AIM 65

MICROCOMPUTER

BASIC LANGUAGE REFERENCE MANUAL

AIM 65

Rockwell International

$5.00
Document No 29650 N49
March 1979

TABLE OF CONTENTS

INTRODUCTION

Before a computer can perform any useful function, it must be "told" what to do. Unfortunately, at this time, computers are not capable of understanding English or any other "human" language. This is primarily because our languages are rich with ambiguities and implied meanings. The computer must be told precise instructions and the exact sequence of operations to be performed in order to accomplish any specific task. Therefore, in order to facilitate human communication with a computer, programming languages have been developed.

Rockwell AIM 65 8K BASIC by Microsoft is a programming language both easily understood and simple to use. It serves as an excellent "tool" for applications in areas such as business, science, and education. After only a few hours of using BASIC, you will find that you can already write programs with an ease that few other computer languages can duplicate.

Originally developed at Dartmouth University, the BASIC language has found wide acceptance in the computer field. Although it is one of the simplest computer languages to use, it is very powerful. BASIC uses a small set of common English words as its "commands." Designed specifically as an "interactive" language, you can give a command such as "PRINT 2 + 2," and BASIC will immediately reply with "4." It is not necessary to submit a card deck with your program on it and then wait hours for the results. Instead, the full power of the computer is "at your fingertips."

We hope that you enjoy BASIC, and are successful in using it to solve all of your programming problems.

ROM INSTALLATION PROCEDURE

Before handling the BASIC ROM circuits, be sure to observe the precautions outlined in Section 1.4 of the AIM 65 User's Guide.

To install the ROMs, turn off power to the AIM 65. Inspect the pins on the two BASIC ROMs to ensure that they are straight and free of foreign material. While supporting the AIM 65 Master Module beneath the ROM socket, insert ROM number R3225 into Socket Z25, being careful to observe the device orientation. Now insert ROM number R3226 into Socket Z26. Be certain that both ROM's are completely inserted into their sockets, then turn on power to the AIM 65.

ENTERING BASIC

To enter and initialize BASIC, type 5 after the monitor prompt is displayed. AIM 65 will respond with:

```
< 5 >

MEMORY SIZE? ^
```

Type the highest address in memory that is to be allocated to the BASIC program, in decimal. End the entry by typing RETURN. BASIC will allocate memory from 530 (212 in hex) through the entered address. If BASIC is to use all available memory, type RETURN without entering an address. The highest address is 1024 (400 hex) in the 1K RAM version of AIM 65, and 4096 (1000 hex) in the 4K RAM version.

BASIC will then ask:

```
WIDTH? ^
```

Type in the output line width of the printer (or any other output device that is being used) and end the input with RETURN.

The entered number may vary from 1 to 255, depending on the output device. If RETURN is typed without entering a number, the output line width is set to a default value of 20, which is the column width of the AIM 65 printer.

BASIC will respond with:

XXXX BYTES FREE

where XXXX is the number of bytes available for BASIC program, variables, matrix storage, and string space. If all available memory was allocated, BASIC will reply with:

494 BYTES FREE (for 1K RAM; i.e., 1024-530)

or

3566 BYTES FREE (for 4K RAM; i.e., 4096-530)

BASIC will display:

∧ AIM 65 BASIC Vn.n

where n.n is the version number.

BASIC is now in the command entry mode as indicated by the BASIC prompt (∧) in the display column 1. Subject 201 gets you started into the BASIC commands.

Read the following paragraphs first, however, to understand how to exit and reenter the BASIC and how the BASIC cursor prompt operates.

CAUTION

Entering BASIC with the 5 key causes the allocated memory to be initialized with AA (hex) in all bytes, starting with address 532. This, of course, destroys any previous BASIC programs, data in the AIM 65 Editor Text Buffer, or machine level routines that may have been stored in this portion of memory. Be sure to save any desired data or programs that may exist in this area before entering BASIC with the 5 key.

Note that text in the Text Buffer or machine level routine may co-exist in memory with BASIC by locating such text or routines in upper memory and entering the highest BASIC address with a value lower than the starting address of such text or routines.

EXITING BASIC

To escape from BASIC and return to the AIM 65 Monitor, type ESC any time the BASIC command cursor is displayed. You can also escape BASIC while a program is running, by pressing the F1 key (see Subject 301).

Pressing RESET will also cause the AIM 65 Monitor to be entered as well as performing a hardware reset of AIM 65.

REENTERING BASIC

BASIC may be reentered by typing 6 whenever the AIM 65 Monitor prompt is displayed. In this case, however, any existing BASIC program is retained in memory. AIM 65 will respond to a Key 6 entry with:

< 6 >

∧ 6 >

BASIC CURSOR

The BASIC cursor (∧), displayed in column 1 whenever BASIC is in the command entry mode, indicates that a BASIC command can be entered. The last displayed data resulting from the previous command is retained except for column 1 to provide information continuity with the previous command or displayed output data. This is especially helpful when the printer control is turned off to preserve printer paper.

When the first character of the next command is typed, the display will blank except for the newly typed character. The cursor then advances across the display in accordance with typed characters to indicate the character input position.

The displayed cursor does not appear on the printer output, thus any data printed in column 1 will be retained.

CAUTION

The minus sign associated with any negative values that are displayed starting in column 1 will be replaced with the cursor in the BASIC command entry mode. In the case of direct commands, the minus sign will only flash before the cursor is displayed if the printer control is on or may not appear at all if the printer control is off. In order to retain the minus sign, a leading blank should be displayed before the value is displayed (see Subject 204).

PRINTER CONTROL

While in the BASIC command entry mode, the printer may be turned on or off by typing PRINT while CNTL is pressed (CNTL PRINT). The on/off state of the printer is displayed after typing PRINT.

If the printer is turned off, statements in the BASIC command entry mode and data output from PRINT commands will be directed to the display only. If the printer is turned on, all commands and data from PRINT commands will be directed to both the printer and display. With the printer off, data can still be directed to the printer by using the PRINT! command (see Subject 305).

Similarly, INPUT statements will output data to the printer in response to the printer control state. An INPUT! statement will output data to the printer even if the printer control is off (see Subject 305).

