MICROSOFT 6502

BASIC

REFERENCE MANUAL

INTRODUCTION

Before a computer can perform any useful function, in 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.

6502 BASIC* 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. With 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, 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 6502 BASIC will immediately reply with "4". It isn't necessary to submit a card deck with your program on it and then wait hours for the results. Instead the full power of the 6502 is "at your fingertips".

Generally, if the computer does not solve a particular problem the way you expect it to, there is a "bug" or error in your program, or else there is an error in the data which the program used to calculate its answer. If you encounter any errors in BASIC itself, please let us know and we'll see that it is corrected. Write a letter to us containing the following information:

1) System Configuration

2) Version of BASIC

3) A detailed description of the error.

   Include all pertinent information such as a listing of the program which the error occurred, the data placed into the program and BASIC's print-out.

All of the information listed above will be necessary in order to properly evaluate the program and correct it as quickly as possible. We wish to maintain as high a level of quality as possible with all of our software.

*BASIC is a registered trademark of Dartmouth University

We hope that you enjoy 6502 BASIC, and are successful in using it to solve all of your programming needs.

In order to maintain a maximum quality level in our documentation, we will be continuously revising this manual. If you have any suggestions on how we can improve it, please let us know.

If you are already familiar with BASIC programming, the following section may be skipped. Turn directly to the Reference Material on page 22.

NOTE: Basic is available under license or purchase agreements. Copying or otherwise distributing Microsoft software outside the terms of such an agreement may be a violation of copyright laws or the agreement itself.

If any immediate problems with Microsoft software are encountered feel free to give as a call at (505) 256-3600. The joint authors of the 6502 BASIC Interpreter will be glad to help you.

GETTING STARTED WITH BASIC

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. The text here will introduce the primary concepts and uses of BASIC enough to get you started writing programs. For further reading suggestions, see Appendix M.

If your 6502 KIM does not have BASIC loaded and running, follow the procedures in Appendices A & B to bring it up.

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.

Once your I/O device has typed "OK", you are ready to use 6502 BASIC.

NOTE: All commands to 6502 BASIC must end with a carriage return. The carriage return tells BASIC that you have finished typing the command. If you make a typing error, type a backspace, usually a CTRL/H, to eliminate the last character. Repeated use of the backspace will eliminate previous characters. An at-sign (@) will eliminate the entire line that you are typing.

Now, try typing in the following:

```
PRINT 10-4 (end with a carriage return)
```

6502 BASIC will immediately print:

```
6

OK
```

The print statement you typed-in was executed as soon as you hit the carriage return key. 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, 6502 BASIC can do division and multiplication as well as subtraction. Note how a " , " (comma) was used with the print command to print two values instead of just one. The comma divides the 72 character line into 5 columns, each 14 characters wide. The last two of the positions on the line are not used. The result is " , " causes BASIC to skip to the next 14 column field on the terminal, where the value 30 was printed.

Commands, such as the "PRINT" statements you have just typed in are called Direct 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 64000.

Try typing in the following lines:

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

A sequence of Indirect Commands is called a "Program". Instead of executing indirect statements immediately, 6502 BASIC saves Indirect Commands in the memory. When you type in RUN , BASIC will execute the lowest numbered indirect statement that has been typed in first, then the next highest, etc. for as many as were typed in.

Suppose we type in RUN now:

```
RUN
```

6502 BASIC will type out:

```
 5
-1

OK
```

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.

If we want a listing of the complete program currently in memory, we type in LIST . Type this in:

```
LIST
```

6502 BASIC will reply with:

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

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

Type in the following:

```
10
LIST
```

6502 BASIC will reply with:

```
20 PRINT 2-3
OK
```

We have now deleted line 10 from the program. There is no way to get it back. To insert a new line 10, just type in 10 followed by the statement we want BASIC to execute.

Type in the following:

```
10 PRINT 2*3
LIST
```

6502 BASIC will reply with:

```
10 PRINT 2*3
20 PRINT 2-3
OK
```

There is an easier way to replace line 10 than deleting it and then inserting a new line. You can do this by just typing the new line 10 and hitting the carriage return. BASIC throws away the old line 10 and replaces it with the new one.

Type in the following:

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

6502 BASIC will reply with:

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

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.

If you want to erase 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. This should be done in order to prevent a mixture of the old and new programs.

Type in the following:

```
NEW
```

6502 BASIC will reply with:

```
OK
```

Now type in:

```
LIST
```

6502 BASIC will reply with:

```
OK
```

Often it is desirable to include text along with answers that are printed out, in order to explain the meaning of the numbers.

Type in the following:

```
PRINT "ONE THIRD IS EQUAL TO",1/3
```

6502 BASIC will reply with:

```
ONE THIRD IS EQUAL TO          .333333333

OK
```

As explained earlier, including a " , " in a print statement causes it to space over to the next fourteen column field before the value following the " , " is printed.

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

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

Try the following examples:

A)
```
PRINT "ONE THIRD IS EQUAL TO";1/3
ONE THIRD IS EQUAL TO .333333333

OK
```

B)
```
PRINT 1,2,3
 1              2              3

OK
```

C)
```
PRINT 1;2;3
 1  2  3

OK
```

D)
```
PRINT -1;2;-3
-1  2 -3

OK
```

We will digress for a moment to explain the format of numbers in 6502 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 represented in 6502 BASIC is 1.70141000*10 to 38th pwr, while the smallest positive number is 2.93874000*10 to -39th pwr.

When a number is printed, the following rules are used to determine the exact 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 of 0 to 999999, it is printed as an integer.

3) If the absolute value of the number is greater than or equal to .1 and less than or equal to 999999, 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 being some integer 0 to 9)

The leading "S" is the sign of the number, a space for a positive number and a " - " 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. Also, 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 poistive 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".

No matter what format is used, a space is always printed following a number. 6502 BASIC checks to see if the entire number will fit on the current line. If not, a carriage return/line feed is executed before print

The following are examples of various numbers and the output format 6502 BASIC will place them into:

NUMBER	OUTPUT FORMAT
+1	1
-1	-1
6523	6523
-23.460	-23.46
1E20	1E+20
-12.3456E-7	-1.234567E-06
1.23456789E-10	1.23456789E-10
10000000	1E+07
999999	999999
.1	.1
.01	1E-02
.000123	1.23E-04

A number input from the terminal 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). However, only the first 9 digits are significant, and the ninth digit is rounded up.

```
PRINT 1.23456789012345678900
 1.23456789

 OK
```

The following is an example of a program that reads a value from the terminal and uses that value to calculate and print a result:

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

OK
```

Here's what's happening. When BASIC encounters the input statement, it types a question mark (?) on the terminal 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 you haven't already guessed, what the program above actully does is to calculate the area of a circle with the radius "R".

If we wanted to calculate the area of various circles, we could keep re-running the program over each time 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
?

OK
```

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 question mark (thus a blank line).

The letter "R" in the program we just used was termed a "variable". A variable name can be any alphabetic character and may be followed by any alphanumeric character. All alphanumeric characters after the first two are ignored. An alphanumeric character is any letter (A-Z) or any number (0-9).

Below are some examples of legal and illegal variable names:

LEGAL	ILLEGAL
A	% (1st character must be alphabetic)
Z1	TO (variable names cannot be reserved words)
TP	RGOTO (variable names cannot contain reserved words)
PSTG$	
COUNT	

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, "PEND" would be illegal because "END" is a reserved word.

