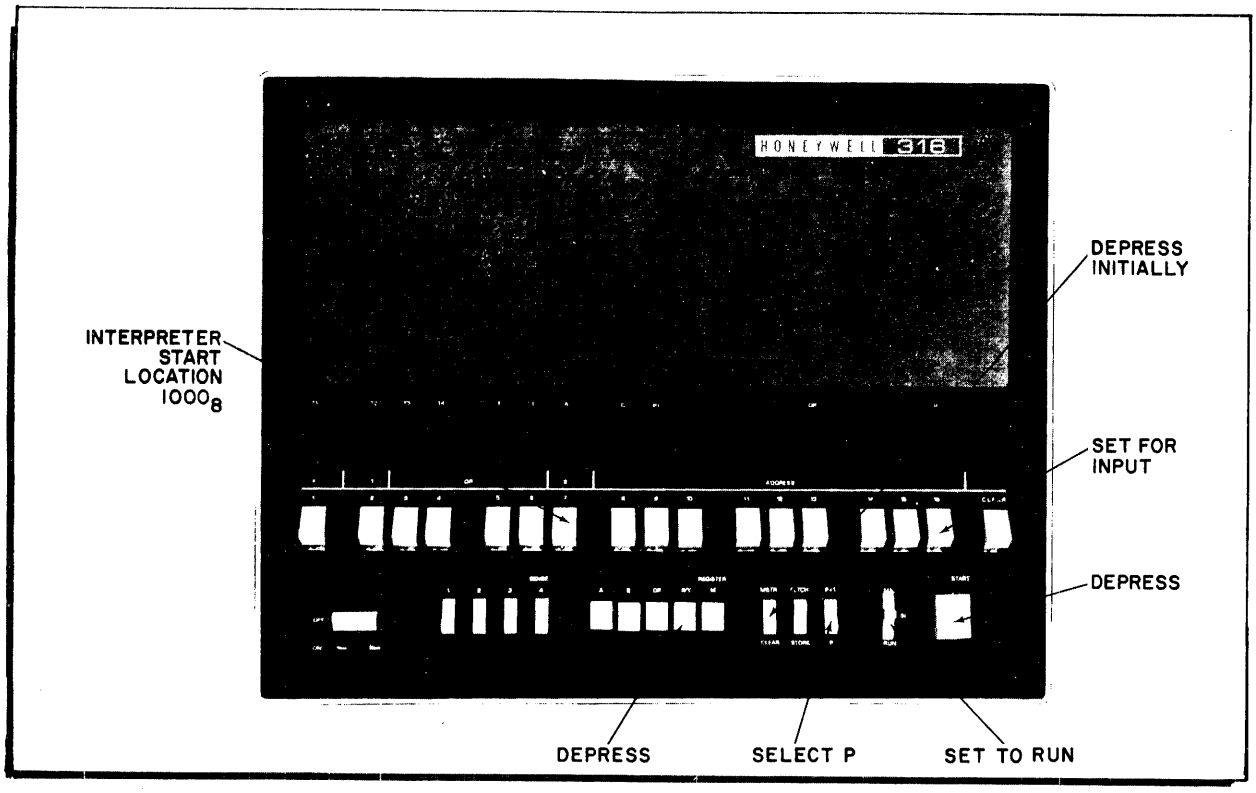


Figure 5-1. Model 316 Control Panel

OP-16 VERSION OF BASIC

Configuring BASIC Interpreter Under OP-16

In order to execute the BASIC Interpreter under the OP-16 Operating System's RTX-16 Executive Program, it is necessary to make five 1-word entries in the Executive Program List Table, XPLT. The table entries, which are described in full in Section 4 of the OP-16 Users Guide, are the following:

```
XPLT    BCI     1, BI           WORD 1 — PROGRAM NAME
        XAC     BASIC           WORD 2 — STARTING ADDRESS
        OCT     0               WORD 3 — STATUS WORD
        VFD     5,0,8,S,3,1     WORD 4 — OPTION WORD
        OCT     MASK            WORD 5 — COORDINATION WORD
```

where S is the number of the starting core segment into which BASIC is to be loaded. BASIC must be loaded above the point specified by the parameter XSPT in the Configuration Module. The Coordination Word, MASK, is specified so that BASIC does not conflict with other programs using the teletypewriter. The value of this word will of course depend on the convention that the user employs at his installation for device coordination. BASIC does not utilize the Communication Word option.

