

# **BASIC REFERENCE MANUAL**

Copyright © by Synertek Systems Corporation

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, electronic, mechanical, photocopying, recording or otherwise, without the prior written consent of Synertek Systems Corporation.

SSC Pub MAN-A-260026 - A

First Printing: December, 1978



**Synertek Systems Corporation**

P.O. BOX 552 SANTA CLARA, CALIFORNIA 95052 TEL. 408 988-5600 TWX. 910-338-0135

TABLE OF CONTENTS

INTRODUCTION . . . . .	1
GETTING STARTED WITH BASIC . . . . .	1
REFERENCE MATERIAL . . . . .	23
<u>Commands</u> . . . . .	23
CLEAR . . . . .	23
LIST . . . . .	23
NULL . . . . .	23
RUN . . . . .	24
NEW . . . . .	24
CONT . . . . .	24
LOAD "A" . . . . .	24
SAVE "A". . . . .	24
<u>Operators</u> . . . . .	25
<u>Statements</u> . . . . .	29
DATA . . . . .	29
DEF . . . . .	29
DIM . . . . .	30
END . . . . .	30
FOR . . . . .	30
GOTO . . . . .	31
GOSUB . . . . .	31
IF...GOTO . . . . .	31
IF...THEN. . . . .	31
INPUT. . . . .	32
LET . . . . .	32
NEXT . . . . .	32
ON...GOTO . . . . .	32
ON...GOSUB . . . . .	33
POKE . . . . .	33
PRINT. . . . .	33
READ . . . . .	34
REM . . . . .	34
RESTORE. . . . .	34
RETURN . . . . .	34
STOP . . . . .	34
WAIT . . . . .	35
<u>Intrinsic Functions</u> . . . . .	35
ABS(X) . . . . .	35
INT(X) . . . . .	35
RND(X) . . . . .	35
SGN(X) . . . . .	35
SIN(X) optional . . . . .	35
SQR(X) . . . . .	35
TAB(I) . . . . .	35
USR(I) . . . . .	36

## TABLE OF CONTENTS (Continued)

<u>Intrinsic Functions (Continued)</u>	
USR(I,J,...Z) . . . . .	36
ATN(X) optional . . . . .	36
COS(X) optional . . . . .	36
EXP(X) . . . . .	36
FRE(X) . . . . .	36
LOG(X) . . . . .	36
PEEK . . . . .	36
POS(I) . . . . .	36
SPC(I) . . . . .	36
TAN(X) . . . . .	37
 <u>Strings</u> . . . . .	
DIM A\$ . . . . .	37
LET A\$ . . . . .	37
INPUT X\$ . . . . .	37
READ X\$ . . . . .	37
PRINT X\$ . . . . .	37
 <u>String Functions</u> . . . . .	
ASC (X\$) . . . . .	38
CHR\$(I) . . . . .	38
FRE(X\$) . . . . .	38
LEFT\$(X\$,I) . . . . .	38
LEN(X\$) . . . . .	38
MID\$(X\$,I) . . . . .	38
MID\$(X\$,I,J) . . . . .	38
RIGHT\$(X\$,I) . . . . .	38
STR\$(X) . . . . .	38
VAL(X\$) . . . . .	39
 <u>Special Characters</u> @, ←, CR, BREAK, :, C <sup>T</sup> , ?	
39	
 <u>Miscellaneous Comments</u> . . . . .	
39	
 <u>APPENDICES</u> . . . . .	
A-1	
A	INITIALIZATION DIALOG . . . . . A-1
B	ERROR MESSAGES. . . . . B-1
C	SPACE HINTS . . . . . C-1
D	SPEED HINTS . . . . . D-1
E	DERIVED FUNCTIONS . . . . . E-1
F	CONVERTING BASIC PROGRAMS NOT WRITTEN FOR SYNERTEK BASIC . . . . . F-1
G	BASIC/MACHINE LANGUAGE INTERFACE . . . . . G-1
H	ASCII CHARACTER CODES . . . . . H-1
I	BASIC TEXTS . . . . . I-1

## INTRODUCTION

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

Synertek 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 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 computer is "at your fingertips."

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

### GETTING STARTED WITH BASIC

You have received <sup>one</sup> ~~two~~ ROMs as your BASIC language. These ROMs are designed to run in your Synertek SYM-1 module using the SUPERMON monitor.