This section is not intended to be a detailed course in BASIC programming. It will, however, serve as an excellent introduction for those of you unfamiliar with the language.

We recommend that you try each example in this section as it is presented. This will enhance your "feel" for BASIC and how it is used. Table 201-1 lists all the AIM 65 BASIC commands.

NOTE

Any time the cursor (^) is displayed in column 1, a BASIC command may be typed in. End all commands to BASIC by typing RETURN. The RETURN tells BASIC that you have finished typing the command. If you make an error, type a DEL (RUBOUT on a TTY) to eliminate the last character. Repeated use of DEL will eliminate previous characters. An @ symbol will eliminate that entire line being typed.

Table 201-1. AIM 65 BASIC Commands

Commands	Input/Output
CLEAR	DATA
CONT	GET
FRE	INPUT
LIST	POS
LOAD	PRINT
NEW	READ
PEEK	SPC
POKE	TAB
RUN	
SAVE	**String Functions**
Program Statements	ASC
	CHR$
DEF FN	LEFT$
DIM	LEN
END	MID$
FOR	RIGHT$
GOSUB	STR$
GOTO	VAL
IF . . . GOTO	
IF . . . THEN	**Arithmetic Functions**
LET	
NEXT	ABS
ON . . . GOSUB	ATN*
ON . . . GOTO	COS
REM	EXP
RESTORE	INT
RETURN	LOG
STOP	RND
USR	SIN
WAIT	SGN
	SQR
	TAN

*Although the ATN function is not included in AIM 65 BASIC, the ATN command is recognized (see Appendix H).

DIRECT COMMANDS

Try typing in the following:

PRINT 10-4 (end with RETURN)

BASIC will immediately print:

6

The print statement you typed in was executed as soon as you hit the RETURN key. This is called a *direct* command. BASIC evaluated the formula after the "PRINT" and then typed out its value, in this case "6".

Now try typing in this:

PRINT 1/2,3*10 *("*" means multiply, "/" means divide)*

BASIC will print:

.5 30

As you can see, BASIC can do division and multiplication as well as subtraction. Note how a "," (comma) was used in the print command to print two values instead of just one. The command divides a line into 10-character-wide columns. The comma causes BASIC to skip to the next 10-column field on the terminal, where the value 30 is printed.

INDIRECT COMMANDS

There is another type of command called an Indirect Command. Every Indirect command begins with a Line Number. A Line Number is any integer from 0 to 63999.

Try typing in these lines:

```
10 PRINT 2+3
20 PRINT 2-3
```

A sequence of Indirect Commands is called a "Program." Instead of executing indirect statements immediately, BASIC saves Indirect Commands in memory. When you type in RUN, BASIC will execute the lowest numbered indirect statement that has been typed in first, then the next higher, etc., for as many as were typed in.

In the example above, we typed in line 10 first and line 20 second. However, it makes no difference in what order you type in indirect statements. BASIC always puts them into correct numerical order according to the Line Number.

Suppose we type in

```
RUN
```

BASIC will print:

```
 5
-1
```


In Subject 202, we typed a two-line program into memory. Now let's see how BASIC can be used to operate on either or both lines.

LISTING A PROGRAM

If we want a listing of the complete program currently in memory, we type in

```
LIST
```

BASIC will reply with:

```
10  PRINT 2+3
20  PRINT 2-3
```

DELETING A LINE

Sometimes it is desirable to delete a line of a program altogether. This is accomplished by typing the Line Number of the line to be deleted, followed by a carriage return.

Type in the following:

```
10
LIST
```

BASIC will reply with:

```
20  PRINT 2-3
```

We have now deleted line 10 from the program.

REPLACING A LINE

You can replace line 10, rather than just deleting it, by typing the new line 10 and hitting RETURN.

Type in the following:

```
10  PRINT 3-3
LIST
```

BASIC will reply with:

```
10 PRINT 3-3
20 PRINT 2-3
```

It is not recommended that lines be numbered consecutively. It may become necessary to insert a new line between two existing lines. An increment of 10 between line numbers is generally sufficient.

DELETING A PROGRAM

If you want to delete the complete program currently stored in memory, type in "NEW." If you are finished running one program and are about to read in a new one, be sure to type in "NEW" first.

Type in the following:

```
NEW
```

Now type in:

```
LIST
```


If is often desirable to include explanatory text along with answers that are printed out.

Type in the following:

```
PRINT "ONE HALF EQUALS", 1/2
```

BASIC will reply with:

```
ONE THIRD EQUALS
 .5
```

As explained in Subject 202, including a "," in a PRINT statement causes it to space over to the next 10-column field before the value following the "," is printed.

If we use a ";" instead of a comma, the next value will be printed immediately following the previous value.

NOTE

Numbers are always printed with at least one trailing space. Any text to be printed must always be enclosed in double quotes.

Try the following examples:

1.
```
PRINT "ONE HALF EQUALS"; 1/2
ONE HALF EQUALS .5
```

2.
```
PRINT 1, 2, 3
 1    2
 3
...
```

3.
```
PRINT 1; 2; 3
 1   2 3
```

4.
```
PRINT -1; 2; -3
-1   2-3
```

SECTION

GETTING STARTED WITH BASIC

SUBJECT

NUMBER FORMAT

We will digress for a moment to explain the format of numbers in BASIC. Numbers are stored internally to over nine digits of accuracy. When a number is printed, only nine digits are shown. Every number may also have an exponent (a power of ten scaling factor).

The largest number that may be presented in AIM 65 BASIC is $1.70141183*10^{38}$, while the smallest positive number is $2.93873588*10^{-39}$.

When a number is printed, the following rules define the format:

1. If the number is negative, a minus sign (-) is printed. If the number is positive, a space is printed.

2. If the absolute value of the number is an integer in the range 0 to 999999999, it is printed as an integer.

3. If the absolute value of the number is greater than or equal to 0.01 and less than or equal to 999999999, it is printed in fixed point notation, with no exponent.

4. If the number does not fall under categories 2 or 3, scientific notation is used.

Scientific notation is formatted as follows: SX.XXXXXXXXESTT. (Each X is some integer, 0 to 9.)