Loading OP-16 Version of BASIC

Using the LDR-APM loader program, load the following five components of BASIC, starting above the point specified by S in the Option Word of the XPLT entry for BASIC.

BASIC-16-B

BASIC-IOS-C

BASIC-MATHPAK

BASIC-INIT-B

DAP-16 object tape of the following subroutine source code:

```
            SUBR    TABLE
            REL
TABLE       DEC     N
            BSS     N
            END
```

where N is the projected size of the user's symbol table. It is suggested that N have a value of 1024 for two sectors minimum. A certain amount of experimentation in selecting this parameter may be needed because of differences in memory size and user programming requirements.

Loading DAP and FORTRAN Subroutines

BASIC maintains a table called CLST which contains 10 entries (CLST, CLST+1, CLST+2...CLST+9), each of which may designate a pointer to some user DAP-16 or FORTRAN IV subroutine. The address of table CLST can be obtained by loading the system tape and examining the contents of 776_8. If the user is configuring his own system the address will appear on the memory map. The user loads these subroutines as follows:

1. Load the LDR-APM loader, starting at some convenient, "out-of-the-way" location.
2. Load the user subroutines (and any subroutines that they may require), starting at a point such that the highest location used will be as close as possible to the end of core (or Bank 1, whichever comes first).
3. To find the address at which the user may start his cross-sector linkage area to Sector 0, load the BASIC system tape and note the address contained in the last nine bits of word '777. The last nine bits are equivalent to the address specified as *BASE on the memory map if the user has configured his own system. The area between the address specified by the contents of location 777_8 and location 1000_8 may be used for cross-sector references for user subroutines.
4. Reload the BASIC system and enter the entry points of the user subroutines in table CLST. If the user has a BASIC system tape, these can be found by examining location 776_8. If the user is configuring his own system, he should consult his system memory map.

   For example, if the user had an 8K system and subroutines which would occupy approximately two sectors of memory, he might accomplish his loading as described in the sample procedure on the following page.

a. Load LDR-APM starting at 11000_8.

b. Load the BASIC System Tape as on page 5-1.

c. Load the subroutines beginning at 15000_8 and start the cross-sector references for the subroutines at the address specified by the contents of the last nine bits of location 777_8.

d. Enter the addresses of the subroutines in table CLST.

e. If the user desires a dump of BASIC and his subroutines, the PAL-AP program should be loaded at 10000_8 and a dump made first from 100_8 to 7577_8 and then from 15000_8 to the highest address occupied by his subroutines.

f. When initiating BASIC, as described below, the user should specify location 14777_8 at the step where he is asked to "GIVE HIGH OCTAL ADDRESS."

Loading and Running BASIC Interpreter under OP-16

Once the BASIC Interpreter has been configured under the RTX-16 Executive routine and the requisite DAP-16 and FORTRAN subroutines have been loaded, the BASIC Interpreter can be executed in the following manner:

1. Call the keyboard program by typing a dollar sign.
2. The system will respond by typing out the inquiry SF=, asking for the System Function you desire.
3. Respond by typing RP for Request Program, followed by a blank and then the program name, B1 for BASIC Interpreter, followed by a carriage return.
4. The system will again type out SF=. Respond by typing @to signify that the Keyboard Program is no longer required.
5. The system will respond to the latter by typing out the program name, followed by the revision level, the date of the revision, a carriage return/line feed, and a question mark. The user may now enter his BASIC commands or statements. The entire interaction would have looked something like this:

```
$
SF=RP BI
SF=@
BASIC-16-B REV.A  09-22-70
?
```

INPUT/OUTPUT AND CONTROL

After the Stand-Alone Only Interpreter has been loaded, but before the computer requests user instructions, it identifies which version of itself the computer is using and the date of that version:

```
BASIC-16  11-19-70
```

It then asks a series of three questions requiring yes or no answers to determine whether your program will need certain intrinsic functions. The functions involved are the trigonometric sine, cosine, tangent, and arctangent and the square root function. For example:

OK TO DELETE THE ATN FUNCTION? (ANSWER YES OR NO)

If these functions are not required, the Interpreter will allow just that much more room for user program statements and data. If an intrinsic function deleted during initialization is called in a BASIC program, the following error message will be printed on the teletype:

ERROR DF LINE XXXX

The last question asked is:

OK TO USE ALL OF CORE? (ANSWER YES OR GIVE HIGH OCTAL ADDRESS)

If the user has loaded subroutines above BASIC, he should specify a location just below the start of his subroutines. If you reply "7777" for decimal 4096, the computer will type out the following message:

414 LOCATIONS FOR USER STORAGE AND DATA

?