The following is a list of the reserved words in 6502 BASIC:

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

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

OK

PRINT A,A*2
 5               10

OK
LET Z=7

OK

PRINT Z,Z-A
 7               2

OK
```

As can be seen 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 thrown away and the space in memory used to store them is released when one of four things occur:

1) A new line is typed into the program or an old line is deleted

2) A CLEAR command is typed in

3) A RUN command is typed in

4) A NEW command 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
 Ø                2                Ø

OK
```

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

This serves mainly as an aid for the programmer himself, 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 if a number is zero or not. 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.

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

```
10 INPUT B
20 IF B=Ø THEN 5Ø
30 PRINT "NON-ZERO"
40 GOTO 1Ø
5Ø PRINT "ZERO"
6Ø GOTO 1Ø
```

When this program is typed into the BASIC and run, it will ask for a value for B. Type any value you wish in. The BASIC 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
<=	LESS THAN OR EQUAL TO
=>	GREATER THAN OR EQUAL TO

The IF statement is either true or false, depending upon whether the two expressions satisfy the relation or not. For example, in the program we just did, if Ø was typed in for the B the IF statement would be true because Ø=Ø.

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.

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 terminal. 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 some 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 annd again we go back to the beginning.

Try running the last two programs several times. It may make it 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 carriage 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.

Suppose that we want a table of square roots from 1 to 10. The BASIC function for square root is "SQR"; the form being SQR(X), X being the number you wish the square root calculated from. We could write the program as follows:

```
10 PRINT 1,SQR(1)
20 PRINT 2,2QR(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)
100 PRINT 10,SQR(10)
```

This program will do the job; however, it is terribly inefficient. We can improve the program tremendously by using the IF statement just introduced as follows:

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

When this program is run, its output will look exactly like that of the 10 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 at 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 caluclation 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<=10 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 10 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.

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 10
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 1 to be added to N, and then if N<=10 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.

Suppose we wanted to print a table of square roots from 10 to 20, only counting by two's. The following program would perform this task:

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

Note the similar structure between this program and the one listed on page 12 for printing square roots for the numbers 1 to 10. 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 one is to be added each time. The "STEP" can be followed by any expression.

Suppose we wanted to count backwards 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>= 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.

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

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

"FOR" loops can also be "nested". An example of this procedure follows:

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

Notice that the "NEXT J" comes before the "NEXT I". This is because the J-loop is inside of 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" of 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, called 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 parenthesis (I) and then follow "A" by this subscript. Therefore, "A(I)" is the I'th element in the matrix "A".

NOTE: In this section of the manual we will be concerned with one-dimensional matrices only. (See References)

"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).

As an example of how matrices are used, try the following program to sort a list of 8 numbers with you picking 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 upon finding any two that are not in order, switching them. "F" is used to indicate if any switches were done. If any were done, 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.

Another useful pair of statements are "GOSUB" and "RETURN". 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"-"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 branched. 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 "WHAT IS THE SECOND NUMBER";
70 GOSUB 100
80 PRINT "THE SUM OF THE TWO NUMBERS IS",T+N
90 STOP
100 INPUT N
110 IF N=INT(N) THEN 140
120 PRINT "NUMBER MUST BE AN INTEGER. TRY AGAIN."
130 GOTO 100
140 RETURN
```

What this program does is to ask for two numbers which must be integers, and then prints the sum of the two. The subroutine in this program is lines 100 to 130. The subroutine asks for a number, and if it is not an integer, asks for a number again. 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.

" WHAT IS THE 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 OF THE TWO NUMBERS IS " is printed, followed by the value of their sum. T contains the value of the first number that was entered and N contains the value of the second number.

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 not included in 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 must have a matching "RETURN" executed later, and the opposite applies, i.e. 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 didn't change each time the program was run, but you would like it to be easy to change them if necessary. BASIC contains special statements for this purpose, called the "READ" and "DATA" statements.

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
```

This is what happens when this program is run. When the "READ"

statement is encountered, the effect is the same as an INPUT statement. But, instead of getting a number from the terminal, 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 returned. The second time one is needed, the second number in the first DATA statement is returned. When the entire contents of the first DATA statement have been read in this manner, the second DATA statement will then 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) have been read. Therefore, if -999999 was read, we know that the guess given 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 piece 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". KIM, BASIC, and THIS IS A TEST are all strings. Like numeric variables, string variables can be assigned specific values. String variables are distinguished from numseric variables by a "$" after the variable name.

For example, try the following:

```
A$="KIM COMPUTER"

OK
PRINT A$
KIM COMPUTER

OK
```

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

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("KIM")
  12              3

OK
```

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

The number of characters in a string expression may range from Ø to 255. A string which contains Ø 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 print head or cursor not be advanced to the next column. Try the following:

```
PRINT LEN(Q$);Q$;3
 Ø  3

OK
```

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.

Often it is desirable to access parts of a string and manipulate them. Now that we have set A$ to "KIM COMPUTER", we might want to print out only the first six characters of A$. We would do so like this:

```
PRINT LEFT$ (A$,6)
KIM CO

OK
```

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

```
FOR N=1 to LEN(A$):PRINT LEFT$(A$,N): NEXT N
K
KI
KIM
KIM
KIM C
KIM CO
KIM COM
KIM COMP
KIM COMPU
KIM COMPUT
KIM COMPUTE
KIM COMPUTER

OK
```

Since A$ has 12 characters, this loop will be executed with N=1,2,3,....,11,12. The first time through only the first character will be printed, the second time the first two characters will be printed,etc.

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

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
KIM COMPUTER
IM COMPUTER
M COMPUTER
 COMPUTER
COMPUTER
OMPUTER
MPUTER
PUTER
UTER
TER
ER
R

OK
```

"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
K               KI
I               IM
M               M
                 C
C               CO
O               OM
M               MP
P               PU
U               UT
T               TE
E               ER
R               R

OK
```

See the reference material for more details on the workings of "LEFT$", "RIGHT$", and "MID$".

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

```
B$="MOS"+" "+A$

OK
PRINT B$
MOS KIM COMPUTER

OK
```

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$,3)+"-"+MID$(B$,5,3)+"-"+RIGHT$(B$,8)

OK
PRINT C$
MOS-KIM-COMPUTER

OK
```

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"

OK
PRINT VAL(STRING$)
 567.8

OK
STRING$=STR$(3.1415)

OK
PRINT STRING$,LEFT$(STRING$,5)
 3.1415         3.14

OK
```

"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:

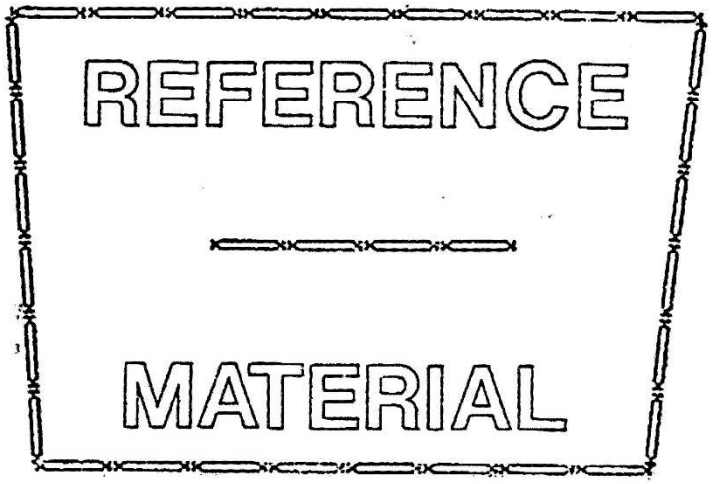
```
PRINT LEN(STR$(3.157))
 6

OK
```

If you have an application where a user is typing in a question such as "WHAT IS THE VOLUME OF CYLINDER OF RADIUS 5.36 FEET, OF HEIGHT 5.1 FEET?" you can use "VAL" to extract the numberic values 5.36 and 5.1 from the question. For further functions "CHR$" and "ASC" see Appendix K.

The following program sorts a list of string data and prints out the sorted list. This program is very similar to the one given earlier for sorting a numeric.

