AIM 65

MICROCOMPUTER

BASIC LANGUAGE REFERENCE MANUAL

Rockwell International

$5.00
Document No 29650 N49
March 1979

• ROCKWELL INTERNATIONAL CORPORATION

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).

SECTION

GETTING STARTED WITH BASIC

SUBJECT

BASIC COMMAND SET

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

CLEAR
CONT
FRE
LIST
LOAD
NEW
PEEK
POKE
RUN
SAVE

Program Statements

DEF FN
DIM
END
FOR
GOSUB
GOTO
IF . . . GOTO
IF . . . THEN
LET
NEXT
ON . . . GOSUB
ON . . . GOTO
REM
RESTORE
RETURN
STOP
USR
WAIT

Input/Output

DATA
GET
INPUT
POS
PRINT
READ
SPC
TAB

String Functions

ASC
CHR$
LEFT$
LEN
MID$
RIGHT$
STR$
VAL

Arithmetic Functions

ABS
ATN*
COS
EXP
INT
LOG
RND
SIN
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.

SECTION

GETTING STARTED WITH BASIC

SUBJECT

NUMBER FORMAT

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.

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 1
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.

SECTION

STATEMENT DEFINITIONS

SUBJECT

SPECIAL CHARACTERS

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 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

STATEMENT	SYNTAX/FUNCTION	EXAMPLE
	should then type in CONT to continue executing your program where it left off, or type a direct GOTO statement to resume execution of the program at a different line. You could also use assignment statements to set some of your variables to different values. Remember, if you interrupt a program with the F1 key and expect to continue it later, you must not get any errors or type in any new program lines. If you do, you won't be able to continue and will get a "CN" (continue not) error. It is impossible to continue a direct command. CONT always resumes execution at the next statement to be executed in your program when F1 was typed.	
FRE	FRE (expression) Gives the number of memory bytes currently unused by BASIC. A dummy operand—0 or 1—must be used.	270 PRINT FRE(0)
LIST	LIST [[start line] [-[end line]]] Lists current program optionally starting at specified line. List can be interrupted with the F1 key. (BASIC will finish listing the current line.)	
	Lists entire program	LIST
	Lists just line 100.	LIST 100
	Lists lines 100 to 1000.	LIST 100-1000
	Lists from current line to line 1000.	LIST -1000
	Lists from line 100 to end of program.	LIST 100-
LOAD	LOAD Loads a BASIC program from the cassette tape. When done, the LOAD will display the cursor. See Appendix G for more information.	LOAD

STATEMENT	SYNTAX/FUNCTION	EXAMPLE
NEW	NEW Deletes current program and all variables.	NEW
PEEK	PEEK (address) The PEEK function returns the contents of memory address I in decimal. The value returned will be =>0 and <=255. If I is >65535 or <0, an FC error will occur. An attempt to read a non-existent memory address will return an unknown value.	356 PRINT PEEK (I)
POKE	POKE location, byte The POKE statement stores the byte specified by its second argument (J) into the location given by its first argument (I). The byte to be stored must be =>0 and <=255, or an FC error will occur. The address (I) must be =>0 and <=65535, or an FC error result. **Caution:** *Careless use of the POKE statement may cause your program, BASIC, or the Monitor functions to operate incorrectly, to hang up, and/or cause loss of your program. Note that Pages 0 and 1 in memory are reserved for use by BASIC and should not be used for user program variable storage.* A POKE to a non-existent memory location is harmless. One of the main uses of POKE is to pass arguments to machine language subroutines. (See Appendix F.) You could also use PEEK and POKE to write a memory diagnostic or an assembler in BASIC.	357 POKE I, J

STATEMENT	SYNTAX/FUNCTION	EXAMPLE
RUN	RUN line number Starts execution of the program currently in memory at the specified line number. RUN deletes all variables (does a CLEAR) and restores DATA. If you have stopped your program and wish to continue execution at some point in the program, use a direct GOTO statement to start execution of your program at the desired line, or CONT to continue after a break.	RUN 200
	Start program execution at the lowest numbered statement.	RUN
SAVE	SAVE Saves the current program in the AIM 65 memory on cassette tape. The program in memory is left unchanged. More than one program may be stored on cassette using this command. See Appendix G for more information.	SAVE

In the following description of statements, an argument of B, C, V or W denotes a numeric variable, X denotes a numeric expression, X$ denotes a string expression and an I or J denotes an expression that is truncated to an integer before the statement is executed. Truncation means that any fractional part of the number is lost, e.g., 3.9 becomes 3, 4.01 becomes 4.

An expression is a series of variables, operators, function calls and constants which after the operations and function calls are performed using the precedence rules, evaluates to a numeric or string value.

A constant is either a number (3.14) or a string literal ("F00").

STATEMENT	SYNTAX/FUNCTION	EXAMPLE
DEF	DEF FNx [(argument list)] = expression The user can define functions like the built-in functions (SQR, SGN, ABS, etc.) through the use of the DEF statement. The name of the function is "FN" followed by any legal variable name, for example: FNX, FNJ7, FNK0, FNR2. User defined functions are restricted to one line. A function may be defined to be any expression, but may only have one argument. In the example, B and C are variables that are used in the program. Executing the DEF statement defines the function. User defined functions can be redefined by executing another DEF statement for the same function. "V" is called the dummy variable.	100 DEF FNA(V)=V/B+C
	Execution of this statement following the above would cause Z to be set to 3/B+C, but the value of V would be unchanged.	100 Z=FNA(3)
DIM	DIM variable (size 1, [size 2 . . .]) Allocates space for matrices. All matrix elements are set to zero by the DIM statement.	113 DIM A(3), B(10)
	Matrices can have from one to 255 dimensions.	114 DIM R3(5,5), D$(2,2,2)