Insert the ROM marked <sup>02-0000A</sup> ~~02-0019-01~~ into socket U21 and ROM marked ~~02-0020-01~~ into ~~socket U22~~. Before applying power, the following on-board jumpers must be changed.

Remove the following  
Jumpers

~~B-2~~ B2  
~~C-2~~ C2  
~~F-5~~ F5  
~~G-5~~ G5  
~~K-11~~ K11  
~~L-12~~ L12

Add the following  
Jumpers

~~B-4~~ B4  
~~C-4~~ C4  
~~F-2~~ F2  
~~G-2~~ G2  
K-11 and 12  
L-13 and 14

B2  
C4  
F4  
G4

These jumper changes configure sockets U21 and U22 for 4Kx8 ROMs each and locate U21 at address C000-CFFF and U22 at address ~~D000-DFFF~~.

To run BASIC, first log on to SUPERMON.

Apply power to the SYM-1. Do not depress any keys on the hex keypad. Enter a "Q" from your Synertek KTM-2 or other RS-232 terminal device. (Do not use BAUD rates above 4800.)

If you are using a TTY, enter these keys on the hex keypad:

(SHIFT) (JUMP) (1) (CR)

A prompting period should now be displayed on your terminal. Enter (J) (Ø) (C/R) to start BASIC.

BASIC will respond with:

MEMORY SIZE? (type a carriage return)

BASIC will then ask:

TERMINAL WIDTH? (type a carriage return)

Now BASIC will type out:

XXXX BYTES FREE

BASIC VI.1  
COPYRIGHT 1978 SYNERTEK CORP.  
OK

Once your I/O device has typed " OK ", you are ready to use BASIC. For more detail on memory size and terminal widths refer to Appendix A.

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 I.

If you are already familiar with BASIC programming, the following section may be skipped. Turn directly to the Reference Material on page 23.

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.

#### NOTE

All commands to BASIC should 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 backarrow ( ← , shift/delete on KTM-2, usually Shift/0 on a TTY), or an underline to eliminate the last character. Repeated use of "←" 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 carriage return)
```

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 this:

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

BASIC will print:

```
.5      30
```

As you can see, BASIC can do division and multiplication as well as subtraction. Note how a " , " (comma) was used in the print command to print two values instead of just one. The 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 a " , " causes BASIC to skip to the next 14 column field on the terminal, where the value 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, 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
```

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
```

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
```

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
```

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
```

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
```

BASIC will reply with:

```
OK
```

Now type in:

```
LIST
```

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
```

BASIC will reply with:

```
ONE THIRD IS EQUAL TO          .33333333
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. A question mark is permitted as an abbreviation for PRINT.

Try the following examples:

A) PRINT "ONE THIRD IS EQUAL TO";1/3  
ONE THIRD IS EQUAL TO .33333333

OK

B) ?1,2,3  
1                    2                    3

OK

C) PRINT 1;2;3  
1 2 3

OK

D) ? -1;2;-3  
-1 2 -3

OK

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

The largest number that may be represented in BASIC is  $1.70141 \times 10^{38}$ , while the smallest positive number is  $2.93874 \times 10^{-39}$ .

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 0 to 99999999, 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 99999999, 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.XXXXXXXESTT . (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 seven 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 zeros are never printed. If there is only one digit to print after all trailing zeroes are suppressed, no decimal point is printed. The exponent sign will be " + " for positive and " - " for negative. Two digits of the exponent are always printed; that is zeroes are not suppressed in the exponent field. The value of any number expressed thus is the number to the left of the "E" times 10 raised to the power of the number to the right of the "E".

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

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

<u>NUMBER</u>	<u>OUTPUT FORMAT</u>
+1	1
-1	-1
6523	6523
-23.460	-23.46
1E20	1E+20
-12.3456E-7	-1.23456E-06
1.234567E-10	1.23457E-10
1000000	1E+06
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 1.2345678901234567890
1.2345679
```

OK

Additionally, a number input from the terminal or a numeric constant used in a BASIC program may be specified in HEX format. A HEX number consists of the ampersand character (&) followed by a four character string constant specifying the HEX number like so:  
&"FFFE".

```
PRINT &"OFF"  
255
```

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  $3.14159 \times 10 \times 10$ , or 314.159.