The leading "S" is the sign of the number: a space for a positive number and a "-" for for a negative one. One non-zero digit is printed before the decimal point. This is followed by the decimal point and then the other eight digits of the mantissa. An "E" is then printed (for exponent), followed by the sign (S) of the exponent; then the two digits (TT) of the exponent itself. Leading zeroes are never printed; i.e., the digit before the decimal is never zero. Trailing zeroes are never printed. If there is only one digit to print after all trailing zeroes are suppressed, no decimal point is printed. The exponent sign will be "+" for positive and "-" for negative. Two digits of the exponent are always printed; that is, zeroes are not suppressed in the exponent field. The value of any number expressed thus is the number to the left of the "E" times 10 raised to the power of the number to the right of the "E".

Regardless of what format is used, a space is always printed following a number. BASIC checks to see if the entire number will fit on the current line. If it cannot, a carriage return/line feed is executed before printing the number.

Following are examples of various numbers and the output format in which BASIC will output them:

NUMBER	OUTPUT FORMAT
+1	1
-1	-1
6523	6523
-23.460	-23.46
1E20	1E+20
-12.3456E-7	-1.23456E-06
1.234567E-10	1.23457E-10
1000000000	1E+09
999999999	999999999
.1	.1
.01	.01
.000123	1.23E-04

A number input from the keyboard or a numeric constant used in a BASIC program may have as many digits as desired, up to the maximum length of a line (72 characters) or maximum numeric value. However, only the first 10 digits are significant, and tenth digit is rounded up.

```
PRINT 1.23456789876543210
1.2345679
```


ASSIGNING VARIABLES WITH AN INPUT STATEMENT

Following is an example of a program that reads a value from the keyboard and uses that value to calculate and print a result:

```
10 INPUT R
20 PRINT 3.14159*R*R
RUN
? 10
 314.159
```

Here's what's happening: When BASIC encounters the input statement, it outputs a question mark (?) on the display and then waits for you to type in a number. When you do (in the above example, 10 was typed), execution continues with the next statement in the program after the variable (R) has been set (in this case to 10). In the above example, line 20 would now be executed. When the formula after the PRINT statement is evaluated, the value 10 is substituted for the variable R each time R appears in the formula. Therefore, the formula becomes 3.14159*10*10, or 314.159.

If we wanted to calculate the area of various circles, we could rerun the program for each successive circle. But, there's an easier way to do it simply by adding another line to the program, as follows:

```
30 GOTO 10
RUN
? 10
 314.159
? 3
 28.27431
? 4.7
 69.3977231
?
```

By putting a "GOTO" statement on the end of our program, we have caused it to go back to line 10 after it prints each answer for the successive circles. This could have gone on indefinitely, but we decided to stop after calculating the area for three circles. This was accomplished by typing a carriage return to the input statement (thus a blank line).

VARIABLE NAMES

The letter "R" in the program above is a "variable." A variable name can be any alphabetic character and may be followed by any alphanumeric character (letters A to Z, numbers 0 to 9).

Any alphanumeric characters after the first two are ignored.

Here are some examples of legal and illegal variable names:

Legal	Illegal
A	% (first character must be alphabetic)
Z1	ZIABCD (variable name too long)
TP	TO (variable names cannot be reserved words)
PSTG$	RGOTO (variable names cannot contain reserved words)
COUNT	

ASSIGNING VARIABLES WITH A LET OR ASSIGNMENT STATEMENT

Besides having values assigned to variables with an input statement, you can also set the value of a variable with a LET or assignment statement.

Try the following examples:

```
A=5

PRINT A, A*2
 5    10

LET Z=7

PRINT Z, Z-A
 7    2
```

As you will notice from the examples, the "LET" is optional in an assignment statement.

BASIC "remembers" the values that have been assigned to variables using this type of statement. This "remembering" process uses space in the memory to store the data.

The values of variables are discarded (and the space in memory used to store them is released) when one of four conditions occur:

- A new line is typed into the program or an old line is deleted
- A CLEAR command is typed in
- A RUN command is typed in
- NEW is typed in

Another important fact is that if a variable is encountered in a formula before it is assigned a value, it is automatically assigned the value zero. Zero is then substituted as the value of the variable in the particular formula. Try the example below:

```
PRINT Q; Q + 2; Q*2
 0   2   0
```

RESERVED WORDS

The words used as BASIC statements are "reserved" for this specific purpose. You cannot use these words as variable names or inside of any variable name. For instance, "FEND" would be illegal because "END" is a reserved word.

Table 206-1 is a list of the reserved words in BASIC.

Table 206-1. AIM 65 BASIC Reserved Words

ABS	FN	LIST	PRINT	SPC
AND	FOR	LOAD	POS	SQR
ASC	FRE	LOG	READ	STEP
ATN	GET	MID$	REM	STOP
CHR$	GOSUB	NEW	RESTORE	STR$
CLEAR	GOTO	NEXT	RETURN	TAB
CONT	IF	NOT	RIGHT$	TAN
COS	INPUT	NULL	RND	THEN
DATA	INT	ON	RUN	TO
DEF	LEFT$	OR	SAVE	USR
DIM	LEN	PEEK	SGN	VAL
END	LET	POKE	SIN	WAIT
EXP				

REMARKS

The REM (short for "remark") statement is used to insert comments or notes into a program. When BASIC encounters a REM statement, the rest of the line is ignored.

This serves mainly as an aid for the programmer, and serves no useful function as far as the operation of the program in solving a particular problem.

Suppose we wanted to write a program to check whether a number is zero. With the statements we've gone over so far, this could not be done. What is needed is a statement which can be used to conditionally branch to another statement. The "IF-THEN" statement does just that.

Type in the following program: (remember, type NEW first)

```
10 INPUT B
20 IF B=0 THEN 50
30 PRINT "NON-ZERO"
40 GOTO 10
50 PRINT "ZERO"
60 GOTO 10
```

When this program is typed and run, it will ask for a value for B. Type in any value you wish. The AIM 65 will then come to the "IF" statement. Between the "IF" and the "THEN" portion of the statement there are two expressions separated by a "relation."