The user may now proceed to enter instructions after the question mark, as described below.

After BASIC has been loaded and initialized, it requests input by typing a question mark. The programmer responds with either a statement or a command in the form of

a. an unnumbered system command
b. a line-numbered BASIC statement
c. a line number and carriage return to delete a line
d. an unnumbered BASIC statement for immediate execution

If the statement is to be stored as part of the program, it must have a statement number. If there is no statement number, immediate execution of either a BASIC statement or a system command is performed. Any instruction to the computer is terminated by a carriage return and (optionally) a line feed.

REMark, DATA, DIMension, and DEFINE statements cannot be executed in the immediate mode. Any reference to line numbers (e.g., GO TO or GO SUB) refers to a statement in the program. Thus, immediate statements that transfer control may be used

to restart execution of the program. However, care must be taken that the program has not been modified in any way; otherwise it must be started by a RUN statement rather than a GO TO or GO SUB.

DATA FORMATS

Input Formats

Detailed statement and data formats have been described in Section II, under Syntax. Statements and data must be terminated by a carriage return. A line feed is optional. The sequence X-OFF RUBout may optionally follow the carriage return or line feed. A null line entered during a paper tape LOAD operation terminates that operation.

During input a leftward arrow (←) may be used to delete one or a succession of preceding characters. The symbol @ may be used to delete the entire line.

Output Formats

Unless it is in the LOAD mode, BASIC outputs a question mark each time it is ready to receive an instruction. Each time BASIC is ready to receive an INPUT value, it types an exclamation point, awaits the carriage return following the entry of the variables, and then continues execution of the program.

Statements are listed or punched in line number order, with all spaces except those in comments and messages normalized. Each statement of output is terminated with a sequence of carriage return, X-OFF, and RUBOUT characters. Each new line begins with a line feed. The end of output for the program is signalled by a null statement.

Error Message Formats

Whenever a STOP or END control statement is encountered, the line number and the word EXIT are printed. If the program terminates without a STOP or an END being encountered, the line number 0000 and EXIT are printed.

For each error encountered during statement execution, the Interpreter outputs the following messages:

ERROR AA LINE BBBB

where AA stands for error codes (listed in Appendix A) and BBBB for a line number. An error of the general form in which the user has misplaced or illegally specified a character results in the following error code:

ERROR c? NNNN

where c is the character questioned by the Interpreter. For example, the following line:

```
10 #RINT X
```

would result in the following error message:

```
ERROR #? 0010
```

If a statement executed in the immediate mode contains an error, the line number of the error statement will be zero. Upon detection of this error, control is returned to the command mode.

DETAILED DESCRIPTION OF BASIC SYSTEM COMMANDS

JOB Command

This command clears the user storage space, thereby deleting any previous BASIC program, and prepares the system to accept new information. It does not affect DAP-16 or FORTRAN subroutines nor the subroutine CALL table.

CLEAR Command

CLEAR voids all user data locations but does not delete any programs.

RUN Command

RUN utilizes BASIC to execute the current program. It initializes all array dimensions as specified in DIMension statements and starts interpreting the program at the statement specified or at the lowest numbered statement. Each statement is interpreted and executed as it is encountered. The program is executed until an error, a STOP, an END, an SS1 command, or the last defined statement is reached.

CONTINUE Command

To halt execution of a program or unneeded listing, depress Sense Switch SS1 on the control panel of the computer. The statement number to be executed next and the word BREAK are printed at the console and the computer pauses, awaiting the execution of a CONTINUE command. Executing CONTINUE resumes processing where it left off after the SS1 interrupt. Note that the program may not be restarted with the CONTINUE command if modified subsequent to BREAK.

LIST Command

This command causes the whole or specified portions of the program in core to be listed in line number order.

QUIT Command

This command in the stand-alone version places the machine in the HALT state. To resume operation, press START. In the OP-16 version, QUIT returns control to the RTX-16 Executive.