```
100 DIM A$(15):REM ALLOCATE SPACE FOR STRING MATRIX
110 FOR I=1 TO 15:READ A$(I):NEXT I:REM READ IN STRINGS
120 F=0:I=1:REM SET EXCHANGE FLAG TO ZERO AND SUBSCRIPT TO 1
130 IF A$(I)<=A$(I+1) THEN 180: REM DON'T EXCHANGE IF ELEMENTS
    IN ORDER
140 T$=A$(I+1):REM USE T$ TO SAVE A$(I+1)
150 A$(I+1)=A$(I):REM EXCHANGE TWO CONSECUTIVE ELEMENTS
160 A$(I)=T$
170 F=1:REM FLAG THAT WE EXCHANGED TWO ELEMENTS
180 I=I+1: IF I<15 GOTO 130
185 REM ONCE WE HAVE MADE A PASS THRU ALL ELEMENTS, CHECK
187 REM TO SEE IF WE EXCHANGED ANY.  IF NOT, DONE SORTING.
190 IF F THEN 120:REM EQUIVALENT TO IF F<>0 THEN 120
200 FOR I=1 TO 15:PRINT A$(I):NEXT I:REM PRINT SORTED LIST
210 REM STRING DATA FOLLOWS
220 DATA APPLE,DOG,CAT,KIM,COMPUTER,MOS,RANDOM
230 DATA MONDAY,":::ANSWER:::"," BOO"
240 DATA PRINT-OUT,   BOO,ELP,MILWAUKEE,SEATTLE,ALBUQUERQUE
```


REFERENCE
MATERIAL

COMMANDS

A command is usually given after BASIC has typed OK. 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.

NAME	EXAMPLE	PURPOSE/USE
CLEAR	CLEAR	Clears all variables, resets "FOR" & "GOSUB" state, RESTORES data.
LIST	LIST	Lists current program
	LIST 100-	Starting at specified line
	LIST XXX	List line XXX
	LIST XXX-YYY	List lines XXX to YYY inclusive
NULL	NULL 3	Sets the number of null (ASCII Ø) characters printed after a carriage return/line feed. The number of nulls printed may be set from Ø to 71. This is a must for hardcopy terminals that require a delay after a CRLF*. It is necessary to set the number of nulls typed on CRLF to Ø before a paper tape of a program is read in from a Teletype (Teletype is a registered trademark of the Teletype Corporation). When you punch a paper tape of a program with a Teletypewriter punch and using the list command, null should be set >=3 for 10 CPS terminals, >=6 for 30 CPS terminals. When not making a tape, we recommend that you use a null setting of Ø or 1 for Teletypes, and 2 or 3 for hard copy 30 CPS printers. A setting of Ø will work with Teletype compatible CRT's. Use the null command to set the number of nulls to zero.
RUN	RUN	Starts execution of the program currently in memory at the lowest numbered statement. 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.

*CRLF=carriage return/line feed

	RUN 200	Starts execution at the specified line number.
NEW	NEW	Deletes current program and all variables.
CONT	CONT	Continues program execution after a stop statement is executed. You cannot continue after any error, after modifiying 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 escape. The KIM will keep executing the series of statements over and over, until you intervene. If you suspect that your program is in an infinite loop, hit the "ST" button on the KIM, and press the space-bar on the terminal. When a "0000 4C" appears, a "G" will place you back in BASIC. After BASIC has typed "OK", you can use PRINT to type out some of the values of your variables. After examining these values you may become satisfied that the program is functioning properly. You 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 (LET) statements to set some of your variables to different values. Remember, if you "ST" a program 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 an error message. It is impossible to continue a direct command. CONT always resumes execution at the next statement to be executed in your program when you hit the KIM "ST" button.
LOAD	LOAD	Loads the program from the cassette tape. A NEW command is automatically done before the LOAD command is executed. When done, a "0000 4C" will appear if the load was successful. Typing a "G" will put you back in basic, and "LOADED" will be printed. See Appendix I for more information.

SAVE	SAVE	Saves on cassette tape the current program in the KIM memory. The program in memory is left unchanged. See Appendix I for more information.

OPERATORS

SYMBOL	SAMPLE STATEMENT	PURPOSE/USE
=	A=100 LET Z=2.5	Assigns a value to a variable The LET is optional
-	B=-A	Negation. Note that 0-A is subtraction, while -A is negation.
^	130 PRINT X^3	Exponentiation. Equal to X*X*X in the sample statement. 0^0=1 (0 to any other power = 0). A^B, with A negative and B not an integer gives an 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 division 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 divide 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 highes precedence:
(Note: Operators listed on the same line have the same precedence.)

1) FORMULAS ENCLOSED IN PARENTHESIS ARE ALWAYS EVALUATED FIRST

2) ^ EXPONENTIATION

3)	NEGATION	-X WHERE X MAY BE A FORMULA
4)	* /	MULTIPLICATION AND DIVISION
5)	+ -	ADDITION AND SUBTRACTION
6)	RELATIONAL OPERATORS (equal precedence for all six)	= EQUAL <> NOT EQUAL < LESS THAN > GREATER THAN <= LESS THAN OR EQUAL >= GREATER THAN OR EQUAL
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 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

CONT.

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=Ø (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 error results.

The operations are performed in bitwise fashion, which 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	ARG. 1	ARG. 2	RESULT
AND	1	1	1
	Ø	1	Ø
	1	Ø	Ø
	Ø	Ø	Ø
OR	1	1	1
	Ø	1	1
	1	Ø	1
	Ø	Ø	Ø
NOT	1	-	Ø
	Ø	-	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 KIM input locations which reflect the state of some external device. As an example, a PEEK (5888) would instruct BASIC to look at KIM address 1700 (Hex) which is the address for the input lines PA-0 through PA-7. Suppose that you were only interested in PA-0, then the command

```
PEEK (5888) AND 1
```

would mask out all other lines, and only PA-0 would be sampled.

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

In another example, suppose bit 1 of location 5000 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 (5000) AND 2) THEN 10    (This line will execute
                                          over and over until bit
                                          1 becomes a "1". When
                                          that happens, we will go
                                          to the next line (20).
20 PRINT "INTRUDER ALERT"
```

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

```
10 WAIT 5000,2
```

This line delays the execution of the next statement in the program until bit 1 of location 5000 becomes 1. The WAIT is much faster than the equivalent IF statement and also takes less bytes of program storage.

Sense switches may also be used as an input device by the PEEK function. The program below prints out any changes in the sense switches.

```
10 A=300:REM SET A TO A VALUE THAT WILL FORCE PRINTING
20 J=PEEK (SWITCH LOCATION):IF J=A THEN 20
30 PRINT J;:A=J:GOTO 20
```

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.

STATEMENTS

NOTE: In the following description of statements, an argument of 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 and 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 ("KIM").

NAME	EXAMPLE	PURPOSE/USE
DATA	10 DATA 1,3,-1E3,.04	Specifies data, read from left to right. Information appears in data statements in the same order as it will be read in the program.
	20 DATA " KIM",ZOO	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 impossible to have a double quote within string data or a string literal. (""KIM"" is illegal)

DEF 100 DEF FNA(V)=V/B+C

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, FNKO. 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 & 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. User defined string functions are not allowed. "V" is called the dummy variable.

110 Z=FNA(3)

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.