A relation is one of the following six symbols:

RELATION	MEANING
=	EQUAL TO
>	GREATER THAN
<	LESS THAN
< >	NOT EQUAL TO
< = or = <	LESS THAN OR EQUAL TO
= > or > =	GREATER THAN OR EQUAL TO

The IF statement is either true or false, depending upon whether the two expressions satisfy the relation. For example, in the program we just did, if 0 was typed in for B the IF statement would be true because 0=0. In this case, since the number after the THEN is 50, execution of the program would continue at line 50. Therefore, "ZERO" would be printed and then the program would jump back to line 10 (because of the GOTO statement in line 60).

Suppose a 1 was typed in for B. Since 1=0 is false, the IF statement would be false and the program would continue execution with the next line. Therefore, "NON-ZERO" would be printed and the GOTO in line 40 would send the program back to line 10.

A PROGRAM USING RELATIONS

Now try the following program for comparing two numbers:

```
10  INPUT A, B
20  IF A <= B THEN 50
30  PRINT "A IS BIGGER"
40  GOTO 10
50  IF A < B THEN 80
60  PRINT "THEY ARE THE SAME"
70  GOTO 10
80  PRINT "B IS BIGGER"
90  GOTO 10
```

When this program is run, line 10 will input two numbers from the keyboard. At line 20, if A is greater than B, A <= B will be false. This will cause the next statement to be executed, printing "A IS BIGGER" and then line 40 sends the computer back to line 10 to begin again.

At line 20, if A has the same value as B, A <= B is true so we go to line 50. At line 50, since A has the same value as B, A < B is false; therefore, we go to the following statement and print "THEY ARE THE SAME." Then line 70 sends us back to the beginning again.

At line 20, if A is smaller than B, A <= B is true so we go to line 50. At line 50, A < B will be true so we then go to line 80. "B IS BIGGER" is then printed and again we go back to the beginning.

Try running the last two programs several times. It may be easier to understand if you try writing your own program at this time using the IF-THEN statement. Actually trying programs of your own is the quickest and easiest way to understand how BASIC works. Remember, to stop these programs just give a RETURN to the input statement.

One advantage of computers is their ability to perform repetitive tasks. Let's take a closer look and see how this works.

A SQUARE ROOT PROGRAM

Suppose we want a table of square roots from 1 to 9. The BASIC function for square root is "SQR"; the form being SQR(X), X being the number whose square root is to be calculated. We could write the program as follows:

```
10 PRINT 1, SQR(1)
20 PRINT 2, SQR(2)
30 PRINT 3, SQR(3)
40 PRINT 4, SQR(4)
50 PRINT 5, SQR(5)
60 PRINT 6, SQR(6)
70 PRINT 7, SQR(7)
80 PRINT 8, SQR(8)
90 PRINT 9, SQR(9)
```

AN IMPROVED SQUARE ROOT PROGRAM

This program will do the job, but is terribly inefficient. We can improve the program considerably by using the IF statement just introduced as follows:

```
10  N=1
20  PRINT N; SQR(N)
30  N=N+1
40  IF N <= 9 THEN 20
```

When this program is run, its output will look exactly like that of the 9 statement program above it. Let's look at how it works:

At line 10 we have a LET statement which sets the value of the variable N equal to 1. At line 20 we print N and the square root of N using its current value. It thus becomes 20 PRINT 1; SQR(1), and this calculation is printed out.

At line 30 we use what will appear at first to be a rather unusual LET statement. Mathematically, the statement N=N+1 is nonsense. However, the important thing to remember is that in a LET statement, the symbol "=" does not signify equality. In this case, "=" means "to be replaced with." All the statement does is to take the current value of N and add 1 to it. Thus, after the first time through line 30, N becomes 2.

At line 40, since N now equals 2, N <= 9 is true so the THEN portion branches us back to line 20, with N now at a value of 2.

The overall result is that lines 20 through 40 are repeated, each time adding 1 to the value of N. When N finally equals 9 at line 20, the next line will increment it to 11. This results in a false statement at line 40, and since there are no further statements to the program it stops.

BASIC STATEMENTS FOR LOOPING

This technique is referred to as "looping" or "iteration." Since it is used quite extensively in programming, there are special BASIC statements for using it. We can show these with the following program:

```
10  FOR N=1 TO 9
20  PRINT N; SQR(N)
30  NEXT N
```

The output of the program listed above will be exactly the same as the previous two programs.

At line 10, N is set to equal 1. Line 20 causes the value of N and the square root of N to be printed. At line 30 we see a new type of statement. The "NEXT N" statement causes one to be added to N, and then if N <= 9 we go back to the statement following the "FOR" statement. The overall operation then is the same as with the previous program.

Notice that the variable following the "FOR" is exactly the same as the variable after the "NEXT." There is nothing special about the N in this case. Any variable could be used, as long as it is the same in both the "FOR" and the "NEXT" statements. For instance, "Z1" could be substituted everywhere there is an "N" in the above program and it would function exactly the same.

ANOTHER SQUARE ROOT PROGRAM

Suppose we want to print a table of square roots of each even number from 10 to 20. The following program performs this task:

```
10  N=10
20  PRINT N; SQR(N)
30  N=N+2
40  IF N <= 20 THEN 20
```


Note the similarity between this program and our "improved" square root program. This program can also be written using the "FOR" loop just introduced.

```
10  FOR N=10 TO 20 STEP 2
20  PRINT N; SQR(N)
30  NEXT N
```

Notice that the only major difference between this program and the previous one using "FOR" loops is the addition of the "STEP 2" clause.

This tells BASIC to add 2 to N each time, instead of 1 as in the previous program. If no "STEP" is given in a "FOR" statement, BASIC assumes that 1 is to be added each time. The "STEP" can be followed by any expression.

A COUNT-BACKWARD PROGRAM

Suppose we wanted to count backward from 10 to 1. A program for doing this would be as follows:

```
10  I=10
20  PRINT I
30  I=I-1
40  IF I >= 1 THEN 20
```

Notice that we are now checking to see that I is greater than or equal to the final value. The reason is that we are now counting by a negative number. In the previous examples it was the opposite, so we were checking for a variable less than or equal to the final value.