If you haven't already guessed, what the program above actually 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  
 28.2743  
? 4.7  
 69.3977  
?
```

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 statement (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.

Any alphanumeric characters after the first two are ignored. An alphanumeric character is any letter (A-Z) or any number (0-9). If the variable name ends with a "\$", it is a string variable; that is, it contains character information. If the name ends in "%", then the variable is an integer variable, and may contain only integer values. Numeric information is otherwise kept in floating point format internally.

Below are some examples of legal and illegal variable names:

<u>LEGAL</u>	<u>ILLEGAL</u>
A	% (1st character must be alphabetic)
Z1	QR (2nd character must be numeric)
TP	TO (variable names cannot be reserved
PSTG\$	words)
COUNT	RGOTO (variable names cannot contain reserved words)

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

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

```

273.  B0 A9 C0 BB C1 9A 9B B8 83 95 85 80
ABS AND ASC ATN CHR$ CLEAR CONT COS DATA DEF DIM END
87 9F 81 B2 8C 88 8A 84 AF C2 BD 87
EXP FN FOR FRE GOSUB GOTO IF INPUT INT LEFT$ LEN LET
99 93 B6 C4 9C 82 A2 91 90 AA BC 96
LIST LOAD LOG MID$ NEW NEXT NOT NULL ON OR PEEK POKE
B3 97 86 8E 8B 8D C3 B5 89 94
POS PRINT READ REM RESTORE RETURN RIGHT$ RND RUN SAVE
AE B9 A0 B4 A3 8F BE 9D BA A1 9E 81
SGN SIN SPC( SQR STEP STOP STR$ TAB( TAN THEN TO USR
BF 92 AD AB AC A4 A5 A6 A2
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 computer's 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) NEW is typed in

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

```
PRINT Q,Q+2,Q*2
      0    2    0
```

OK

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

This serves mainly as an aid for the programmer 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=0 THEN 50
30 PRINT "NON-ZERO"
40 GOTO 10
50 PRINT "ZERO"
60 GOTO 10
```

When this program is typed into the computer and run, it will ask for a value for B. Type any value you wish in. The computer 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:

<u>RELATION</u>	<u>MEANING</u>
=	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 0 was typed in for B the IF statement would be true because  $0=0$ . In this case, since the number after the THEN is 50, execution of the program would continue at line 50. Therefore, "ZERO" would be printed and then the program would jump back to line 10 (because of the GOTO statement in line 60).

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

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

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

Try running the last two programs several times. It may 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 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,SQR(2)
30 PRINT 3,SQR(3)
40 PRINT 4,SQR(4)
50 PRINT 5,SQR(5)
60 PRINT 6,SQR(6)
70 PRINT 7,SQR(7)
80 PRINT 8,SQR(8)
90 PRINT 9,SQR(9)
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 calculation is printed out.

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

At line 40, since N now equals 2, N<=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 one to be added to N, and then if  $N \leq 10$  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 they are 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>=1 THEN 20
```



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

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 Reference Material)

"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, we switch 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 "SORRY, 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, an RG error would occur. Each "GOSUB" executed in a program should 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) has 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". DOG, KUMQUAT, and THIS IS A TEST are all strings. Like numeric variables, string variables can be assigned specific values. String variables are distinguished from numeric variables by a "\$" after the variable name.

For example, try the following:

```
A$="SYNERTEK SYM-1"  
  
OK  
PRINT A$  
SYNERTEK SYM-1  
  
OK
```

In this example, we set the string variable A\$ to the string value "SYNERTEK SYM-1". 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("BITS")  
14      4  
  
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 0 to 255. A string which contains 0 characters is called the "NULL" string. Before a string variable is set to a value in the program, it is initialized to the null string. Printing a null string on the terminal will cause no characters to be printed, and the print head or cursor will not be advanced to the next column. Try the following:

```
PRINT LEN(Q$);Q$;3  
0      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 "SYNERTEK SYM-1", we might want to print out only the first eight characters of A\$. We would do so like this:

```
PRINT LEFT$(A$,8)  
SYNERTEK  
  
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
S
SY
SYN
SYNE
SYNER
SYNERT
SYNERTE
SYNERTEK
SYNERTEK
SYNERTEK S
SYNERTEK SY
SYNERTEK SYM
SYNERTEK SYM-
SYNERTEK SYM-1
OK

```

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

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
SYNERTEK SYM-1
YNERTEK SYM-1
NERTEK SYM-1
ERTEK SYM-1
RTEK SYM-1
TEK SYM-1
EK SYM-1
K SYM-1
SYM-1
SYM-1
YM-1
M-1
-1
1
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
S          SY
Y          YN
N          NE
E          ER
R          RT
T          TE
E          EK
K          K
          S
S          SY
Y          YM
M          M-
-          -1
I          I
```

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$ = A$ + " " + "BASIC"
```

```
OK
PRINT B$
SYNERTEK SYM-1 BASIC
```

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$,11)+"*"+MID$(B$,13,5)+RIGHT$(B$,5)
```

```
OK
PRINT C$
SYNERTEK*SYM-1*BASIC
```

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 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 a question such as "WHAT IS THE VOLUME OF A CYLINDER OF RADIUS 5.36 FEET, OF HEIGHT 5.1 FEET?" you can use "VAL" to extract the numeric values 5.36 and 5.1 from the question. 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 list.

```
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,BITS,SYNERTEK,RANDOM  
230 DATA MONDAY,"\*\*\*ANSWER\*\*\*", " FOO "  
240 DATA COMPUTER, FOO,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 CLOAD will terminate program execution when they finish.

<u>NAME</u>	<u>EXAMPLE</u>	<u>PURPOSE/USE</u>
CLEAR		CLEARs all variables, resets "FOR" and "GOSUB" state, RESTORES data.
LIST	LIST LIST 100 LIST 100- LIST 100-300 LIST -100	Lists the current program in its entirety. List just line 100. List current program starting at line 100. List just lines 100 through line 300. List current program from beginning up to line 100.
	LISTx LISTx-y	The listing can be control-C'd (BASIC will finish listing the current line). Lists just line x Lists lines x-y
NULL	NULL 3	Sets the number of null (ASCII 0) characters printed after a carriage return/line feed. The number of nulls printed may be set from 0 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 0 before a paper tape of a program is read in from a Teletype (TELETYPE is a registered trademark of the TELETYPE CORPORATION). Use the null command to set the number of nulls to zero. When you punch a paper tape of a program using the list command, null should be set $\geq 3$ for 10 CPS terminals, $\geq 6$ for 30 CPS terminals. When not making a tape, we recommend that you use a null setting of 0 or 1 for Teletypes, and 2 or 3 for hard copy 30 CPS terminals. A setting of 0 will work with Teletype compatible CRT's.
RUN	RUN  RUN 200	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 optionally starting at the specified line number.

NEW	NEW	Deletes current program and all variables.
CONT	CONT	<p>Continues program execution after a <b>BREAK</b> is typed or a <b>STOP</b> statement is executed. You cannot continue after any error, after modifying your program, or before your program has been run. One of the main purposes of <b>CONT</b> 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 <b>BASIC</b> statements from which there is no escape. The <b>SYM-1</b> will keep executing the series of statements over and over, until you intervene or until power to the <b>SYM-1</b> is cut off. If you suspect your program is in an infinite loop, type in a <b>BREAK</b>. The line number of the statement <b>BASIC</b> was executing will be typed out. After <b>BASIC</b> has typed out <b>OK</b>, you can use <b>PRINT</b> to type out some of the values of your variables. After examining these values you may become satisfied that your program is functioning correctly. You should then type in <b>CONT</b> to continue executing your program where it left off, or type a direct <b>GOTO</b> statement to resume execution of the program at a different line. You could also use assignment (<b>LET</b>) statements to set some of your variables to different values. Remember, if you <b>BREAK</b> 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 a "CN" (continue not) error. It is impossible to continue a direct command. <b>CONT</b> always resumes execution at the next statement to be executed in your program when <b>BREAK</b> was typed.</p>
LOAD A		<p>Loads the program named <b>A</b> from the cassette tape. A <b>NEW</b> command is automatically done before the <b>LOAD</b> command is executed. When done, the <b>LOAD</b> will type out <b>OK</b> as usual. See Appendix <b>G</b> for more information.</p>
SAVE A		<p>Saves on cassette tape the current program in the memory. The program in memory is left unchanged. The tape file is named <b>A</b>. Note that since the file is named by the user, more than one tape file can be stored on one tape. See Appendix <b>G</b> for more information.</p>

## OPERATORS

<u>SYMBOL</u>	<u>SAMPLE STATEMENT</u>	<u>PURPOSE/USE</u>
=	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.
↑ (usually a shift/N on a TTY)	130 PRINT X↑3	Exponentiation (equal to X*X*X in the sample statement) 0↑0=1    0 to any other power = 0 A↑B, with A negative and B not an integer gives an FC error.
*	140 X=R*(B*D)	Multiplication
/	150 PRINT X/1.3	Division
+	160 Z=R+T+Q	Addition
-	170 J=100-I	Subtraction

### RULES FOR EVALUATING EXPRESSIONS:

- 1) Operations of higher precedence are performed before operations of lower precedence. This means the multiplication and divisions are performed before additions and subtractions. As an example,  $2+10/5$  equals 4, not 2.4. When operations of equal precedence are found in a formula, the left hand one is executed first:  $6-3+5=8$ , not -2.
- 2) The order in which operations are performed can always be specified explicitly through the use of parentheses. For instance, to add 5 to 3 and then 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 highest 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
(equal precedence for all six)	<>	NOT EQUAL
	<	LESS THAN
	>	GREATER THAN
	≤	LESS THAN OR EQUAL
	≥	GREATER THAN OR EQUAL

(These 3 below are Logical Operators)

7) NOT	LOGICAL AND BITWISE "NOT" LIKE NEGATION, NOT TAKES ONLY THE FORMULA TO ITS RIGHT AS AN ARGUMENT
8) AND	LOGICAL AND BITWISE "AND"
9) OR	LOGICAL AND BITWISE "OR"

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

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

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

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

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

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

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

The following truth table shows the logical relationship between bits:

<u>OPERATOR</u>	<u>ARG. 1</u>	<u>ARG. 2</u>	<u>RESULT</u>
AND	1	1	1
	0	1	0
	1	0	0
	0	0	0
OR	1	1	1
	1	0	1
	0	1	1
	0	0	0
NOT	1	-	0
	0	-	1

FC  
11=0  
00=1  
10=1  
01=1

EXAMPLES: (In all of the examples below, leading zeros or 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 111 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 I=-2            The sixteen bit complement of 1 is 111111111111110, which is equal to -(1+1) or -2.

A typical use of the bitwise operators is to test bits set in SYM's I/O ports which reflect the state of some external device. Bit position 7 is the most significant bit of a byte, while position 0 is the least significant.

For instance, suppose bit 1 of location A800 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 (&"A800" AND 2) THEN 10            This line will execute over
and over until bit 1 (masked
or selected by the 2)
becomes a 1. When that
happens, we go to line 20.

20 PRINT "INTRUDER ALERT"                        Line 20 will output "INTRUDER
ALERT".
```