STATEMENT	SYNTAX/FUNCTION	EXAMPLE
	Matrices can be dimensioned dynamically during program execution. If a matrix is not explicitly dimensioned with a DIM statement, it is assumed to be a single dimensioned matrix of whose single subscript may range 0 to 10 (eleven elements).	115 DIM Q1(N), Z(2*I)
	If this statement was encountered before a DIM statement for A was found in the program, it would be as if a DIM A(10) had been executed previous to the execution of line 117. All subscripts start at zero (0), which means that DIM X(100) really allocates 101 matrix elements.	117 A(8)=4
END	END Terminates program execution without printing a BREAK message. (See STOP.) CONT after an END statement causes execution to resume at the statement after the END statement. END can be used anywhere in the program, and is optional.	999 END
FOR	FOR variable = expression to expression [STEP expression] (See NEXT statement) V is set equal to the value of the expression following the equal sign, in this case 1. This value is called the initial value. Then the statements between FOR and NEXT are executed. The final value is the value of the expression following the TO. The step is the value of the expression following STEP. When the NEXT statement is encountered, the step is added to the variable.	300 FOR V=1 TO 9.3 STEP .6

STATEMENT	SYNTAX/FUNCTION	EXAMPLE
	If no STEP was specified, it is assumed to be one. If the step is positive and the new value of the variable is <= the final value (9.3 in this example), or the step value is negative and the new value of the variable is => the final value, then the first statement following the FOR statement is executed. Otherwise, the statement following the NEXT statement is executed. All FOR loops execute the statements between the FOR and the NEXT at least once, even in cases like FOR V=1 TO 0.	310 FOR V=1 TO 9.3
	Note that expressions (formulas) may be used for the initial, final and step values in a FOR loop. The values of the expressions are computed only once, before the body of the FOR . . . NEXT loop is executed.	315 FOR V=10*N TO 3.4/Q STEP SQR(R)
	When the statement after the NEXT is executed, the loop variable is never equal to the final value, but is equal to whatever value caused the FOR . . . NEXT loop to terminate. The statements between the FOR and its corresponding NEXT in both examples above (310 and 320) would be executed *nine* times.	320 FOR V=9 TO 1 STEP -1
	Error: do not use nested FOR . . . NEXT loops with the same index variable. FOR loop nesting is limited only by the available memory. (See Appendix C.)	330 FOR W=1 TO 10: FOR W=1 TO 5:NEXT W:NEXT W
GOSUB	GOSUB line number Branches to the specified statement (910) until a RETURN is encountered; when a branch is then made to the statement after the GOSUB. GOSUB nesting is limited only by the available memory.	10 GOSUB 910

STATEMENT	SYNTAX/FUNCTION	EXAMPLE
GOTO	GOTO line number Branches to the statement specified.	50 GOTO 100
IF . . . GOTO	IF expression GOTO line number . . . Equivalent to IF . . . THEN, except that IF . . . GOTO must be followed by a line number, while IF . . . THEN can be followed by either a line number or another statement.	32 IF X<=Y+23.4 GOTO 92
IF . . . THEN	IF expression THEN line number . . . Branches to specified statement if the relation is True.	IF X<10 THEN 5
	Executes all of the statements on the remainder of the THEN if the relation is True.	20 IF X<0 THEN PRINT "X LESS THAN 0"
	WARNING: *The "Z=A" will never be executed because if the relation is true, BASIC will branch to line 50. If the relation is false BASIC will proceed to to the line following line 25.*	*25 IF X=5 THEN 50:Z=A*
	In this example, if X is less than 0, the PRINT statement will be executed and then the GOTO statement will branch to line 350. If the X was 0 or positive, BASIC will proceed to execute the lines after line 26.	26 IF X<0 THEN PRINT "ERROR, X NEGATIVE": GOTO 350
LET	[LET] variable = expression Assigns a value to a variable.	300 LET W=X
	"LET" is optional.	310 V=5.1
NEXT	NEXT [variable] [, variable] . . . Marks the end of a FOR loop.	340 NEXT V

STATEMENT	SYNTAX/FUNCTION	EXAMPLE
	If no variable is given, matches the most recent FOR loop.	345 NEXT
	A single NEXT may be used to match multiple FOR statements. Equivalent to NEXT V:NEXT W.	350 NEXT V, W
ON . . . GOSUB	ON expression GOSUB line [, line] . . . Identical to "ON . . . GOTO," except that a subroutine call (GOSUB) is executed instead of a GOTO. RETURN from the GOSUB branches to the statement after the ON . . . GOSUB.	110 ON I GOSUB 50, 60
ON . . . GOTO	ON expression GOTO line [, line] . . . Branches to the line indicated by the I'th number after the GOTO. That is: IF I=1, THEN GOTO LINE 10 IF I=2, THEN GOTO LINE 20 IF I=3, THEN GOTO LINE 30 IF I=4, THEN GOTO LINE 40. If I=0, or I attempts to select a nonexistent line (>=5 in this case), the statement after the ON statement is executed. However, if I is >255 or <0, an FC error message will result. As many line numbers as will fit on a line can follow an ON . . . GOTO.	100 ON I GOTO 10, 20, 30, 40
	This statement will branch to line 40 if the expression X is less than zero, to line 50 if it equals zero, and to line 60 if it is greater than zero.	105 ON SGN(X)+2 GOTO 40, 50, 60

STATEMENT	SYNTAX/FUNCTION	EXAMPLE
REM	REM any text Allows the programmer to put comments in his program. REM statements are not executed, but can be branched to. A REM statement is terminated by end of line, but not by a ":".	500 REM NOW SET V=0
	In this case the V=0 will never be executed by BASIC.	505 REM SET V=0: V=0
	In this case V=0 will be executed.	505 V=0: REM SET V=0
RESTORE	RESTORE Allows the re-reading of DATA statements. After a RESTORE, the next piece of data read will be the first piece listed in the first DATA statement of the program. The second piece of data read will be the second piece listed in the first DATA statement, and so on as in a normal READ operation.	510 RESTORE
RETURN	RETURN Causes a subroutine to return to the statement after the most recently executed GOSUB.	50 RETURN
STOP	STOP Causes a program to stop execution and to enter command mode. Prints BREAK IN LINE 900. (As per this example.) CONT after a STOP branches to the statement following the STOP.	900 STOP
USR	USR (argument) Calls the user's machine language subroutine with the argument. See PEEK and POKE in Subject 303, and Appendix F.	200 V=USR(W)