SOME OTHER LOOPING OPERATIONS

The "STEP" statement previously shown can also be used with negative numbers to accomplish this same result. This can be done using the same format as in the other program:

```
10  FOR I=10 TO 1 STEP -1
20  PRINT I
30  NEXT I
```

"FOR" loops can also be "nested." For example:

```
10  FOR I=1 TO 5
20  FOR J=1 TO 3
30  PRINT I, J
40  NEXT J
50  NEXT I
```

Notice that "NEXT J" precedes "NEXT I." This is because the J-loop is inside the I-loop. The following program is incorrect; run it and see what happens:

```
10  FOR I=1 TO 5
20  FOR J=1 TO 3
30  PRINT I, J
40  NEXT I
50  NEXT J
```

It does not work because when the "NEXT I" is encountered, all knowledge of the J-loop is lost. This happens because the J-loop is "inside" the I-loop.

It is often convenient to be able to select any element in a table of numbers. BASIC allows this to be done through the use of matrices.

A matrix is a table of numbers. The name of this table (the matrix name) is any legal variable name, "A" for example. The matrix name "A" is distinct and separate from the simple variable "A," and you could use both in the same program.

To select an element of the table, we subscript "A": that is, to select the I'th element, we enclose I in parentheses "(I)" and then follow "A" by this subscript. Therefore, "A(I)" is the I'th element in the matrix "A."

"A(I)" is only one element of matrix A, and BASIC must be told how much space to allocate for the entire matrix. This is done with a "DIM" statement, using the format "DIM A(15)." In this case, we have reserved space for the matrix index "I" to go from 0 to 15. Matrix subscripts always start at 0; therefore, in the above example, we have allowed for 16 numbers in matrix A.

If "A(I)" is used in a program before it has been dimensioned, BASIC reserves space for 11 elements (0 through 10).

A SORT PROGRAM

As an example of how matrices are used, try the following program to sort a list of 8 numbers, in which you pick the numbers to be sorted:

```
 10 DIM A(8)
 20 FOR I=1 TO 8
 30 INPUT A(I)
 50 NEXT I
 70 F=0
 80 FOR I=1 TO 7
 90 IF A(I) <= A(I+1) THEN 140
100 T=A(I)
110 A(I)=A(I+1)
120 A(I+1)=T
130 F=1
140 NEXT I
150 IF F=1 THEN 70
160 FOR I=1 TO 8
170 PRINT A(I)
180 NEXT I
```

When line 10 is executed, BASIC sets aside space for 9 numeric values, A(0) through A(8). Lines 20 through 50 get the unsorted list from the user. The sorting itself is done by going through the list of numbers and switching any two that are not in order. "F" is used to indicate if any switches were made; if any were made, line 150 tells BASIC to go back and check some more.

If we did not switch any numbers, or after they are all in order, lines 160 through 180 will print out the sorted list. Note that a subscript can be any expression.

SECTION

GETTING STARTED WITH BASIC

SUBJECT

SUBROUTINES

If you have a program that performs the same action in several different places, you could duplicate the same statements for the action in each place within the program.

The "GOSUB" and "RETURN" statements can be used to avoid this duplication. When a "GOSUB" is encountered, BASIC branches to the line whose number follows the "GOSUB." However, BASIC remembers where it was in the program before it branches. When the "RETURN" statement is encountered, BASIC goes back to the first statement following the last "GOSUB" that was executed. Observe the following program:

```
 10 PRINT "WHAT IS THE NUMBER";
 30 GOSUB 100
 40 T=N
 50 PRINT "SECOND NUMBER";
 70 GOSUB 100
 80 PRINT "THE SUM IS"; T+N
 90 STOP
100 INPUT N
110 IF N = INT(N) THEN 140
120 PRINT "MUST BE INTEGER."
130 GOTO 100
140 RETURN
```

This program asks for two numbers (which must be integers), and then prints their sum. The subroutine in this program is lines 100 to 140. The subroutine asks for a number, and if it is not an integer, asks for a new number. It will continue to ask until an integer value is typed in.

The main program prints "WHAT IS THE NUMBER," and then calls the subroutine to get the value of the number into N. When the subroutine returns (to line 40), the value input is saved in the variable T. This is done so that when the subroutine is called a second time, the value of the first number will not be lost.

"SECOND NUMBER" is then printed, and the second value is entered when the subroutine is again called.

When the subroutine returns the second time, "THE SUM IS" is printed, followed by the sum. T contains the value of the first number that was entered and N contains the value of the second number.

STOPPING A PROGRAM

The next statement in the program is a "STOP" statement. This causes the program to stop execution at line 90. If the "STOP" statement was excluded from the program, we would "fall into" the subroutine at line 100. This is undesirable because we would be asked to input another number. If we did, the subroutine would try to return; and since there was no "GOSUB" which called the subroutine, an RG error would occur. Each "GOSUB" executed in a program should have a matching "RETURN" executed later. The opposite also applies: a "RETURN" should be encountered only if it is part of a subroutine which has been called by a "GOSUB."

Either "STOP" or "END" can be used to separate a program from its subroutines. "STOP" will print a message saying at what line the "STOP" was encountered.

Suppose you had to enter numbers to your program that did not change each time the program was run, but you would like it to be easy to change them if necessary. BASIC contains special statements, "READ" and "DATA," for this purpose.

Consider the following program:

```
 10 PRINT "GUESS A NUMBER";
 20 INPUT G
 30 READ D
 40 IF D = -999999 THEN 90
 50 IF D <> G THEN 30
 60 PRINT "YOU ARE CORRECT"
 70 END
 90 PRINT "BAD GUESS, TRY AGAIN."
 95 RESTORE
100 GOTO 10
110 DATA 1, 393, -39, 28, 391, -8, 0, 3.14, 90
120 DATA 89, 5, 10, 15, -34, -999999
```

When the "READ" statement is encountered, the effect is the same as an INPUT statement. But, instead of getting a number from the keyboard, a number is read from the "DATA" statements.