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

```
10 WAIT &"A800", 2                                This line delays the execution of the next
statement in the program until bit 1 of
A800 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 prints out any changes in the sense switches.

```
10 A=300:REM SET A TO A VALUE THAT WILL FORCE PRINTING
20 J=PEEK(sense 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, 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 ("FOO").

<u>NAME</u>	<u>EXAMPLE</u>	<u>PURPOSE/USE</u>
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 " FOO",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. ("SYM" 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, FNK0, FNR2. User defined functions are restricted to one line. A function may be defined to be any expression, but may only have one argument. In the example B & 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),D\$(2,2,2)	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 maximum is about 34 dimensions.
	115 DIM Q1(N),Z(2*1)	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 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 no STEP was specified, it is assumed to be one. If the step is positive and the new value of the variable is = the final value (9.3 in this example), or the step value is negative and the new value of the variable is = the final value, then the first statement following the FOR statement is executed. Otherwise, the statement following the NEXT statement is executed. All FOR loops execute the statements between the FOR and the NEXT at least once, even in cases like FOR V=1 TO 0.

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 & 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+23.4 GOTO 92

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 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 relation 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 26.

INPUT 3 INPUT V,W,W2

Requests data from the terminal (to be typed in). Each value must be separated from the preceding 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 4.5E-3 or "CAT". If more data was requested in an INPUT statement than was typed in, a "???" is printed and the rest of the data should be typed in. If more data was typed in than was requested, the extra data will be ignored. BASIC will print the warning "EXTRA IGNORED" when this happens. 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 accepting data from the terminal. No "?" is typed as a prompt character. If carriage return is typed to an input statement, BASIC returns to command mode. Typing CONT after an INPUT command has been interrupted will cause execution to resume at the INPUT statement.