DIM 113 DIM A(3),B(10)

Allocates space for matrices. All matrix elements are set to zero by the DIM statement.

114 DIM R3(5,5,8)

Matrices can have more than one dimension. Up to 255 dimensions are allowed, but due to the restriction of 72 characters per line the practical limit is about 34 dimensions.

115 DIM Q1(N),Z(2*I)

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 from 0 to 10 (eleven elements).

117 A(8)=4

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.

END 999 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 it is optional.

FOR 300 FOR V=1 TO 9.3 STEP .6

(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.

```
310 FOR V=1 TO 9.3
```

If not STEP was specified, it is assumed to be one. If the STEP is positive and the new value of the variable is <= to 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.

```
315 FOR V=10*n to 3.4/Q STEP SQR(R)
```

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.

```
320 FOR V=9 TO 1 STEP -1
```

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 9 times.

```
330 FOR W=1 to 10: FOR W=1 TO :NEXT W: NEXT W
```

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 D)

GOTO

```
50 GOTO 100
```

Branches to the statement specified.

GOSUB

```
10 GOSUB 910
```

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. (See Appendix D)

IF...GOTO

```
32 IF X<=Y+5 GOTO 40
```

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.

IF....THEN

```
        IF X<10 THEN 5
```
Branches to a specified statement if the relation is true.

```
        20 IF X<0 THEN PRINT "X LESS THAN 0"
```
Executes all of the statements on the remainder of the line after the THEN if the rlation is True.

```
        25 IF X=5 THEN 50:Z=A
```
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 the line after line 25.

```
        26 IF X<0 THEN PRINT "ERROR, X NEGATIVE": GOTO 350
```
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.

INPUT

```
        3 INPUT V,W,W2
```
Requests data from the terminal (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 typed as a prompt character. Only constants may be typed in as a response to an INPUT statement, such as 4.53E-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 extra data will be ignored and the warning "EXTRA IGNORED" will be printed. Strings must be input in the same format as they are specified in DATA statements.

```
        5 INPUT "VALUE";V
```
Optionally types a prompt string ("VALUE") before requesting data from the terminal. If a carriage return is typed to an input statement, BASIC returns to the command mode. Typing CONT after an INPUT command has been interrupted will cause the program to restart where it left off at the INPUT statement.

LET

```
        300 LET W=X
        310 V=5.1
```
Assigns a value to a variable. The "LET" is optional.

```
NEXT     340 NEXT V      Marks the end of a FOR loop.
         345 NEXT        If no variable is given, matches the most
                         recent FOR loop.
         350 NEXT V,W    A single NEXT may be used to match multiple
                         FOR statements.  Equivalent to NEXT V:NEXT W.

ON....GOTO
      100 ON I GOTØ 10,20,30          Branches to the line indicated
                                      by the I'th number after the
                                      GOTO.  That is:
                          IF I=1, THEN GOTO LINE 1Ø
                          IF I=2, THEN GOTO LINE 2Ø
                          IF I=3, THEN GOTO LINE 3Ø

                          If I=Ø or I attempts to select a non-existent
                          line (>=4 in this case), the statement after
                          then ON statement is executed.  However, if I
                          is >255 or <Ø, an FC error message will result.
                          As many line numbers as will fit on a line can
                          follow an ON....GOTO.

ON....GOSUB
     110 ON I GOSUB 50,60       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.

POKE     357 POKE I,J     The POKE statement stores the byte specified
                          by its second argument (J) into the location
                          given by its first argument (I), where I is the
                          decimal equivalent of the memory location.  The
                          byte to be stored must be =>Ø and <=255, or an
                          FC error will occur.  The address (I) must be
                          =>Ø and <=65535, or an FC error will result.
                          Careless use of teh POKE statement will probably
                          cause you to POKE BASIC to death; that is, the
                          machine will hang, and you will have to reload
                          BASIC and will lose any program you had typed
                          in.  A POKE to a non-existent memory location
                          is harmless.  One of the main uses of POKE is
                          to pass arguments to machine language sub-
                          routines.  You could also use PEEK and POKE to
                          write a memory diagnostic or an assembler in
                          BASIC.

PRINT    360 PRINT X,Y;Z  Prints the value of expressions on the terminal.
         370 PRINT        If the list of values to be printed out does
         380 PRINT X,Y;   not end with a comma (,) or a semicolon (;),
         390 PRINT "VALUE IS";A    then a carriage return/line feed is
         400 PRINT A2,B,           executed after all the values have
                                   been printed.  Strings enclosed in
                          quotes (") may also be printed.  If a semi-
                          colon 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 56 or more, then a carriage return/line feed is executed. If the print head is before print position 56, then spaces are printed until the carriage is at the beginning of the next 14 column field (until the carriage is at column 14, 28, 42 or 56...). If there is no list of expressions to be printed, as in line 370 of the examples, then a carriage return/line feed is executed.

```
410 PRINT MID$(A$,2)
```
String expressions may be printed.

READ

```
490 READ V,W
```
Reads 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 of data 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. The line number given in the error statement will refer to the line number where the error actually is located.

REM

```
500 REM NOW SET V=0
```
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 ":".

```
505 REM SET V=0:V=0
```
In this case the V=0 will never be executed by BASIC.

```
506 V=0:REM SET V=0
```
In this case V=0 will be executed.

RESTORE

```
510 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.

RETURN

```
50 RETURN
```
Causes a subroutine to return to the statement after the most recently executed GOSUB. A GOSUB statement without an accompanying RETURN statement in the subroutine that the program branched to as a result of the GOSUB, will cause the output of an error statement.

STOP 9000 STOP

Causes a program to stop execution and to enter command mode.
Prints BREAK IN LINE 9000 (in this example)
Enter a CONT command after a STOP and the program branches to the statement following the STOP.

WAIT 805 WAIT I,J,K

This statement reads the status of location I, exclusive OR's K with the status, and then AND's the result with J until a non-zero result is obtained. Execution of the program continues at the statement following the wait statement. If the WAIT statement only has two arguments, K is assumed to be zero. If you are waiting for a bit to become zero, there should be a one in the corresponding position of K. I, J, and K must be =>0 and <=255.

INTRINSIC FUNCTIONS

ABS(X) 120 PRINT ABS(X)

Gives the absolute value of the expression X. ABS returns X if X>=0, -X otherwise.

INT(X) 140 PRINT INT(X)