The first time a number is needed for a READ, the first number in the first DATA statement is read. The second time one is needed, the second number in the first DATA statement is read. When the all numbers of the first DATA statement have been read in this manner, the second DATA statement will be used. DATA is always read sequentially in this manner, and there may be any number of DATA statements in your program.

The purpose of this program is to play a little game in which you try to guess one of the numbers contained in the DATA statements. For each guess that is typed in, we read through all of the numbers in the DATA statements until we find one that matches the guess.

If more values are read than there are numbers in the DATA statements, an out of data (OD) error occurs. That is why in line 40 we check to see if -999999 was read. This is not one of the numbers to be matched, but is used as a flag to indicate that all of the data (possible correct guesses) has been read. Therefore, if -999999 was read, we know that the guess was incorrect.

Before going back to line 10 for another guess, we need to make the READ's begin with the first piece of data again. This is the function of the "RESTORE." After the RESTORE is encountered, the next piece of data read will be the first number in the first DATA statement again.

DATA statements may be placed anywhere within the program. Only READ statements make use of the DATA statements in a program, and any other time they are encountered during program execution they will be ignored.

A list of characters is referred to as a "String." Rockwell, R6500, and THIS IS A TEST are all strings. Like numeric variables, string variables can be assigned specific values. String variables are distinguished from numeric variables by a "$" after the variable name.

For example, try the following:

```
A$="ROCKWELL R6500"
PRINT A$
ROCKWELL R6500
```

In this example, we set the string variable A$ to the string value "ROCKWELL R6500." Note that we also enclosed the character string to be assigned to A$ in quotes.

LEN FUNCTION

Now that we have set A$ to a string value, we can find out what the length of this value is (the number of characters it contains). We do this as follows:

```
PRINT LEN(A$), LEN("MICROCOMPUTER")
 14            13
```

The "LEN" function returns an integer equal to the number of characters in a string.

A string expression may contain from 0 to 255 characters. A string containing 0 characters is called the "null" string. Before a string variable is set to a value in the program, it is initialized to the null string. Printing a null string on the terminal will cause no characters to be printed, and the printer or cursor will not be advanced to the next column. Try the following:

```
PRINT LEN(Q$); Q$; 3
 0  3
```

Another way to create the null string is: Q$=''''

Setting a string variable to the null string can be used to free up the string space used by a non-null string variable.

LEFT$ FUNCTION

It is often desirable to access parts of a string and manipulate them. Now that we have set A$ to "ROCKWELL R6500," we might want to print out only the first eight characters of A$. We would do so like this:

```
PRINT LEFT$ (A$, 8)
ROCKWELL
```

"LEFT$" is a string function which returns a string composed of the leftmost N characters of its string argument. Here is another example:

```
FOR N=1 TO LEN(A$):PRINT LEFT$(A$, N):NEXT N
R
RO
ROC
ROCK
ROCKW
ROCKWE
ROCKWEL
ROCKWELL
ROCKWELL R
ROCKWELL R6
ROCKWELL R65
ROCKWELL R650
ROCKWELL R6500
```

Since A$ has 14 characters, this loop will be executed with N=1, 2, 3, . . ., 13, 14. The first time through only the first character will be printed, the second time the first two characters will be printed, etc.

RIGHT$ FUNCTION

Another string function, called "RIGHT$," returns the right N characters from a string expression. Try substituting "RIGHT$" for "LEFT$" in the previous example and see what happens.

MID$ FUNCTION

There is also a string function which allows us to take characters from the middle of a string. Try the following:


```
FOR N=1 TO LEN(A$):PRINT MID$(A$, N):NEXT N
ROCKWELL R6500
OCKWELL R6500
CKWELL R6500
KWELL R6500
WELL R6500
ELL R6500
LL R6500
L R6500
 R6500
R6500
6500
500
00
0
```

"MID$" returns a string starting at the Nth position of A$ to the end (last character) of A$. The first position of the string is position 1 and the last possible position of a string is position 255.

Very often it is desirable to extract only the Nth character from a string. This can be done by calling MID$ with three arguments. The third argument specifies the number of characters to return.

For example:

```
FOR N=1 TO LEN(A$):PRINT MID$(A$, N, 1), MID$(A$, N, 2):NEXT N
R             RO
O             OC
C             CK
K             KW
W             WE
E             EL
L             LL
L             L
               R
R             R6
6             65
5             50
0             00
0             0
```

CONCATENATION–JOINING STRINGS

Strings may also be concatenated (put or joined together) through the use of the "+" operator. Try the following:

```
B$="BASIC FOR"+" "+A$
PRINT B$
BASIC FOR ROCKWELL R6500
```

Concatenation is especially useful if you wish to take a string apart and then put it back together with slight modifications. For instance:

```
C$=LEFT$(B$, 9)+"-"+MID$(B$, 11, 8)+"-"+RIGHT$(B$, 5)
PRINT C$
BASIC FOR-ROCKWELL-R6500
```

VAL AND STR$ FUNCTIONS

Sometimes it is desirable to convert a number to its string representation, and vice-versa. "VAL" and "STR$" perform these functions.

Try the following:

```
STRING$="567.8"
PRINT VAL(STRING$
 567.8
STRING$=STR$(3.1415)
PRINT STRING$, LEFT$(STRING$, 5)
 3.1415        3.14
```

"STR$" can be used to perform formatted I/O on numbers. You can convert a number to a string and then use LEFT$, RIGHT$, MID$ and concatenation to reformat the number as desired.

"STR$" can also be used to conveniently find out how many print columns a number will take. For example:

```
PRINT LEN(STR$(3.157))
 6
```

If you have an application in which a user is typing in a question such as "WHAT IS THE VOLUME OF A CYLINDER OF RADIUS 5.36 FEET, OF HEIGHT 5.1 FEET?" you can use "VAL" to extract the numeric values 5.36 and 5.1 from the question.

CHR$ FUNCTION

CHR$ is a string function which returns a one character string which contains the alphanumeric equivalent of the argument, according to the conversion table in Appendix E. ASC takes the first character of a string and converts it to its ASCII decimal value.