SYMBOL	SYNTAX/FUNCTION	EXAMPLE
WAIT	WAIT (address, mask [, select]) This statement reads the contents of the addressed location, does an Exclusive–OR with the select value, and then ANDs the result with the mask. This sequence is repeated until a non-zero result is obtained, at which time execution continues at the statement that follows WAIT. If the WAIT statement has no select argument, the select value is assumed to be zero. If you are waiting for a bit to become zero, there should be a "one" in the corresponding bit position of the select value. The select value (K) and the mask value (J) can range from 0 to 255. The address (I) can range from 0 to 65535.	805 WAIT I, J, K 806 WAIT I, J

SECTION

STATEMENT DEFINITIONS

SUBJECT

INPUT/OUTPUT STATEMENTS

STATEMENT	SYNTAX/FUNCTION	EXAMPLE
DATA	DATA item [, item . . .] Specifies data, read from left to right. Information appears in data statements in the same order as it will be read in the program.	10 DATA 1, 3, -1E3, .04
	Strings may be read from DATA statements. If you want the string to contain leading spaces (blanks), colons (:) or commas (,), you must enclose the string in double quotes. It is illegal to have a double quote within string data or a string literal. (""BASIC"" is illegal.)	20 DATA "F00", Z1
INPUT	INPUT [!] ["prompt string literal";] variable [, variable] . . . Requests data from the keyboard (to be typed in). Each value must be separated from the preceeding value by a comma (,). The last value typed should be followed by a carriage return. A "?" is displayed as a prompt character. Only constants may be typed in as a response to an INPUT statement, such as 4.5E-3 or "CAT." If more data was requested in an INPUT statement than was typed in, a "??" is printed and the rest of the data should be typed in. If more data was typed in than was requested, the warning "EXTRA IGNORED" will be displayed. Strings must be input in the same format as they are specified in DATA statements.	3 INPUT V, W, W2
	Optionally displays a prompt string ("VALUE") before requesting data from the keyboard. If RETURN is typed to an input statement, BASIC returns to command mode. Typing CONT after an INPUT command has been interrupted will cause execution to resume at the INPUT statement.	5 INPUT "VALUE"; V

STATEMENT	SYNTAX/FUNCTION	EXAMPLE
	If the optional character ! is included following INPUT, then the prompts from the INPUT statement and the user's entries will be printed (even if the printer is turned off) and displayed.	15 INPUT! "VALUE"; V
POS	POS (expression) Gives the current position of the cursor on the display. The leftmost character position on the display is position zero. A dummy operand—0 or 1—must be used.	260 PRINT POS(1)
PRINT	PRINT [!] expression [, expression] Prints the value of expressions on the display/printer. If the list of values to be printed out does not end with a comma (,) or a semicolon (;), then a carriage return/line feed is executed after all the values have been printed. Strings enclosed in quotes (") may also be printed. If a semicolon separates two expressions in the list, their values are printed next to each other. If a comma appears after an expression in the list, and the print head is at print position 11 or more, then a carriage return/line feed is executed. If the print head is before print position 11, then spaces are printed until the carriage is at the beginning of the next 10 column field. If there is a blank string enclosed in quotes, as in line 370 of the examples, then a carriage return/line feed is executed.	360 PRINT X, Y; Z 370 PRINT " " 380 PRINT X, Y; 390 PRINT "VALUE IS"; A 400 PRINT A2, B,
	"VALUE IS" will be displayed and printed.	410 PRINT ! "VALUE IS"; A
	String expressions may be printed.	420 PRINT MID$(A$, 2);

STATEMENT	SYNTAX/FUNCTION	EXAMPLE
READ	READ variable [, variable] Read data into specified variables from a DATA statement. The first piece of data read will be the first piece of data listed in the first DATA statement of the program. The second piece of data read will be the second piece listed in the first DATA statement, and so on. When all of the data have been read from the first DATA statement, the next piece of data to be read will be the first piece listed in the second DATA statement of the program. Attempting to read more data than there is in all the DATA statements in a program will cause an OD (out of data) error.	490 READ V, W
SPC	SPC (expression) Prints I space (or blank) characters on the terminal. May be used only in a PRINT statement. I must be =>0 and <=255 or an FC error will result.	250 PRINT SPC(I)
TAB	TAB (expression) Spaces to the specified print position (column) on the printer. May be used only in PRINT statements. Zero is the leftmost column on the termainl, 19 the rightmost. If the carriage is beyond pos position I, then no printing is done. I must be =>0 and <=255. If I is greater than 19, the printer will skip the required number of lines to arrive at the specified position.	240 PRINT TAB(I)

SECTION

STATEMENT DEFINITIONS

SUBJECT

STRING FUNCTIONS

STATEMENT	SYNTAX/FUNCTION	EXAMPLE
ASC	ASC (string expression) Returns the ASCII numeric value of the first character of the string expression X$. See Appendix E for an ASCII/number conversion table. An FC error will occur if X$ is the null string.	300 PRINT ASC(X$)
CHR$	CHR$ (expression) Returns one character, the ASCII equivalent of the argument (I) which must be a number between 0 and 255. See Appendix E.	275 PRINT CHR$(I)
GET	GET string variable Inputs a single character from the keyboard. If data is at the keyboard, it is put in the variable specified in the GET statement. If no data is available, the BASIC program will continue execution. GET can only be used as an indirect command.	10 GET A$
LEFT$	LEFT$ (string expression, length) Gives the leftmost I characters of the string expression X$. If I<=0 or >255 an FC error occurs.	310 PRINT LEFT$(X$, I)
LEN	LEN (string expression) Gives the length of the string expression X$ in characters (bytes). Non-printing characters and blanks are counted as part of the length.	220 PRINT LEN(X$)