LET 300 LET W=X  
310 V=5.1

Assigns a value to a variable.  
"LET" is optional.

NEXT 340 NEXT V  
345 NEXT  
350 NEXT V,W

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

ON...GOTO 100 ON I GOTO 10,20,30,40

Branches to the line indicated by the I'th number after the GOTO. That is:  
IF I=1, THEN GOTO LINE 10  
IF I=2, THEN GOTO LINE 20  
IF I=3, THEN GOTO LINE 30  
IF I=4, THEN GOTO LINE 40.

If I=0 or I attempts to select a non-existent line ( $\geq 5$  in this case), the statement after the ON statement is executed. However, if I is  $\geq 255$  or  $< 0$ , an FC error message will result. As many line numbers as will fit on a line can follow an ON...GOTO.

105 ON SGN(X)+2 GOTO 40,50,60

This statement will branch to line 40 if the expression X is less than zero, to line 50 if it equals zero, and to line 60 if it is greater than zero.

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). The byte to be stored must be  $\geq 0$  and  $\leq 255$ , or an FC error will occur. The address (I) must be  $\geq 0$  and  $\leq 65535$ , or an FC error will result. Careless use of the POKE statement will probably cause you to "poke" BASIC to death; that is, the machine will hang, and you will have to reset the SYM-1 and restart 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 subroutines. You could also use PEEK and POKE to write a memory diagnostic or an assembler in BASIC.

PRINT 360 PRINT X,Y;Z  
370 PRINT  
380 PRINT X,Y;  
390 PRINT "VALUE IS";A  
400 PRINT A2,B,

Prints the value of expressions on the terminal. If the list of values to be printed out does not end with a comma (,) or a semicolon (;), then a carriage return/line feed is executed after all the values have been printed. Strings enclosed in quotes (") may also be printed. If a semicolon separates two expressions in the list, their values are printed next to each other. If a comma appears after an expression in the list, and the print head is at print position 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 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 SN error 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.
STOP	9000 STOP	Causes a program to stop execution and to enter command mode. Prints BREAK IN LINE 9000 (as per this example). CONT after a STOP branches to the statement following the STOP.

WAIT 805 WAIT I,J,K  
806 WAIT I,J

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  $\geq 0$  and  $\leq 65535$ .

## INTRINSIC FUNCTIONS

ABS(X) 120 PRINT ABS(X)

Gives the absolute value of the expression X. ABS returns X if  $X \geq 0$ ,  $-X$  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:

$INT(X*10^D+.5)/10^D$

RND(X) 170 PRINT RND(X)

Generates a random number between 0 and 1. The argument X controls the generation of random numbers as follows:

$X < 0$  starts a new sequence of random numbers using X. Calling RND with the same X starts the same random number sequence.  $X = 0$  gives the last random number generated. Repeated calls to  $RND(0)$  will always return the same random number.  $X > 0$  generates a new random number between 0 and 1.

Note that  $(B-A)*RND(1)+A$  will generate a random number between A & 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= $SIN(X/57.2958)$ . (This function must be loaded separately. See Application Note # 53-SSC)

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  $\geq 0$  and  $\leq 255$ .

USR(I)	200 PRINT USR(I)	Calls the user's machine language sub-routine with the argument I. The sub-routine's address must have been previously POKE'd into locations in page zero. See POKE, PEEK, and Appendix H.G
USR(I,J,...,Z)	340 PRINT USR (I,J,K)	Calls the user's machine language sub-routine whose address is specified by the first parameter I, with the arguments J thru Z. No POKING of page zero is necessary. Note that at least one argument must be given, or BASIC will assume that the call is of the format given above. See Appendix H.G
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 $\pi/2$ . ( $\pi/2=1.5708$ ) (This function must be loaded separately. See Application Note # 53-SSC)
COS(X)	200 PRINT COS(X)	Gives the cosine of the expression X. X is interpreted as being in radians. (This function must be loaded separately. See Application Note # 53-SSC)
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(O)	Gives the number of memory bytes currently unused by BASIC.
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 $\text{LOG}(X)/\text{LOG}(Y)$ . Example: The base 10 (common) log of 7 = $\text{LOG}(7)/\text{LOG}(10)$ .
PEEK(I)	356 PRINT PEEK(I)	The PEEK function returns the contents of memory address I. The value returned will be $\geq 0$ and $\leq 255$ . If I is $>65535$ or $< 0$ , an FC error will occur. An attempt to read non-existent memory address will return an unknown 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. I must be $\geq 0$ and $\leq 255$ or an FC error will result.