One of the most common uses of CHR$ is to send a special character to a terminal.

```
100 DIM A$(15)
110 FOR I=1 TO 15
112 READ A$(I)
114 NEXT I
120 F=0:  I=1
130 IF A$(I) < =A$(I+1) THEN 180
140 T$=A$(I+1)
150 A$(I+1)=A$(I)
160 A$(I)=T$
170 F=1
180 I=I+1
185 IF I < 15 THEN 130
190 IF F THEN 120
200 FOR I=1 TO 15
202 PRINT A$(I)
204 NEXT I
220 DATA AIM 65, DOG
230 DATA CAT, R6500
240 DATA ROCKWELL, RANDOM
250 DATA SATURDAY, "***ANSWER***"
260 DATA MICRO, FOO
270 DATA COMPUTER, MED
280 DATA NEWPORT BE-ACH, DALLAS, ANAHEIM
```

ADDITIONAL STRING CONSIDERATIONS

1. A string may contain from 0 to 255 characters. All string variable names end in a dollar sign ($); for example, A$, B9$, K$, HELLO$.

2. String matrices may be dimensioned exactly like numeric matrices. For instance, DIM A$(10, 10) creates a string matrix of 121 elements, eleven rows by elevon columns (rows 0 to 10 and columns 0 to 10). Each string matrix element is a complete string, which can be up to 255 characters in length.

NAME	EXAMPLE	PURPOSE/USE
DIM	25 DIM A$(10, 10)	Allocates space for a pointer and length for each element of a string matrix. No string space is allocated.
LET	27 LET A$="F00"+V$	Assigns the value of a string expression to a string variable. LET is optional.
= > < <= or =< >= or => < >		String comparison operators. Comparison is made on the basis of ASCII codes, a character at a time until a difference is found. If during the comparison of two strings, the end of one is reached, the shorter string is considered smaller. Note that "A" is greater than "A" since trailing spaces are significant.
+	30 LET Z$=R$+Q$	String concatenation. The resulting string must be less than 256 characters in length or an LS error will occur.
INPUT	40 INPUT X$	Reads a string from the keyboard. String does not have to be quoted; but if not, leading blanks will be ignored and the string will be terminated on a "," or ":" character.
READ	50 READ X$	Reads a string from DATA statements within the program. Strings do not have to be quoted; but if they are not, they are terminated on a "," or ":" character and leading spaces are ignored. See DATA for the format of string data.
PRINT	60 PRINT X$ 70 PRINT "F00"+A$	Prints the string expression on the display/printer.

CHARACTER	USE
@	Erases current line being typed, and types a carriage return/line feed.
DEL	Erases last character typed. If no more characters are left on the line, types a carriage return/line feed.
RETURN	A RETURN must end every line typed in. Returns cursor to the first position (leftmost) on line, and prints the line if the printer is on.
F1	Interrupts execution of a program or a list command. F1 has effect when a statement finishes execution, or in the case of interrupting a LIST command, when a complete line has finished printing. In both cases a return is made to BASIC's command level and OK is typed. Prints "BREAK IN LINE XXXX," where XXXX is the line number of the next statement to be executed. There is no F1 key on a TTY. However, when TTY is being used, the AIM 65's F1 key is operational and can be used.
: (colon)	A colon is used to separate statements on a line. Colons may be used in direct and indirect statements. The only limit on the number of statements per line is the line length. It is not possible to GOTO or GOSUB to the middle of a line.
?	Question marks are equivalent to PRINT. For instance, ? 2+2 is equivalent to PRINT 2+2. Question marks can also be used in indirect statements. 10 ? X, when listed, will be typed as 10 PRINT X.
$	A dollar sign ($) suffix on a variable name establishes the variable as a character string.

CHARACTER	USE
%	A percent sign (%) suffix on a variable name establishes the variable as an integer
!	An exclamation sign (!) suffix on an INPUT, PRINT, or ? command causes the input or output to be printed even though the printer is turned off.
ESC	Returns control to the Monitor.
CNTL PRINT	Turns the AIM 65 printer on if it is off, and off if it is on.

SYMBOL	SAMPLE STATEMENT	PURPOSE/USE
=	A=100	Assigns a value to a variable
	LET Z=2.5	The LET is optional
-	B=-A	Negation. Note that 0-A is subtraction, while -A is negation.
^ (F3 key)	130 PRINT X^3	Exponentiation (equal to X*X*X in the sample statement)
		0^0=1 0 to any other power = 0
		A^B, with A negative and B not an integer gives an FC error.
*	140 X=R*(B*D)	Multiplication.
/	150 PRINT X/1.3	Division.
+	160 Z=R+T÷Q	Addition
-	170 J=100-I	Subtraction

RULES FOR EVALUATING EXPRESSIONS:

1) Operations of higher precedence are performed before operations of lower precedence. This means the multiplication and divisions are performed before additions and subtractions. As an example, 2+10/5 equals 4, not 2.4. When operations of equal precedence are found in a formula, the left hand one is executed first: 6-3+5=8, not -2.

2) The order in which operations are performed can always be specified explicitly through the use of parentheses. For instance, to add 5 to 3 and then divided that by 4, we would use (5+3)/4, which equals 2. If instead we had used 5+3/4, we would get 5.75 as a result (5 plus 3/4).

The precedence of operators used in evaluating expressions is as follows, in order beginning with the highest precedence:

NOTE

Operators listed on the same line have the same precedence.

1) Expressions in parentheses are always evaluated first

2) ^ (F3 KEY) — Exponentiation

3) NEGATION — -X where X may be a formula

4) * and / — Multiplication and Division

5) + and - — Addition and Subtraction

6) RELATIONAL OPERATORS: *(equal precedence for all six)*

=	Equal
<>	Not Equal
<	Less Than
>	Greater Than
=< or <=	Less Than or Equal
=> or >=	Greater Than or Equal

(These three below are Logical Operators)

7) NOT — Logical and bitwise "NOT" like negation, not takes only the formula to its right as an argument

8) AND — Logical and bitwise "AND"

9) OR — Logical and bitwise "OR"

A relational expression can be used as part of any expression.