STATEMENT	SYNTAX/FUNCTION	EXAMPLE
MID$	MID$ (string expression, start [, length]) MID$ called with two arguments returns characters from the string expression X$ starting at character position I. If I> LEN(I$), then MID$ returns a null (zero length) string. If I <=0 or >255, an FC error occurs.	330 PRINT MID$(X$, I)
	MID$ called with three arguments returns a string expression composed of the characters of the string expression X$ starting at the Ith character for J characters. If I> LEN(X$), MID$ returns a null string. If I or J <=0 or >255, an FC error occurs. If J specifies more characters than are left in the string, all characters from the Ith on are returned.	340 PRINT MID$(X$, I, J)
RIGHT$	RIGHT$ (string expression, length) Gives the rightmost I characters of the string expression X$. When I<=0 or >255 an FC error will occur. If I>=LEN(X$) then RIGHT$ returns all of X$.	320 PRINT RIGHT$ (X$, I)
STR$	STR$ (expression) Gives a string which is the character representation of the numeric expression X. For instance, STR$(3.1)="3.1."	290 PRINT STR$(X)
VAL	VAL (string expression) Returns the string expression X$ converted to a number. For instance, VAL("3.1")=3.1. If the first non-space character of the string is not a plus (+) or minus (-) sign; a digit or a decimal point (.) then zero will be returned.	280 PRINT VAL(X$)

STATEMENT	SYNTAX/FUNCTION	EXAMPLE
ABS	ABS (expression) Gives the absolute value of the expression X. ABS returns X if X>=0, -X otherwise.	120 PRINT ABS(X)
ATN	ATN (expression) Gives the arctangent of the expression X. The result is returned in radians and ranges from -PI/2 to PI/2 (PI/2=1.5708). **If you want to use this function, you must provide the code in memory. See Appendix H for implementation details.**	210 PRINT ATN(X)
COS	COS (expression) Gives the cosine of the expression X. X is interpreted as being in radians.	200 PRINT COS(X)
EXP	EXP (expression) Gives the constant "E" (2.71828) raised to the power X (E∧X). The maximum argument that can be passed to EXP without overflow occurring is 88.0296.	150 PRINT EXP(X)
INT	INT (expression) Returns the largest integer less than or equal to its expression X. For example: INT(.23)=0, INT(7)=7, INT(-.1)=-1, INT(-2)=-2, INT(1.1)=1. The following would round X to D decimal places: INT(X*10∧D+.5)/10∧D	140 PRINT INT(X)
LOG	LOG (expression) Gives the natural (Base E) logarithm of its expression X. To obtain the Base Y logarithm of X use the formula LOG(X)/LOG(Y). Example: The base 10 (common) log of 7 = LOG(7)/LOG(10).	160 PRINT LOG(X)

STATEMENT	SYNTAX/FUNCTION	EXAMPLE
RND	RND (parameter) Generates a random number between 0 and 1. The parameter X controls the generation of random numbers as follows: X<0 starts a new sequence of random numbers using X. Calling RND with the same X starts the same random number sequence. X=0 gives the last random number generated. Repeated calls to RND(0) will always return the same random number. X>0 generates a new random number between 0 and 1. Note that (B-A) * RND(1)+A will generate a random number between A and B.	170 PRINT RND(X)
SGN	SGN (expression) Gives 1 if X>0, 0 if X=0, and -1 if X<0.	230 PRINT SGN(X)
SIN	SIN (expression) Gives the sine of the expression X. X is interpreted as being in radians. **Note:** *COS(X) = SIN(X + 3.14159/2) and that 1 Radian = 180/PI degrees = 57.2958 degrees; so that the sine of X degrees = SIN(X/57.2958).*	190 PRINT SIN(X)
SQR	SQR (expression) Gives the square root of the expression X. An FC error will occur if X is less than zero.	180 PRINT SQR(X)
TAN	TAN (expression) Gives the tangent of the expression X. X is interpreted as being in radians.	200 PRINT TAN(X)

DERIVED FUNCTIONS

The following functions, while not intrinsic to BASIC, can be calculated using the existing BASIC functions:

FUNCTION	FUNCTION EXPRESSED IN TERMS OF BASIC FUNCTIONS
SECANT	SEC(X) = 1/COS(X)
COSECANT	CSC(X) = 1/SIN(X)
COTANGENT	COT(X) = 1/TAN(X)
INVERSE SINE*	ARCSIN(X) = ATN(X/SQR(-X * X+1))
INVERSE COSINE*	ARCCOS(X) = -ATN(X/SQR(-X * X+1))+1.5708
INVERSE SECANT*	ARCSEC(X) = ATN(SQR(X * X-1))+(SGN(X)-1) * 1.5708
INVERSE COSECANT*	ARCCSC(X) = ATN(1/SQR(X * X-1))+(SGN(X)-1) * 1.5708
INVERSE COTANGENT*	ARCCOT(X) = -ATN(X)+1.5708
HYPERBOLIC SINE	SINH(X) = (EXP(X)-EXP(-X))/2
HYPERBOLIC COSINE	COSH(X) = (EXP(X)+EXP(-X))/2
HYPERBOLIC TANGENT	TANH(X) = -EXP(-X)/(EXP(X)+EXP(-X)) * 2+1
HYPERBOLIC SECANT	SECH(X) = 2/(EXP(X)+EXP(-X))
HYPERBOLIC COSECANT	CSCH(X) = 2/(EXP(X)-EXP(-X))
HYPERBOLIC COTANGENT	COTH(X) = EXP(-X)/(EXP(X)-EXP(-X)) * 2+1

*These functions require the user-defined ATN function. See Appendix H for details.