Returns the largest integer less than or equal to its argument 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
```

If D=2, the above would always round to two decimal places for dollars and cents.

RND(X) 170 PRINT RND(X)

Generates a random number between 0 and 1. The argument X controls the generation of the 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 & B.

SGN(X) 230 PRINT SGN(X)

Gives 1 if X>0, 0 if X=0, and -1 if X<0.

SIN(X) 190 PRINT SIN(X)

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).

SQR(X)	180 PRINT SQR(X)	Gives the square root of the argument X. An FC error will occur if X is less than zero.
TAB(I)	240 PRINT TAB(I)	Spaces to the specified print position (column) on the terminal. May be used only in PRINT statements. Zero is the leftmost column on the terminal, 71 the rightmost. If the carriage is beyond position I, then no printing is done. I must be =>0 and <=255.
USR(I)	200 PRINT USR(I)	Calls the user's machine language subroutine. See POKE, PEEK, and APPENDIX J.
ATN(X)	210 PRINT ATN(X)	Gives the arctangent of the argument X. The result is returned in radians and ranges from -PI/2 to PI/2. (PI/2=1.5708)
COS(X)	200 PRINT COS(X)	Gives the cosine of the expression X. X is interpreted as being in radians.
EXP(X)	150 PRINT EXP(X)	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 87.3365
FRE(X)	270 PRINT FRE(0)	Gives the number of memory bytes currently unused by BASIC. Memory allocated for STRING space is not included in the count returned by FRE.
LOG(X)	160 PRINT LOG(X)	Gives the natural (Base E) logarithm of its argument 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).
PEEK	356 PRINT PEEK(I)	The PEEK function returns the contents of memory address I. The value returned will be =>0 and <=255. If I is >65535 or <0, an FC error will occur. Remember that I is the decimal (not HEX) address of the memory! An attempt to read a non-existent memory address will return an unkown value. (See POKE statement)
POS(I)	260 PRINT POS(I)	Gives the current position of the terminal print head (or cursor on CRT's). The leftmost character position on the terminal is position zero and the rightmost is 71.
SPC(I)	250 PRINT SPC(I)	Prints I space (or blank) characters on the terminal. May be used only in a PRINT statement. X must be =>0 and <=255.

TAN(X)	200 PRINT TAN(X)	Gives the tangent of the expression X. X is interpreted as being in radians.

STRINGS

1) A string may be from 0 to 255 characters in length. All string variables 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 eleven 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.

3) The total number of characters in use in strings at any time during program execution cannot exceed the amount of string space, or an OS error will result. AT initialization, you should set up string space so that it can contain the maximum number of characters which can be used by strings at any one time during program execution.

DIM	25 DIM A$(10,10)	Allocates space for a pointer and length for each element of a string matrix. No string space is allocated. See Appendix D.
LET	27 LET A$="HI"+V$	Assigns the value of a string expression to a string variable. LET is optional.
= > > <= >= <>		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 user's terminal. String does not have to be quoted; but if not leading blanks will be ignored and the string will be terminated on a "," (comma) or ":" (colon) 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 or end of line and leading spaces are ignored. See DATA

PRINT	60 PRINT X$ 70 PRINT "HI"+A$	Prints the string expression on the user's terminal.
ASC(X$)	30 PRINT ASC(X$)	Returns the ASCII numeric value of the first character of the string expression X$. See Appendix K for an ASCII/number conversion table. An FC error will result if X$ is a null string.
CHR$(I)	27 PRINT CHR$(I)	Returns a one character string whose single character is the ASCII equivalent of the value of the argument (I) which must be =>0 and <=256. See Appendix K.
LEFT$(X$,I)	310 PRINT LEFT$(X$,I)	Gives the leftmost I characters of the string expression X$. If I<=0 or >255 an FC error occurs.
LEN(X$)	22 PRINT LEN(X$)	Gives the length of the string expression X$ in characters (bytes). Non-printing characters and blanks are counted as part of the length.
MID$(X$,I)	330 PRINT MID$(X$,I)	MID$ called with two arguments returns characters from the string extpression 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.
MID$(X$,I,J)	330 PRINT MID$(X$,I,J)	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.
RIGHT$(X$,I)	32 PRINT RIGHT$(X$,I)	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$.
STR$(X)	29 PRINT STR$(X)	Gives a string which is the character representation of the numeric expression X. For instance, STR$(3.1)=" 3.1".
VAL(X$)	30 PRINT VAL(X$)	Returns the string expression X$ converted to a number. For instance, VAL("3.1")=3.1. If the first non-space character is not a "+" or "-" or a digit or a decimal point, then zero will be returned.

SPECIAL CHARACTERS

@	Erases current line being typed, and types a carriage return/line feed.
CTRL/H	(CTRL and H key are pressed together) Backspace to erase last character typed. If no more characters are left on the line, types a carriage return/line feed.
CTRL/L CTRL/S	Special characters in the MICRO-Z version of BASIC to permit the user to load and save numeric data computed in programs, in the same manner that LOAD and SAVE are used to store and reload user programs. See Appendix G for full instructions.
CARRIAGE RETURN	A carriage return must end every line typed in. Returns print head or CRT cursor to the first position (leftmost) on a line. A line feed is always executed automatically after a carriage return.
: (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

MISCELLANEOUS

1) To read in a paper tape with a program on it, type a CTRL/O and feed in tape. There will be no printing as the tape is read in. Type CTRL/O again when the tape is through. Alternatively, set NULLS=0 and feed in the paper tape, and when done reset nulls to the appropriate setting for your terminal.

2) To leave BASIC with the KIM computer, simply press the "ST" button on the KIM. If you have initialized the KIM system as described in Appendix A, and have set the addresses 17FA through 17FF to 1C00, the computer will go to the KIM monitor. Any time that you wish to reenter BASIC, simply press the space bar on the terminal several times to get 0000 4C and then press "G". BASIC will be re-entered through the WARM START location and if you had previously loaded a program, the program will still be intact. Enter RUN or LIST to use the program per normal procedure.

3) The maximum line length is 72 characters. If you attempt to type too many characters, a bell (ASCII 7) is executed, and the character you typed in will not be inserted in the program.

GET

```
190 GET A$
200 IF A$="Y" THEN 300
```

The GET command permits the operator to input simple variables without the necessity for hitting the "RETURN" key. After a character is typed, the program goes immediately to the next step. The GET command is useful as a waiting point in your program - to hold the operation until a key is pressed - for allowing time to read text, etc.

```
250 GET B
260 PRINT
270 GET C
280 PRINT B+C
```

The GET command can also be used to input simple, single digit variables. Again, the program moves immediately to the next step after the variable is typed, without the operator pressing the "RETURN" key.

%

```
300 INPUT A
310 A%=A
320 PRINT A%
330 INPUT CD%
```

The percent "%" sign, when appended to a variable, will only store an integer in the variable. In other words, AB% is the equivalent of INT(AB).

APPENDICES

APPENDIX A

Starting up your KIM BASIC.

1. Load in the Basic tape using the standard KIM-1 procedure:

   a. 17F9 01 (enters tape ID number)
      17FA 00
      17FB 1C
      17FE 00
      17FF 1C

      00F1 00
      00F2 FF

   b. 1873 A9 - - - press "G" on terminal

   c. Start tape playback

   d. BASIC will load from 2000 (H) to 44B0 (H)

2. Go to "Cold Start" location 4065 - and press "G" on terminal.

3. Read Appendix B - "Initialization Dialogue".

4. After initializing, you can now write and run your programs. If you have set the NMI and IRQ vectors as shown in step 1a, pressing the "ST" key on your KIM will take you out of BASIC and there are two ways to re-enter without erasing the program written in BASIC that you may be running.

   a. Key in the "WARM-START" address (2348 H), and press G.

or

   b. Simply press the space-bar on your terminal to get address 0000 and then press "G". (Address 0000 provides a jump-4C back to the BASIC warm start automatically.)

5. As discussed earlier in the manual, you can record and reload programs that you have written in BASIC, using the "SAVE" and "LOAD" instructions. (Appendix G discusses in detail how to use the CTRL/L and CTRL/S instructions to save numerical data.)

As supplied by Micro-Z, the BASIC program uses the hypertape tape load routines to load tape. Hypertape is included as part of BASIC.

To SAVE a program that you have written in BASIC -

   a. Type SAVE on the terminal - don't press RETURN yet.

   b. Start the tape recorder (in the RECORD mode).

c. Press "RETURN" after the tape is up to full speed.

If the SAVE is accomplished properly, the program goes to location 0000 per normal KIM practice. Pressing the "G" on the terminal will jump the KIM back into BASIC using a special entry point. The word "SAVED" will appear, and the program will be fully operational.

6. To LOAD a program that was previously "SAVE'd" -

a. Load and initialize BASIC.

b. Type "LOAD" and press RETURN.

c. Start the tape recorder (in playback).

After the tape is loaded properly, the terminal goes back to 0000, again per normal KIM practice. Simply press "G" and you are back in BASIC, the print-out "LOADED" will appear, and your program is ready to "LIST" or "RUN".

NOTE:

After the tape is loaded or saved, and a 0000 4C appears, pressing a "G" does not jump to the warm start location - but to special "LOAD" and "SAVE" entry points. Also note that after the reentry is complete, the address in 0000, 0001, and 0002 is changed automatically back to the warm start address - so that anytime that you leave BASIC, pressing the SPACE BAR to get address 0000 and then pressing "G" always provides the correct entry back into BASIC after it has once been initialized.

7. RECORDING BASIC IN HYPERTAPE

All programs are recorded in HYPERTAPE, which permits recording and playback at about 6 times the normal KIM speed. The HYPERTAPE routines are included in the BASIC program and were loaded into memory when you loaded BASIC.

BASIC itself can be recorded using HYPERTAPE. This will permit loading the full BASIC into KIM in about 3 minutes. It is recommended that BASIC be re-recorded on a separate cassette, using HYPERTAPE, and the original cassette be stored in a safe, remote place to prevent accidental erasure.

To record BASIC in HYPERTAPE, use the following procedure:

a. Load BASIC fresh into the KIM

NOTE: IT IS VERY IMPORTANT THAT BASIC BE LOADED INTO KIM FRESH BEFORE

IT IS RECORDED ON TAPE. DO NOT INITIALIZE - DO NOT GO TO THE WARM OR COLD START LOCATION!!

b. Load the following:

```
17F5  FF
17F6  1F
17F7  B0
17F8  44
17F9  01    (recommended ID number)
00F1  00
00F2  FF
```

c. Go to the starting address of Hypertape - 42B8 (do not press "G" yet)

d. With a fresh tape - start the recorder (on record)

e. After the recorder is started and the tape leader is clear - press "G" to start the data flow.

9. To reload BASIC into the KIM, use the same procedure as outlined in item 1 above. Be sure to use the proper ID number.

10. Important addresses in KIM-BASIC (Hex)

Cold Start Entry (to start BASIC)	4065
Warm Start Entry (preserves program)	2348
Exit to Monitor Print-Out	2A51
Entry from Keyboard	2456
Jump to Record Tape (programs)	2782
Jump to Record Tape (data & strings)	43F3
Jump to Load Programs	27A3
Jump to Load Data and Strings	4441
Entry after Load Data and Strings	4446
Entry after Record Tape (Data)	43F8
Entry after Record Tape (programs)	2786
Entry after Load Programs	27A6

MICROSOFT KIM BASIC - PAGE ZERO USE

00-02	Jump to re-enter BASIC
03-14	BASIC scratch pad
15	Number of Nulls
16	Current Terminal Column
17	Line length
18	Position of Last comma field
1B	Input buffer - 72 decimal bytes
78/79	Pointer to start of user programs
7A/7B	Pointer to start of simple variable table
7C/7D	pointer to start of array table
7E/7F	First location unused by array table
80/81	Lowest location used by string data
84/85	Highest memory location used by BASIC
86	Current line number
AE	Floating accumulator
CD	Routine to read character from current position
D8	Current Random Number
DD	First unused page 0 location
FF	Used by STR $ function

APPENDIX B

INITIALIZATION

STARTING BASIC

After you have loaded BASIC, typed 4065 (HEX) and pressed "G", BASIC will respond:

MEMORY SIZE?

If you type a carriage return in answer to MEMORY SIZE?, BASIC will use all the contiguous memory upwards from the top of BASIC that it can find. It will stop searching when it finds one byte of ROM or non-existent memory.

If you wish to allocate only part of the KIM memory to BASIC, type the highest address you wish to allocate - in decimal, not in hex! This might be done, for instance, if you were using part of the memory for a machine language subroutine. The top of KIM BASIC is 44B0 (Hex). the decimal equivalent of this is 17584. If you allocate less than this number, an insufficient memory indication will result.

BASIC will then ask:

TERMINAL WIDTH? This is to set the output line width for PRINT statements only. Type in the number of characters for the line width for the particular terminal or other output device you are using. This may be any number from 1 to 255, depending on the terminal. If no answer is given (i.e. a carriage return is typed) the line width is set to 72 characters.

Now BASIC will type out:

XXXX BYTES FREE

"XXXX" is the number of bytes available for program, variables, matrix storage, and the stack. After the copyright notice, the terminal will type:

OK

and you are ready to use your KIM Microsoft BASIC.

APPENDIX C

ERROR MESSAGES

After an error occurs, BASIC returns to command level and types OK. 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 occurred for the indirect statement.

The following are the possible error codes and their meanings:

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).
DD	Double Dimension. After a matrix was dimensioned, another dimension 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:

a) a negative matrix subscript (LET A(-1)=0)

b) an unreasonably large matrix subscript (>32767)

c) LOG-negative or zero argument

d) SQR-negative argument

e) A^B with A negative and B not an integer

f) a call to USR before the address of the machine language subroutine has been patched in.

g) 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 or DEF FN statement as a direct command.

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 th above.

OV Overflow. the result of a calculation was too large to be represented in BASIC's number format. If an underflow occurs, zero is given as the result and execution continues without any error message being printed.

SN Syntax Error. Missing parenthesis in an expression, illegal chracter in a line, incorrect punctuation, etc.

RG RETURN without GOSUB. A RETURN statement was encountered without a previous GOSUB statement being executed.

US Undefined Statement. An attmpt was made to GOTO, GOSUB or THEN to a statement which does not exist.

/Ø Division by Zero.

CN Continue Error. Attempt to continue a program when none exists, an error occurred, or after a new line was typed into the program.

LS Long String. String more than 255 characters long.

ST String Temporaries. String too complex - break it up.

TM Type Mismatch. Left hand side of assignment statement was numeric variable and right hand side was string, or vice versa; or a function which expected a string argument was given a numeric one, or vice versa.

UF Undefined Function. User defined function not defined.

APPENDIX D

SPACE HINTS

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 small amount of overhead (5bytes) 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 65529), 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 of bytes in the comment text. For instance, the statement 130 REM THIS IS A COMMENT uses up 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 the same 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.

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

STORAGE ALLOCATION INFORMATION

Simple (non-matrix) numeric variables like V use 6 bytes; 2 for the variable name, and 4 for the value. Simple non-matrix string variables also use 6 bytes; 2 for the variable name, 2 for the length, and 2 for a pointer.

Matrix variables use a minimum of 12 bytes. Two bytes are used for the variable name, two for the size of the matrix, two for the number of dimensions and two for each dimension along with four bytes for each of the matrix elements.

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, 6 bytes are used to store the definition.

Reserved words such as FOR, GOTO or NOT, and the names or 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.

APPENDIX E

SPEED HINTS

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 BY A FACTOR OF 10.* 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) NEXT statements without the index variable. NEXT is somewhat faster than NEXT I because no check is made to see if the variable specified in the NEXT is the same as the variable in the most recent FOR statement.

APPENDIX F

DERIVED FUNCTIONS

The following functions, while not intrinsic to ALTAIR 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
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((SQR(-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

DATA/SAVE COMMANDS

NOTE:

Your version of Microsoft BASIC has been updated with the latest improvements, as of 1 November 1978. Specifically, the DATA/SAVE commands have been expanded to included the capability for storing of both data and strings from the programs that you are running. Read Appendix "G" for full instructions on how to utilize these commands.

The program below is written to assist in demonstrating this added capability. Use this program in place of the program listed in Appendix "G" to become familiar with the operation of the DATA/SAVE commands.