Relational Operator expressions will always have a value of True (-1) or a value of False (0). Therefore, (5 = 4)=0, (5=5)=-1, (4>5)=0, (4<5)=-1, etc.

The THEN clause of an IF statement is executed whenever the formula after the IF is not equal to 0. That is to say, IF X THEN . . . is equivalent to IF X<>0 THEN . . . .

SYMBOL	SAMPLE STATEMENT	PURPOSE/USE
=	10 IF A=15 THEN 40	Expression Equals Expression
<>	70 IF A<>0 THEN 5	Expression Does Not Equal Expression
>	30 IF B>100 THEN 8	Expression Greater Than Expression
<	160 IF B<2 THEN 10	Expression Less Than Expression
<=,=<	180 IF 100<=B+C THEN 10	Expression Less Than or Equal To Expression
>=,=>	190 IF Q=>R THEN 50	Expression Greater Than Or Equal To Expression
AND	2 IF A<5 AND B<2 THEN 7	If expression 1 (A<5) AND expression 2 (B<2) are both true, then branch to line 7
OR	IF A<1 OR B<2 THEN 2	If either expression 1 (A<1) OR expression 2 (B<2) is true, then branch to line 2
NOT	IF NOT Q3 THEN 4	If expression "NOT Q3" is true (Because Q3 is false), then branch to line 4
		Note: *NOT -1=0 (NOT true=false)*

AND, OR, and NOT can be used for bit manipulation, and for performing boolean operations.

These three operators convert their arguments to sixteen bit, signed two's-complement integers in the range -32768 to +32767. They then perform the specified logical operation on them and return a result within the same range. If the arguments are not in this range, an "FC" error results.

The operations are performed in bitwise fashion, this means that each bit of the result is obtained by examining the bit in the same position for each argument.

The following truth table shows the logical relationship between bits:

OPERATOR	ARGUMENT 1	ARGUMENT 2	RESULT
AND	1	1	1
	0	1	0
	1	0	0
	0	0	0
OR	1	1	1
	1	0	1
	0	1	1
	0	0	0
NOT	1	–	0
	0	–	1

EXAMPLES: *(In all of the examples below, leading zeroes on binary numbers are not shown.)*

63 AND 16=16 — Since 63 equals binary 111111 and 16 equals binary 10000, the result of the AND is binary 10000 or 16.

15 AND 14=14 — 15 equals binary 1111 and 14 equals binary 1110, so 15 AND 14 equals binary 1110 or 14.

-1 AND 8=8 — -1 equals binary 1111111111111111 and 8 equals binary 1000, so the result is binary 1000 or 8 decimal.

4 AND 2=0 — 4 equals binary 100 and 2 equals binary 10, so the result is binary 0 because none of the bits in either argument match to give a 1 bit in the result.

4 OR 2=6 — Binary 100 OR'd with binary 10 equals binary 110, or 6 decimal.

10 OR 10=10 — Binary 1010 OR'd with binary 1010 equals binary 1010, or 10 decimal.

-1 OR -2=-1 — Binary 1111111111111111 (-1) OR'd with binary 1111111111111110 (-2) equals binary 1111111111111111, or -1.

NOT 0=-1 — The bit complement of binary 0 to 16 places is sixteen ones (1111111111111111) or -1. Also NOT -1=0.

NOT X — NOT X is equal to -(X+1). This is because to form the sixteen bit two's complement of the number, you take the bit (one's) complement and add one.

NOT 1=-2 — The sixteen bit complement of 1 is 1111111111111110, which is equal to -(1+1) or -2.

A typical use of the bitwise operators is to test bits set in the computer's locations which reflect the state of some external device.

Bit position 7 is the most significant bit of a byte, while position 0 is the least significant.

For instance, suppose bit 1 of location 40963 is 0 when the door to Room X is closed, and 1 if the door is open. The following program will print "Intruder Alert" if the door is opened:

```
10 IF NOT (PEEK(40963) AND 2) THEN 10
```

This line will execute over and over until bit 1 (masked or selected by the 2) becomes a 1. When that happens, we go to line 20.

```
20 PRINT "INTRUDER ALERT"
```

Line 20 will output "INTRUDER ALERT."

However, we can replace statement 10 with a "WAIT" statement, which has exactly the same effect.

```
10 WAIT 40963, 2
```

This line delays the execution of the next statement in the program until bit 1 of location A003 becomes 1. The WAIT is much faster than the equivalent IF statement and also takes less bytes of program storage.

The following is another useful way of using relational operators:

```
125 A=-(B>C)*B-(B<=C)*C
```

This statement will set the variable A to MAX(B,C) = the larger of the two variables B and C.

A BASIC command may be entered when the cursor is displayed. This is called the "Command Level." Commands may be used as program statements. Certain commands, such as LIST, NEW, and LOAD will terminate program execution when they finish. Each command may require one or more arguments in addition to the command statement, as defined in the syntax/function description. An argument without parenthesis is required to be entered without parenthesis. Arguments contained within parenthesis are required to be entered with the shown parenthesis. Arguments within brackets are optional. Optional arguments, if included, must be entered with or without accompanying parenthesis, however shown.

STATEMENT	SYNTAX/FUNCTION	EXAMPLE
CLEAR	CLEAR Clears all program variables, resets "FOR" and "GOSUB" state, and restores data.	CLEAR
CONT	CONT Continues program execution after the F1 key or a STOP or INPUT statement terminates execution. You cannot continue after any error, after modifying your program, or before your program has been run. One of the main purposes of CONT is debugging. Suppose at some point after running your program, nothing is printed. This may be because your program is performing some time consuming calculation, but it may be because you have fallen into an "infinite loop." An infinite loop is a series of BASIC statements from which there is no excape. BASIC will keep executing the series of statements over and over; until you intervene or until power to the AIM 65 is turned off. If you suspect your program is in an infinite loop, press F1 until the BREAK message is displayed. The line number of the statement BASIC was executing will be displayed. After BASIC has displayed the cursor, you can use PRINT to type out some of the values of your variables. After examining these values you may become satisfied that your program is functioning correctly. You	CONT