FUNCTION	FUNCTION EXPRESSED IN TERMS OF BASIC FUNCTIONS
INVERSE HYPERBOLIC SINE	ARGSINH(X) = LOG(X+SQR(X*X+1))
INVERSE HYPERBOLIC COSINE	ARGCOSH(X) = LOG(X+SQR(X*X-1))
INVERSE HYPERBOLIC TANGENT	ARGTANH(X) = LOG((1+X)/(1-X))/2
INVERSE HYPERBOLIC SECANT	ARGSECH(X) = LOG((XQR(-X*X+1)+1)/X
INVERSE HYPERBOLIC COSECANT	ARGCSCH(X) = LOG((SGN(X)*SQR(X*X+1)+1)/X)
INVERSE HYPERBOLIC COTANGENT	ARGCOTH(X) = LOG((X+1)/(X-1))/2

If an error occurs, BASIC outputs an error message, returns to command level and displays the cursor. Variable values and the program text remain intact, but the program can not be continued and all GOSUB and FOR context is lost.

When an error occurs in a direct statement, no line number is printed.

Format of error messages:

Direct Statement	?XX ERROR
Indirect Statement	?XX ERROR IN YYYYY

In both of the above examples, "XX" will be the error code. The "YYYYY" will be the line number where the error occured for the indirect statement.

The following are the possible error codes and their meanings:

ERROR CODE	MEANING
BS	Bad Subscript. An attempt was made to reference a matrix element which is outside the dimensions of the matrix. This error can occur if the wrong number of dimensions are used in a matrix reference; for instance, LET A(1,1,1)=Z when A has been dimensioned DIM A(2,2).
CN	Continue error. Attempt to continue a program when none exists, an error occured, or after a new line was typed into the program.
DD	Double Dimension. After a matrix was dimensioned, another DIM statement for the same matrix was encountered. This error often occurs if a matrix has been given the default dimension 10 because a statement like A(I)=3 is encountered and then later in the program a DIM A(100) is found.
FC	Function Call error. The parameter passed to a math or string function was out of range. FC errors can occur due to: 1. A negative matrix subscript (LET A(-1)=0) 2. An unreasonably large matrix subscript (>32767)

ERROR CODE	MEANING
	3. LOG-negative or zero argument 4. SQR-negative argument 5. A^B with A negative and B not an integer 6. A call to USR before the address of the machine language subroutine has been patched in 7. Calls to MID$, LEFT$, RIGHT$, WAIT, PEEK, POKE, TAB, SPC or ON. . .GOTO with an improper argument.
ID	Illegal Direct. You cannot use an INPUT, DEF or GET statement as a direct command.
LS	Long String. Attempt was made by use of the concantenation operator to create a string more than 255 characters long.
NF	NEXT without FOR. The variable in a NEXT statement corresponds to no previously executed FOR statement.
OD	Out of Data. A READ statement was executed but all of the DATA statements in the program have already been read. The program tried to read too much data or insufficient data was included in the program.
OM	Out of Memory. Program too large, too many variables, too many FOR loops, too many GOSUB's, too complicated an expression, or any combination of the above. (see Appendix B)
OV	Overflow. The result of a calculation was too large to be represented in BASIC's number format. If an underflow (too small result) occurs, zero is given as the result and execution continues without any error message being printed.
RG	RETURN without GOSUB. A RETURN statement was encountered without a previous GOSUB statement being executed.
SN	Syntax error. Missing parenthesis in an expression, illegal character in a line, incorrect punctuation, etc.

ERROR CODE	MEANING
ST	String Temporaries. A string expression was too complex. Break it into two or more shorter expressions.
TM	Type Mismatch. The left side of an assignment statement was a numeric variable and the right side was a string, or vice versa; or, a function which expected a string argument was given a numeric one or vice versa.
UF	Undefined Function. Reference was made to a user function which has never been defined.
US	Undefined Statement. An attempt was made to GOTO, GOSUB or THEN to a statement which does not exist.
/0	Division by Zero

In order to make your program smaller and save space, the following hints may be useful.

1. Use multiple statements per line. There is a five-byte of overhead associated with each line in the program. Two of these five bytes contain the line number of the line in binary. This means that no matter how many digits you have in your line number (minimum line number is 0, maximum is 63999), it takes the same number of bytes. Putting as many statements as possible on a line will cut down on the number of bytes used by your program.

2. Delete all unnecessary spaces from your program. For instance:

   ```
   10 PRINT X, Y, Z
   ```

   uses three more bytes than

   ```
   10 PRINTX,Y,Z
   ```

   Note: All spaces between the line number and the first non-blank character are ignored.

3. Delete all REM statements. Each REM statement uses at least one byte plus the number in the comment text. For instance, the statement 130 REM THIS IS A COMMENT uses 24 bytes of memory.

   In the statement 140 X=X+Y: REM UPDATE SUM, the REM uses 14 bytes of memory including the colon before the REM.

4. Use variables instead of constants. Suppose you use the constant 3.14159 ten times in your program. If you insert a statement

   ```
   10 P=3.14159
   ```

   in the program, and use P instead of 3.14159 each time it is needed, you will save 40 bytes. This will also result in a speed improvement.

5. A program need not end with an END, so an END statement at the end of a program may be deleted.

6. Reuse variables. If you have a variable T which is used to hold a temporary result in one part of the program and you need a temporary variable later in your program, use it again. Or, if you are asking the terminal user to give a YES or NO answer to two different questions at two different times during the execution of the program, use the same temporary variable A$ to store the reply.

7. Use GOSUB's to execute sections of program statements that perform identical actions.

8. Use the zero elements of matrices; for instance, A(0), B(0,X).

STORAGE ALLOCATION INFORMATION

Simple (non-matrix) numeric and strong variables like V use 7 bytes; 2 for the variable name, and 5 for the value. Simple non-matrix string variables also use 7 bytes; 2 for the variable name, 1 for the length, 2 for a pointer, and 2 are unused.