LOAD Command

A null line entered during a paper tape LOAD operation terminates that operation. The reader will not start if the paper tape is improperly loaded.

On ASR-33, the reader reads one character after the X-OFF before stopping. Therefore it is conventional to follow all X-OFFs with RUB-OUTs.

On ASR-35, the reader may read two additional characters before stopping. However, if the first character after the X-OFF is a RUB-OUT, the second character will not be read.

APPENDIX A
DIAGNOSTICS

Error Code	Meaning
AS	Array subscript out of bounds
DA	Attempt to READ more data than available
DF	Attempt to use a function deletion during initialization
DL	Statement terminator error
DP	Two decimal points in a number
DV	Dummy variable in DEF statement is subscripted
DZ	Divide by zero
FD	Invalid delimiter in FOR statement
FN	Characters FN misplaced in DEF statement
GS	GO SUBs nested more than eight deep
IC	Condition in IF statement is incorrect
ID	General error
IV	Index variable in FOR statement is subscripted
LG	Negative logarithmic function argument
MO	Memory overflow
M,	Missing or misplaced comma
M=	Missing or misplaced equals sign
M)	Missing or misplaced right parenthesis
M(	Missing or misplaced left parenthesis
NO	Numerical overflow
NU	Numerical underflow
NX	Next statement has no matching FOR
ON	Expression in ON statement is nonpositive, as the GO TO is missing
PD	Illegal item delimiter in PRINT statement
RT	RETURN statement not in subroutine
SN	Statement number error (range 1-9999)
SQ	Negative square root function argument
SS	Subroutine selector in CALL out of range (1-10) or subroutine is missing
TH	THEN left out of IF statement
TX	No end of quote

Error Code	Meaning
UF	Undefined function
UM	Unitary minus error
US	Undefined statement number
UV	Undefined variable

APPENDIX B

SYNTACTIC STRUCTURE OF BASIC

The following BASIC syntax is described in Backus Normal Form. Quantities enclosed within angle brackets (<>) are metalinguistic variables representing a class of syntactic variables. A colon followed by an equal sign (:=) means "is defined as." A vertical line (|) connecting two elements means logical OR. An element or group of elements enclosed in square brackets followed by a subscript and superscript ($[\ ]_a^b$) may be repeated any number of times within the inclusive range of the subscript and superscript. All letters and symbols not enclosed in angle brackets are actual characters of the syntax.

<alphabetic character> := 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

<special character> := +|-|*|/|↑|=|(|)|<|>|.|,|;|Δ

<integer> := $[\ \text{<digit>}\ ]_1^9$

<decimal number> := $[\ \text{<digit>}\ ]_1^{N\leq 9}$. $[\ \text{<digit>}\ ]_0^{9-N}$

<sign> := $[+|-]_0^1$

<exponent> := E<sign> $[\ \text{<digit>}\ ]_1^2$

<number> := <integer>|<fraction>|<decimal number> $_1^1$ $[\text{<exponent>}]_0^1$

<signed number> := <sign> <number>

<simple variable> := <alphabetic character> $[\ \text{<digit>}\ ]_0^1$

<fraction> := . <integer>

<subscripted variable> := <alphabetic character>(<expression> $[\ \text{<expression>}\ ]_0^\infty$

<variable> := <simple variable> | <subscripted variable>

<function name> := SIN|COS|TAN|ATN|EXP|ABS|LOG|SQR|INT|RND|SGN|
FN <alphabetic character>

<function term> := <function name> (<expression>)

<term> := <number>|<variable>|<function term>|(<expression>)

<involution factor> : =<term> |<involution factor> ↑ <term>

<multiply factor> : = <involution factor> | <multiply factor> $[* | /]_{1}^{1}$ < involution factor>

<expression> : = <multiply factor> | <sign> <expression> | <expression> $[+ | -]_{1}^{1}$
<involution factor>

<assignment statement> : = LET <variable> $[, \text{<variable>}]_{0}^{\infty}$ = <expression>