TAN(X)      200 PRINT TAN(X)      Gives the tangent of the expression X. X is interpreted as being in radians. (This function must be loaded separately. See Application Note # 53-SSC)

## 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.

<u>NAME</u>	<u>EXAMPLE</u>	<u>PURPOSE/USE</u>
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\$="FOO"+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 "," or ":" character.
READ	50 READ X\$	Reads a string from DATA statements within the program. Strings do not have to be quoted; but if they are not, they are terminated on a "," or ":" character or end of line and leading spaces are ignored. See DATA for the format of string data.
PRINT	60 PRINT X\$ 70 PRINT "FOO"+A\$	Prints the string expression on the user's terminal.



## STRING FUNCTIONS

ASC(X\$)	300 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 occur if X\$ is the null string.
CHR\$(I)	275 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 $\geq 0$ and $\leq 255$ . See Appendix K.
FRE(X\$)	272 PRINT FRE("")	When called with a string argument, FRE gives the number of free bytes unused by BASIC. Identical to FRE with numeric argument.
LEFT\$(X\$,I)	310 PRINT LEFT\$(X\$,I)	Gives the leftmost I characters of the string expression X\$. If $I \leq 0$ or $> 255$ an FC error occurs.
LEN(X\$)	220 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 expression X\$ starting at character position I. If $I > \text{LEN}(X\$)$ , then MID\$ returns a null (zero length) string. If $I \leq 0$ or $> 255$ , an FC error occurs.
MID\$(X\$,I,J)	340 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 $I > \text{LEN}(X\$)$ , MID\$ returns a null string. If I or J $\leq 0$ 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)	320 PRINT RIGHT\$(X\$,I)	Gives the rightmost I characters of the string expression X\$. When $I \leq 0$ or $> 255$ an FC error will occur. If $I > \text{LEN}(X\$)$ then RIGHT\$ returns all of X\$.
STR\$(X)	290 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\$)      280 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 of the string is not a plus (+) or minus (-) sign, a digit or a decimal point (.) then zero will be returned.

### SPECIAL CHARACTERS

<u>CHARACTER</u>	<u>USE</u>
@	Erases current line being typed, and types a carriage return/line feed.
←	(backarrow or underline) Erases last character typed. IF no more characters are left on the line, types a carriage return/line feed. "←" is usually a shift/0.
CARRIAGE RETURN	A carriage return must end every line typed in. Returns print head or CRT cursor to the first position (leftmost) on line. A line feed is always executed after a carriage return.
BREAK	Interrupts execution of a program or a list command. BREAK has effect when a statement finishes execution, or in the case of interrupting a LIST command, when a complete line has finished printing. In both cases a return is made to BASIC's command level and OK is typed. Prints "BREAK IN LINE XXXX" , where XXXX is the line number of the next statement to be executed.
: (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.
CONTROL/T	Typing a Control/T once causes BASIC to suppress all output until a return is made to command level, an input statement is encountered, another control/T is typed, or an error occurs.
?	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 COMMENTS

- 1) To read in a paper tape with a program on it, type a control/T and feed in tape. Type control/T 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.

Each line must be followed by three rubouts. If there are lines without line numbers (direct commands) the SYM-1 will fall behind the input coming from paper tape, so this is not recommended.

Using null in this fashion will produce a listing of your tape (use control/T method if you don't want a listing).

- 2) To punch a paper tape of a program, set the number of nulls to 3 for 110 BAUD terminals (Teletypes) and 6 for 300 BAUD terminals. Then, type LIST; but, do not type a carriage return.

Now, turn on the terminal's paper tape punch. Put the terminal on local and hold down the Repeat, Control, Shift and P keys at the same time. Stop after you have punched about a 6 to 8 inch leader of nulls. These nulls will be ignored by BASIC when the paper tape is read in. Put the terminal back on line.