Matrix variables require 7 bytes to hold the header, plus additional bytes to hold each matrix element. Each element that is an integer variable requires 2 bytes. Elements that are string variables or floating point variables require 3 bytes or 5 bytes, respectively.

String variables also use one byte of string space for each character in the string. This is true whether the string variable is a simple string variable like A$, or an element of a string matrix such as Q1$(5,2).

When a new function is defined by a DEF statement, 7 bytes are used to store the definition.

Reserved words such as FOR, GOTO or NOT, and the names of the intrinsic functions such as COS, INT and STR$ take up only one byte of program storage. All other characters in programs use one byte of program storage each.

When a program is being executed, space is dynamically allocated on the stack as follows:

1. Each active FOR. . .NEXT loop uses 22 bytes.

2. Each active GOSUB (one that has not returned yet) uses 6 bytes.

3. Each parenthesis encountered in an expression uses 4 bytes and each temporary result calculated in an expression uses 12 bytes.

The hints below should improve the execution time of your BASIC program. Note that some of these hints are the same as those used to decrease the space used by your programs. This means that in many cases you can increase the efficiency of both the speed and size of your programs at the same time.

1. Delete all unnecessary spaces and REM's from the program. This may cause a small decrease in execution time because BASIC would otherwise have to ignore or skip over spaces and REM statements.

2. *THIS IS PROBABLY THE MOST IMPORTANT SPEED HINT.*

   Use variables instead of constants. It takes more time to convert a constant to its floating point representation than it does to fetch the value of a simple or matrix variable. This is especially important within FOR. . .NEXT loops or other code that is executed repeatedly.

3. Variables which are encountered first during the execution of a BASIC program are allocated at the start of the variable table. This means that a statement such as 5 A=0:B=A:C=A, will place A first, B second, and C third in the symbol table (assuming line 5 is the first statement executed in the program). Later in the program, when BASIC finds a reference to the variable A, it will search only one entry in the symbol table to find A, two entries to find B and three entries to find C, etc.

4. Use NEXT statements without the index variable. NEXT is somewhat faster than NEXT I because no check is made to see whether the variable specified in the NEXT is the same as the variable in the most recent FOR statement.

Though implementations of BASIC on different computers are in many ways similar, there are some incompatibilities which you should watch for if you are planning to convert some BASIC programs that were not written in AIM 65 BASIC.

1. Matrix subscripts. Some BASICs use "[" and "]" to denote matrix subscripts. AIM 65 BASIC uses "(" and ")".

2. Strings. A number of BASICs force you to dimension (declare) the length of strings before you use them. You should remove all dimension statements of this type from the program. In some of these BASICs, a declaration of the form DIM A$(I,J) declares a string matrix of J elements each of which has a length I. Convert DIM statements of this type to equivalent ones in AIM 65 BASIC: DIM A$(J).

   AIM 65 BASIC uses "+" for string concatenation, not "," or "&".

   AIM 65 BASIC uses LEFT$, RIGHT$ and MID$ to take substrings of strings. Other BASICs uses A$(I) to access the Ith character of the string A$, and A$(I,J) to take a substring of A$ from character position I to character position J. Convert as follows:

OLD	AIM 65
A$(I)	MID$(A$,I,1)
A$(I,J)	MID$(A$,I,J-I+1)

   This assumes that the reference to a substring of A$ is in an expression or is on the right side of an assignment. If the reference to A$ is on the left hand side of an assignment, and X$ is the string expression used to replace characters in A$, convert as follows:

OLD	AIM 65
A$(I)=X$	A$=LEFT$(A$,I-1)+X$+MID$(A$,I+1)
A$(I,J)=X$	A$=LEFT$(A$,I-1)+X$+MID$(A$,J+1)

3. Multiple assignments. Some BASICs allow statements of the form: 500 LET B=C=0. This statement would set the variables B & C to zero.

In AIM 65 BASIC this has an entirely different effect. All the "="s" to the right of the first one would be interpreted as logical comparison operators. This would set the variable B to -1 if C equaled 0. If C did not equal 0, B would be set to 0. The easiest way to convert statements like this one is to rewrite them as follows:

```
500 C=0:B=C.
```

4. Some BASICs use "/" instead of ":" to delimit multiple statements per line. Change all occurrences of "/" to ":" in the program.

5. Programs which use the MAT functions available in some BASICs will have to be re-written using FOR. . .NEXT loops to perform the appropriate operations.

6. A PRINT statement with no arguments will not cause a paper feed on the printer. To generate a paper feed (blank line), use PRINT "space"

DECIMAL	CHAR.	DECIMAL	CHAR.	DECIMAL	CHAR.
000	NUL	043	+	086	V
001	SOH	044	,	087	W
002	STX	045	-	088	X
003	ETX	046	.	089	Y
004	EOT	047	/	090	Z
005	ENQ	048	0	091	[
006	ACK	049	1	092	/
007	BEL	050	2	093	]
008	BS	051	3	094	↑
009	HT	052	4	095	←
010	LF	053	5	096	`
011	VT	054	6	097	a
012	FF	055	7	098	b
013	CR	056	8	099	c
014	SO	057	9	100	d
015	SI	058	:	101	e
016	DLE	059	;	102	f
017	DC1	060	<	103	g
018	DC2	061	=	104	h
019	DC3	062	>	105	i
020	DC4	063	?	106	j
021	NAK	064	@	107	k
022	SYN	065	A	108	l
023	ETB	066	B	109	m
024	CAN	067	C	110	n
025	EM	068	D	111	o
026	SUB	069	E	112	p
027	ESCAPE	070	F	113	q
028	FS	071	G	114	r
029	GS	072	H	115	s
030	RS	073	I	116	t
031	US	074	J	117	u
032	SPACE	075	K	118	v
033	!	076	L	119	w
034	"	077	M	120	x
035	#	078	N	121	y
036	$	079	O	122	z
037	%	080	P	123	{
038	&	081	Q	124	|
039	'	082	R	125	}

DECIMAL	CHAR.	DECIMAL	CHAR.	DECIMAL	CHAR.
040	(	083	S	126	~
041	)	084	T	127	DEL
042	*	085	U		

!