```
LIST

 2 PRINT:PRINT
 5 DIM A(5,5)
 7 X=20:REM WE ARE SETTING UP 20 STRINGS
 8 DIM A$(X)
 9 PRINT:PRINT
 10 INPUT "CTRL/L TO LOAD DATA";ZZ
 15 PRINT
 20 PRINT "PRIOR CUMULATIVE DATA";A(2,2)
 25 PRINT
 30 INPUT "NEW DATA";C
 35 PRINT:A(2,2)=A(2,2)+C
 40 PRINT:PRINT "NEW DATA";C,"CUMULATIVE DATA";A(2,2)
 50 PRINT:PRINT
 70 INPUT "INPUT STRING NUMBER";N
 75 PRINT "TYPE OUT STRING"
 76 PRINT:INPUT A$(N)
 78 PRINT
 80 FOR C=0 TO X
 90 PRINT C;A$(C)
 100 NEXT
 110 INPUT "CTRL/S TO SAVE DATA";ZZ
 120 PRINT
 130 PRINT "DEMO AT END"
OK
```

If you are running a program on a continuous basis where you are using the DATA/SAVE commands to save "strings" - each time that you reload BASIC into the computer to run that program, when you initialize BASIC, be sure to answer the "MEMORY SIZE" question in the same manner as when the program was first recorded. This is required because strings are stored at the very top end of RAM, and the DATA/SAVE program will put the strings back at the same address as when the strings were first recorded.

APPENDIX G

DATA SAVE/DATA LOAD

A unique feature of the Micro-Z version of BASIC is the ability to save and reload numerical data that is computed and stored as a variable when a program is run in BASIC.

This is accomplished by utilizing two commands, CTRL/S and CTRL/L (the CTRL/ indicates that the "CONTROL" key and the "S" or the "L" key are pressed simultaneously). These commands operate in conjunction with the INPUT statements to permit the operator to leave and re-enter the program as it is running in BASIC.

NOTE: If you are loading and saving data from a program written in BASIC on a continuing basis, such as a payroll or ledger account program, any changes in the program could destroy the ability to reload data.

WRITING THE PROGRAM:

Load BASIC and write the following demonstration program:

```
2 PRINT:PRINT
5 DIM A(5,5)
10 INPUT "CTRL/L TO LOAD PRIOR DATA";ZZ
15 PRINT
20 PRINT "PRIOR CUMULATIVE DATA";A(2,2)
25 PRINT
30 INPUT "NEW DATA";C
35 PRINT:A(2,2)=A(2,2)+C
40 PRINT:PRINT "NEW DATA";C,"CUMULATIVE DATA";A(2,2)
45 PRINT
50 INPUT "CTRL/S TO SAVE DATA";ZZ
55 PRINT
60 PRINT "PROGRAM END"
65 REM-----------------THESE SPACES ALLOW FOR CHANGES---
66 REM-----------------------------------------------
67 REM-----------------------------------------------
```

Note in particular lines 10 and 50. These are the key instructions that permit the loading and saving of data. The variable "ZZ" is a dummy and should not be used elsewhere in the program.

Line 10 is the prompt to load data back in the program that was previously saved. All DIM and READ statements should be inserted in the program prior to this command. Line 50 is the prompt to save data on tape, and should be placed at the end of the program after all the computations are complete. In between these two lines (10 and 50) are the primary working commands in the program. The REM statements

after line 60 are simply dummy inputs to allow for minor program changes after the program is operational. If you add commands to the program at a later date - these REM statements will absorb the changes and prevent losing actual program commands and stored data. They are only needed in programs where you wish to save data in the future.

Obviously, in your programs - lines 10 and 50 will have different numbers, but their relative position (near the beginning and end of the program) will stay the same. In addition, the prompting phrases inside the quotation marks can be changed to say anything you wish.

RUNNING THE PROGRAM - (FIRST TIME)

a. Type "RUN" and press RETURN.

b. The first prompt statement will appear:

   CTRL/L TO LOAD PRIOR DATA?

c. Since there is no prior data to load, type any number and press RETURN. This number is entered into the dummy variable (ZZ) and the program progresses to the next step.

   PRIOR CUMULATIVE DATA 0

   NEW DATA?

d. The main part of the program begins at this point. type a number you wish to insert (i.e. 11.56) and press RETURN.

e. The program continues to the next steps:

   NEW DATA 11.56 CUMULATIVE DATA 11.56

   CTRL/S TO SAVE DATA?

f. Start the tape recorder (in record mode) - and when the tape is up to speed and the tape leader is clear - type CTRL/S. The word "SAVE" will appear and when the data is stored in the cassette, a 0000 4C will appear.

g. Stop the recorder, rewind, and press "G" and the program will continue to the end.

   PROGRAM END

NOTE: If you do not wish to save data when the prompt -

   CTRL/S TO SAVE DATA?

appears, simply type any number and press RETURN. The program will go immediately to the next step.

RUNNING THE PROGRAM - (AFTER SAVING DATA)

When you type "RUN" and press RETURN, normally all of the data computed during any prior "RUN" is erased and any data must be re-entered manually. However, we have stored the data on tape and will now re-enter it into the program.

a. Type "RUN" and press RETURN.

b. The first prompt statement will appear:

   CTRL/L TO LOAD PRIOR DATA?

c. Again, if we don't wish to load data - type any number (1) and press RETURN.

d. To load data - press CTRL/L and RETURN, then start the tape recorder (in playback mode). The word "LOAD" will appear - and when the data is loaded into memory, a 0000 4C will appear.

e. Stop the recorder, rewind, and press "G".

f. The data is now loaded into the computer memory. Even though a month may have elapsed since the first time you ran the program and saved the data, it is as if line 50 was replaced with a "GOTO 20" when you first ran the program. The prompt for new data will appear:

   PRIOR CUMULATIVE DATA 11.56

   NEW DATA?

g. Type in a test number (i.e. 12.31) and press RETURN

   NEW DATA 12.31 CUMULATIVE DATA 23.87

   CTRL/S TO SAVE DATA?

h. The cycle is now complete and you can save the new value if you desire.

You will note that the old cumulative data was 11.56, and is now updated to 11.56 + 12.31 = 23.87 .

The above process could continue over and over, as often as you desire.

CHANGING THE CTRL/S AND CTRL/L COMMANDS

There may be difficulty with some terminals that utilize the CTRL/S or CTRL/L commands for other functions on the CRT or hardcopy terminal. These commands can be changed easily to any other CTRL/- character you may desire.

The location of the two commands are:

CTRL/S	4262	13	(13 is ASCII CTRL/S)
CTRL/L	245A	0C	(0C is ASCII CTRL/L)

For instance, to change the CTRL/S (S for SAVE) command to CTRL/R (R for RECORD) go to location 4262 Hex and replace the 13 with a 12. 12 is the ASCII code for a CTRL/R.

In the same fashion, CTRL/L (L for LOAD) could be changed to CTRL/P (P for PLAYBACK) by changing location 245A Hex from a 0C to a 10 since 10 is ASCII code for CTRL/P.

APPENDIX H

CONVERTING BASIC PROGRAMS NOT WRITTEN FOR THE KIM

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

1) Matrix subscripts. Some BASICs use " [" and "] " to denote matrix subscripts. KIM 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$(I,J) declares a string matrix of J elements each of which has a length I. Convert DIM statements of this type to equivalent ones in KIM BASIC: DIM A$(J).

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

KIM BASIC uses LEFT$, RIGHT$ and MID$ to take substrings of strings. Other BASICs use 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	NEW
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	NEW
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 KIM 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 the " \'s " to " :'s " in the program.

5) Paper tapes punched by other BASICs may have no nulls at the end of each line, instead of the three per line recommended for use with KIM BASIC.

To get around this, try to use the tape feed control on the Teletype to stop the tape from reading as soon as KIM BASIC types a carriage return at the end of the line. Wait a second, and then continue feeding in the tape.

When you have finished reading in the paper tape of the program, be sure to punch a new tape in KIM BASIC's format. This will save you from having to repeat this process a second time.

6) 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.

APPENDIX I

USING THE CASSETTE INTERFACE

Your BASIC program includes the listing for KIM Hypertape to permit you to record and playback programs and data using the faster speed. The recording speed is approximately six times the normal KIM speed.

When you wish to store a program you have written in BASIC, first type SAVE, start your recorder in the "RECORD" mode, then type "RETURN". This assures that the data doesn't start to flow to the recorder until it is up-to-speed and the tape "leader" has passed the record head. When the program is completely recorded on tape, the KIM terminal will display a 0000 4C. This indicates that the "SAVE" has been accomplished successfully.

If you now type a "G", the KIM will reenter BASIC and the word "SAVED" will appear. "SAVING" a program does not alter it in any way. You can RUN your program in BASIC before or after you SAVE it on tape. The SAVE command automatically goes to the Hypertape routine to record your program.

To LOAD a program written in BASIC that you have previously "SAVED", first load and initialize BASIC. Then, type "LOAD" and press "RETURN". This assures that the program is ready to receive the first data as it comes from the recorder. Start the tape recorder in the "PLAYBACK" mode. When a 0000 4C appears on the terminal, it indicates that the playback is successful. Simply press "G" and the program reenters BASIC - and your program is ready to "RUN" or "LIST". When you LOAD a program from the cassette, a "NEW" command is executed first and any other programs loaded in BASIC are removed from memory.

BASIC is shipped on a cassette that was recorded using the standard KIM speed. Hypertape was not used since we have found it to be unreliable in playback if the same machine was not used for the original recording. As a result, the BASIC program provided takes approximately 18.5 to 19 minutes to load. However, as indicated in APPENDIX A, you can use the Hypertape routine in BASIC to re-record BASIC at the faster speed on another cassette.

It is very important that if you do re-record BASIC, record it on a fresh cassette BEFORE YOU INITIALIZE. In other words, load BASIC, enter your start and stop addresses, start your recorder with a fresh cassette, and go to Hypertape. Then, after it is recorded, you can go to 4065 and press "G" to initialize your BASIC if you wish. The reason for following this procedure is that certain addresses in BASIC change after initializing. Read APPENDIX A fully for complete instructions on LOAD, SAVE, and re-recording BASIC in Hypertape.

APPENDIX J

BASIC/MACHINE LANGUAGE INTERFACE

There are four steps required to use a machine language routine:

1) Set aside memory. The KIM version of Microsoft 6502 BASIC starts at 2000 Hex. Contiguous memory above BASIC is used for program storage (above 44B0 Hex). The highest location used by BASIC is determined by the response to the "MEMORY SIZE?" question. If a decimal number is typed in, that will be the highest location in memory used by BASIC. Otherwise, BASIC will search for the highest contiguous RAM address by storing and reading a "24" in memory.

A machine language routine must not be in a memory area used by BASIC, so it must -

a) Below 2000 Hex, but above 0200 Hex - or
b) Above the decimal address typed into the answer to "MEMORY SIZE?" - or
c) In some higher RAM that is not contiguous with the RAM starting at 2000 HEX.

2) Store the routine into memory. This can be done either before or after BASIC is loaded. To enter your machine language program you can use the KIM cassette, utilize an assembler, key the routine directly into memory, or you can use BASIC itself and a series of "POKE" commands. End the routine with an RTS (60) command.

3) Notify BASIC of the location of the routine. USRLOC, which is 2040 Hex, must be changed to contain the starting address of the "USR" machine language routine. This may be done using the "POKE" function in your BASIC program. Enter the low 8-bits of the address in 2040 Hex, and enter the high 8-bits of the address in 2041 Hex.

Invoking the "USR" function before modifying the USRLOC will cause an "ILLEGAL QUANTITY" error - since the original contents of USERLOC contains the address of the "ILLEGAL QUANTITY" error routine.

4) The machine language routine must be called in your program. The "USR" function is used for this purpose. A single numeric value must be given as the argument to the USR command. BASIC will jump immediately to the address in USRLOC and perform that series of commands. As indicated earlier, the routine should end in an RTS (60) command. If this is done, when the machine program is complete - BASIC will resume with the next command following the USR command.

Data can be passed to the machine language program by using the POKE command to store data in a memory location in RAM. The machine language program simply reads this data as needed. Remember that the POKE command uses the DECIMAL value of the memory address, while the machine language program will use the HEX address.

In the same fashion, data can be passed back to BASIC. The machine program can store the program results in a known RAM memory location. The BASIC program can "PEEK" this location and read the data from the machine program.

Note that in both cases above, the memory location where data is stored must not be any memory used by BASIC. The same rules as in item 1) above apply.

NOTES ON PEEK AND POKE

As mentioned before, POKE can be used to set up your machine language program in high memory. BASIC does not restrict which addresses you can POKE. Modifiying USRLOC can be accomplished using two successive calls to POKE. Patches which a user wishes to include in his BASIC can also be made using POKE.

Using the PEEK function and POKE statement, the user can write a binary dump program in BASIC. Using PEEK and POKE it is possible to write a binary loader.

PEEK and POKE can be used to store byte oriented information. When you initialize BASIC, answer the MEMORY SIZE? question with the amount of memory in your KIM minus the amount of memory you wish to use as storage for byte formatted data.

You are now free to use the memory in the top of RAM in your KIM as byte storage. See PEEK and POKE in the Reference Material for a further description of their parameters.

APPENDIX K

ASCII CHARACTER CODES

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

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

CHR$ is a string function which returns a one character string which contains the ASCII equivalent of the argument, according to the conversion table on the preceeding page. 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 the user's terminal. The most often used of these characters is the BEL (ASCII 7). Printing this character will cause a bell to ring on some terminals and a "beep" on many CRT's. This may be used as a preface to an error message, as a novelty, or just to wake up the user if he has fallen asleep. (Example: PRINT CHR$(7);)

A major use of special characters is on those CRT's that have cursor positioning and other special functions (such as turning on a hard copy printer).

As an example, try sending a form feed (CHR$(12)) to your CRT. On most CRT's this will usually cause the screen to erase and the cursor to "home" or move to the upper left corner.

Some CRT's give the user the capability of drawing graphs and curves in a special point-plotter mode. This feature may easily be taken advantage of through use of KIM BASIC's CHR$ function.

APPENDIX M

BASIC TEXTS

Below are a few of the many texts that may be helpful in learning BASIC.

1) BASIC PROGRAMMING, John G. Kemeny, Thomas E Kurtz, 1967, p145

2) BASIC, Albrecht, Finkel and Brown, 1973

3) A GUIDED TOUR OF COMPUTER PROGRAMMING IN BASIC, Thomas A Dwyer and Michael S. Kaufman; Boston: Houghton Mifflin Co., 1973

Books numbered 1 & 2 may be obtained from:

People's Computer Company
P.O. Box 310
Menlo Park, California
94025

They also have other books of interest, such as:

101 BASIC GAMES, Ed. David Ahl, 1974 p250

WHAT TO DO AFTER YOU HIT RETURN or PCC's FIRST BOOK OF COMPUTER GAMES

COMPUTER LIB & DREAM MACHINES, Theodore H. Nelson, 1974, p186