<READ statement> : = READ <read list>

<INPUT statement> : = INPUT <read list>

<read list> : = <variable> $[, \text{<variable>}]_{0}^{\infty}$

<DATA statement> : = DATA <number list>

<number list> : = <expression> $[, \text{<expression>}]_{0}^{\infty}$

<RESTORE statement> : = RESTORE

<PRINT Statement> : = PRINT $[\text{<print list>}]_{0}^{1}$

<print list>: =<print item> $[[, | ;]_{1}^{1} \text{<print item>}]_{0}^{\infty} [, | ;]_{0}^{1}$

<print item> : = <expression> | <message> | <message> <expression> | TAB
(<expression>)

<message> : = " $[$ <alphabetic character> | <digit> | <special character> $]_{1}^{\infty}$ "

<comment> : = REM $[$ <alphabetic character> | <digit> | <special character> $]_{0}^{\infty}$

<GO TO statement> : = GO TO <line number>

<GO SUB statement> := GO SUB <line number>

<RETURN statement> : = RETURN

<ON statement> : = ON<expression> GO TO <line number> $[\text{<line number>}]_{0}^{\infty}$

<line number> : = $[\text{<digit>}]_{1}^{5}$

<logical IF statement> : = IF <expression> <relational operator> <expression> $[$ THEN
<statement body> $[$: <statement body> $]_{0}^{\infty}$ | THEN <line number> |
GO TO <line number> $]_{1}^{1}$

<arithmetic IF statement> := IF <expression> <line number>, <line number>,
<line number>

<relational operator> := > | >= | = | = < | < | < >

<FOR statement> : = FOR<simple variable> =<expression> [TO|,] $_{1}^{1}$ <expression>
[[STEP|,] $_{1}^{1}$ <expression>] $_{0}^{1}$

<NEXT statement> : = NEXT <simple variable>

<STOP statement> : = STOP

<END statement> : = END

<DIMENSION statement> : = DIM <dimension variable> [, <dimension variable>] $_{0}^{\infty}$

<dimension variable> : = <alphabetic character> (<size>)

<size> : = <integer> [, <integer>] $_{0}^{\infty}$

<DEFINE statement> : = DEFFN <alphabetic character> (<simple variable>) =
<expression>

<CALL statement> : = CALL (<subroutine identifier> [, <subroutine parameter>] $_{0}^{\infty}$

<subroutine identifier> : = <expression>

<subroutine parameter> : = <expression>

<command request response> : = <system command> | <BASIC statement> | <line deletion> |
<immediate command>

<system command> : = JOB | CLEAR | RUN | LIST | CONTINUE | QUIT | PUNCH | LOAD

<line deletion> : = <line number>

<immediate command> : = <immediate statement> [: <immediate statement>] $_{0}^{\infty}$

<immediate statement> : = <assignment statement> | <READ statement> | <INPUT
statement> | <RESTORE statement> | <PRINT statement> |
<GO TO statement> | <logical IF statement> |
<arithmetic IF statement> | <ON statement> | <FOR
statement> | <NEXT statement> | <GO SUB statement> |
<RETURN statement> | <CALL statement> | <comment
statement> | <STOP statement>

<statement body> : = <immediate statement> | <DATA statement> | <DIMENSION
statement> | <DEFINE statement>

<BASIC statement> : = <line number> <statement body> [: <statement body>] $_{0}^{\infty}$

<BASIC program> : = [<BASIC statement>] $_{1}^{\infty}$ <line number> <END statement>

HONEYWELL INFORMATION SYSTEMS
Technical Publications Remarks Form*

4987

TITLE: SERIES 16 BASIC LANGUAGE

ORDER No.: AB85, REV. 1

DATED: MARCH 1972

ERRORS IN PUBLICATION:

SUGGESTIONS FOR IMPROVEMENT TO PUBLICATION:

(Please Print)

FROM: NAME _____ DATE: _____

COMPANY _____

TITLE _____

_____

_____

CUT ALONG LINE

*Your comments will be promptly investigated by appropriate technical personnel, action will be taken as required, and you will receive a written reply. If you do not require a written reply, please check here. ☐

CUT ALONG

FIRST CLASS
PERMIT NO. 39531
WELLESLEY HILLS,
MASS. 02181

Business Reply Mail
Postage Stamp Not Necessary if Mailed in the United States

POSTAGE WILL BE PAID BY:

HONEYWELL INFORMATION SYSTEMS
60 WALNUT STREET
WELLESLEY HILLS, MASS. 02181

ATTN: PUBLICATIONS, MS 050

Honeywell