LF=Line Feed FF=Form Feed CR=Carriage Return DEL=Rubout on TTY

AIM 65 BASIC allows a user to link to assembly language subroutines, via the USR(W) function. This function allows one parameter to be passed between BASIC and a subroutine.

The first step is to allocate sufficient memory for the subroutine. AIM 65 BASIC always uses all RAM memory locations, beginning at decimal location 530 (hex location 212), unless limited by the user. You can limit BASIC's memory useage by answering the prompt MEMORY SIZE? (see Subject 100) with some number less than 4096, assuming a 4K system. This will leave sufficient space for the subroutine at the top of RAM.

For example, if your response to MEMORY SIZE? is "2048", 1518 bytes at the top of RAM will be free for assembly language subroutines.

Parameter (W), passed to a subroutine by USR(W), will be converted to floating-point accumulator located at $A9. The floating-point accumulator has the following format:

ADDRESS	CONTENT
$A9	Exponent + $81 ($80 if mantissa = 00)
$AA - $AD	Mantissa, normalized so that Bit 7 of MSB is set. $AA is MSB, $AD is LSB.
$AE	Sign of mantissa

A parameter passed to an assembly language subroutine from BASIC can be truncated by the subroutine to a 2-byte integer and deposited in $AC (MSB) and $AD (LSB). If the parameter is greater than 32767 or less than -32768, an FC error will result. The address of the subroutine that converts a floating-point number to an integer is located in $B006, $B007.

A parameter passed to BASIC from an assembly language subroutine will be converted to floating-point. The address of the subroutine that performs this conversion is in $B008, $B009. The integer MSB ($AC) must be in the accumulator; the integer LSB ($AD) must be in the Y register.

Prior to executing USR, the starting address of the assembly language subroutine must be stored in locations $04 (LSB) and $05 (MSB). This is generally performed using the POKE command. Note that more than one assembly language subroutine may be called from a BASIC program, by changing the starting address in $04 and $05.

Figure F-1 is the listing for a BASIC program that calls an assembly language subroutine located at $A00. Here's what the BASIC program does:

- Line 10 — Stores the starting address of the assembly language subroutine ($A00) into locations $04 and $05, using POKE.
- Line 20 — Asks for a number "N".
- Line 30 — Calls the subroutine, with N as the parameter.
- Line 40 — Upon return from the subroutine, the BASIC program prints X, the parameter passed from the subroutine to the BASIC program.
- Line 50 — Loops back to get a new "N".

```
  ROCKWELL AIM 65

<5>
MEMORY SIZE? 2048
WIDTH?
 1510 BYTES FREE
AIM 65 BASIC V1.1
OK
10 POKE 04,0:POKE 05
,10
20 INPUT"NUMBER";N
30 X=USR(N)
40 PRINTX
50 GOTO 20
```

Figure F-1. BASIC Program That Calls Assembly Language Subroutine

The assembly language subroutine (Figure F-2) performs these operations:

- Prints the floating-point accumulator ($A9 - $AE), using Monitor subroutines NUMA ($EA46), BLANK ($E83E) and CRLF ($E9F0).

- Converts the floating-point accumulator to an integer, using the subroutine at $BF00. The address $BF00 was found in locations $B006, $B007. (Address $BF00 may vary with different versions of BASIC. Be sure to check locations $B006 and $B007 for the correct address.)

- After conversion, the program again prints the floating-point accumulator.

- The program then swaps the bytes of the integer.

- Finally, the program converts the result to floating-point and returns to BASIC (JMP C0D3). Address $C0D3 was found in locations $B008, $B009. (Address $C0D3 may vary with different versions of BASIC. Be sure to check locations $B008 and $B009 for the correct address.

```
<1>
0A26          *=A00
0A00  A0  LDY  #00
0A02  A2  LDX  #00
0A04  B5  LDA  A9,X
0A06  20  JSR  EA46
0A09  20  JSR  E83E
0A0C  E8  INX
0A0D  E0  CPX  #05
0A0F  D0  BNE  0A04
0A11  20  JSR  E9F0
0A14  C0  CPY  #00
0A16  F0  BEQ  0A1F
0A18  A5  LDA  AD
0A1A  A4  LDY  AC
0A1C  4C  JMP  C0D3
0A1F  20  JSR  BF00
0A22  C8  INY
0A23  D0  BNE  0A02
0A25  60  RTS
0A26
```

Figure F-2. Assembly Language Subroutine

Figure F-3 shows the print-out for various values of "N".

```
(6)
OK
RUN
NUMBER? 128
80 80 00 00 00 00
80 00 00 00 80 00
-32768


NUMBER? 1
81 80 00 00 00 00
81 00 00 00 01 00
 256


NUMBER? 4097
8D 80 08 80 00 00
8D 00 00 10 01 00
 272


NUMBER? 256
89 80 00 80 00 00
89 00 00 01 00 00
 1
```

Figure F-3. Output for Example

AIM 65 BASIC programs can be stored on cassette tape by using BASIC's SAVE and LOAD commands, or by using the AIM 65 Editor. Before employing either procedure, be sure to carefully observe the recorder installation and operation procedures given in Section 9 of the AIM 65 User's Guide.

RECORDING ON CASSETTE USING THE BASIC SAVE COMMAND

The procedure to store a BASIC program is:

1. Install a cassette in the recorder, and manually position the tape to the program record position. Be sure to initialize the counter at the start of the tape.

   Note: Since remote control must be used to retrieve a BASIC program, observe the tape gap CAUTION in Section 9.1.5 (Step 1) of the AIM 65 User's Guide.

2. While in BASIC, type in SAVE. BASIC will respond with:

   ```
   OUT=
   ```

3. Enter a T (for "Tape"). BASIC will display:

   ```
   OUT=T  F=
   ```

4. Enter the file name (up to five characters). If the file name is FNAME, BASIC will display:

   ```
   OUT=T  F=FNAME  T=
   ```

5. Put the recorder into Record mode.

6. Enter the recorder number (1 or 2) and type RETURN.

7. If remote control is being used, observe the procedures outlined in Section 9.1.5 of the AIM 65 User's Guide.

8. When recording has been completed, BASIC will display the cursor.

9. Switch the recorder out of record mode.

RETRIEVING A PROGRAM FROM CASSETTE USING THE BASIC LOAD COMMAND

The procedure to retrieve a BASIC program is:

1. Install the cassette in the recorder, and manually position the tape to about five counts before the beginning of the desired file.

   Note: Remote control must be used when retrieving a file via BASIC.

2. While in BASIC, type in LOAD. BASIC will respond with:

   ```
   IN=
   ```

3. Enter a 1 (for "Tape"). BASIC will display:

   ```
   IN=T  F=
   ```

4. Enter the file name. If the file name is FNAME, BASIC will display:

   ```
   IN=T  F=FNAME  T=
   ```

5. Enter the recorder number (1 or 2) and type RETURN.

6. Put the recorder into play mode. Be sure to observe the procedures outlined in Section 9.1.6 of the AIM 65 User's Guide.

   While the file is being read, each line will be displayed (and printed, if the printer is on). If the printer is on, the tape gap ($A409) will probably have to be increased.

   The file being loaded will not overlay any BASIC statements already entered unless the statement numbers are the same.

7. When loading has been completed, BASIC will display the cursor.

8. Switch the recorder out of play mode.

CASSETTE OPERATIONS USING THE AIM 65 EDITOR

AIM 65 BASIC programs can also be stored and retrieved from cassette using the AIM 65 Editor. However, if the program is to be retrieved by BASIC at some future time, one rule must be observed:

> When BASIC stores a program on cassette, it inserts a CTRL/Z after the last line. The AIM 65 Editor will strip off the CTRL/Z when it retrieves the program. Therefore, before storing a BASIC program from the Editor, the user must insert a CTRL/Z following the last line of the program.

The ATN function (see Subject 307) can be programmed in RAM using the AIM 65 Mnemonic Entry (I) and Alter Memory Locations (/) commands, as shown below. The program is written for the AIM 65 with 4K bytes of RAM. The ATN function can be relocated elsewhere in memory by changing the starting addresses of the instructions and constants, the conditional branch addresses, the vector to the constants start address and the vector to the ATN function start address.

ATN FUNCTION CONSTANTS ENTERED BY ALTER MEMORY <M>

```
<M> = 0F80   XX   XX   XX   XX      Constants Starting Address = 0F80(8)
</>  = 0F80   0B   76   B3   83
</>    0F84   BD   D3   79   1E
</>    0F88   F4   A6   F5   7B
</>    0F8C   83   FC   B0   10
</>    0F90   7C   0C   1F   67
</>    0F94   CA   7C   DE   53
</>    0F98   CB   C1   7D   14
</>    0F9C   64   70   4C   7D
</>    0FA0   B7   EA   51   7A
</>    0FA4   7D   63   30   88
</>    0FA8   7E   7E   92   44
</>    0FAC   99   3A   7E   4C
</>    0FB0   CC   91   C7   7F
</>    0FB4   AA   AA   AA   13
</>    0FB8   81   00   00   00
</>    0FBC   00
```

ATN FUNCTION INSTRUCTIONS STORED BY MNEMONIC ENTRY (I)

```
<I>
XXXX *=0FBD                      Instructions Starting Address = 0FBD
0FBD   A5  LDA   AE
0FBF   48  PHA
0FC0   10  BPL   0FC5
0FC2   20  JSR   CCB8
0FC5   A5  LDA   A9
0FC7   48  PHA
0FC8   C9  CMP   #81
0FCA   90  BCC   0FD3
0FCC   A9  LDA   #FB
0FCE   A0  LDY   #C6
0FD0   20  JSR   C84E
```

```
0FD3   A9   LDA   #80 }
0FD5   A0   LDY   #0F }   Starting Address of Constants = 0F80
0FD7   20   JSR   CD44
0FDA   68   PLA
0FDB   C9   CMP   #81
0FDD   90   BCC   0FE6
0FDF   A9   LDA   #4E
0FE1   A0   LDY   #CE
0FE3   20   JSR   C58F
0FE6   68   PLA
0FE7   10   BPL   0FEC
0FE9   4C   JMP   CCB8
0FEC   60   RTS
0FEC
```

BASIC INITIALIZATION FOR ATN FUNCTION

BASIC memory must be initialized below the memory allocated to the ATN function. The ATN vector in RAM must also be changed from the address of the FC error message to the starting address of the ATN function instructions. This can be done using BASIC initialization, as follows:

```
<5>
MEMORY SIZE? 3968              Limit BASIC to F80(16)
WIDTH?
 3438 BYTES FREE
  AIM 65 BASIC V1.1
POKE 188, 189                  Change ATN function vector low to BD(16)
POKE 189, 15                   Change ATN function vector high to 0F(16)
?ATN (TAN(.5))                 Test case to verify proper ATN function program
 .5                            Expected answer = .5
```

SAVING ATN OBJECT CODE ON CASSETTE

The object code for the ATN function can be saved on cassette by dumping addresses $00BB through $00BD (Jump instruction to ATN) and $0F80 through $0FEC (constants and instructions) after the function is initially loaded and verified.

The ATN function can then be loaded from cassette by executing the Monitor L command after BASIC has been initialized via the 5 command. After the ATN function has been loaded, reenter BASIC with the 6 command.

NOTES