Now hit carriage return. After the program has finished punching, put some trailer on the paper tape by holding down the same four keys as before, with the terminal on local. After you have punched about a six inch trailer, tear off the paper tape and save for later use as desired.

- 3) Restarting BASIC at location zero (by entering the SYM-1 command G0 (CR)) will cause BASIC to return to command level and type "OK".
- 4) The maximum line length is 72 characters.\*\* If you attempt to type too many characters into a line, a bell (ASCII 7) is executed. At this point you can either type backarrow to delete part of the line, or at-sign to delete the whole line. The character you typed which caused BASIC to type the bell is not inserted in the line as it occupies the character position one beyond the end of the line.

\*\*For inputting only.

APPENDIX A  
INITIALIZATION DIALOG

STARTING BASIC

After you execute BASIC, it will respond:

MEMORY SIZE?

If you type a carriage return to MEMORY SIZE?, BASIC will use all the contiguous memory upwards from location 0200 hex that it can find. BASIC will stop searching when it finds one byte of ROM or non-existent memory. Memory must be greater than 512 bytes.

If you wish to allocate only part of the computer's memory to BASIC, type the number of bytes of memory you wish to allocate in decimal. This might be done, for instance, if you were using part of the memory for a machine language subroutine.

There are 4096 bytes of memory in a 4K system, and 16,536 bytes in a 16K system.

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

4096 210  
5582 7679

BASIC V1.1  
COPYRIGHT 1978 SYNERTEK CORP.

"XXXX" is the number of bytes available for program, variables, matrix storage and string space.

OK

You will now be ready to begin using BASIC.

APPENDIX B  
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:

<u>ERROR CODE</u>	<u>MEANING</u>
BS Bad Subscript	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 Redim'd Array	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 Illegal Quantity	Function Call error. The parameter passed to a math or string function was out of range. FC errors can occur due to: <ol style="list-style-type: none"><li>a) a negative matrix subscript (LET A(-1)=0)</li><li>b) an unreasonably large matrix subscript (&gt;32767)</li><li>c) LOG-negative or zero argument</li><li>d) SQR-negative argument</li><li>e) A+B with A negative and B not an integer</li></ol>

<u>ERROR CODE</u>	<u>MEANING</u>
FC (Con't)	f) a call to USR before the address of the machine language subroutine has been patched in
	g) calls to MID\$, LEFT\$, RIGHT\$, INP, OUT, WAIT, PEEK, POKE, TAB, SPC or ON...GOTO with an improper argument.
ID Illegal Direct	Illegal Direct. You cannot use an INPUT or DEFFN statement as a direct command.
NF Next without For	NEXT without FOR. The variable in a NEXT statement corresponds to no previously executed FOR statement.
OD Out of Data	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	Out of Memory. Program too large, too many variables, too many FOR loops, too many GOSUB's, too complicated an expression or any combination of the above (see Appendix D).
OV Overflow	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	Syntax error. Missing parenthesis in an expression, illegal character in a line, incorrect punctuation, etc.
RG Return without GOSUB	RETURN without GOSUB. A RETURN statement was encountered without a previous GOSUB statement being executed.
US Undef'd Statement	Undefined Statement. An attempt was made to GOTO, GOSUB or THEN to a statement which does not exist.
<b>/0</b> Division by Zero	Division by Zero.
CN Can't Continue	Continue error. Attempt to continue a program when an error occurred, or after a new line was typed into the program.

ERROR CODE

MEANING

LS  
String too long

Long String. Attempt was made by use of the concatenation operator to create a string more than 255 characters long.

ST  
Formula too complex

String Temporaries. A string expression was too complex. Break it into two or more shorter ones.

TM  
Type Mismatch

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

UF  
Undef'd Function

Undefined Function. Reference was made to a user defined function which had never been defined.

## APPENDIX C

### 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 (5 bytes) 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 64000), 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.
- 8) 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 of the intrinsic functions such as COS, INT and STR\$ take up only one byte of program storage. All other characters in programs use only 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 D

### 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 E  
DERIVED FUNCTIONS

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

<u>FUNCTION</u>	<u>FUNCTION EXPRESSED IN TERMS OF BASIC FUNCTIONS</u>
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	$ARGCOTH(X) = LOG((X+1)/(X-1))/2$

## APPENDIX F

### CONVERTING BASIC PROGRAMS NOT WRITTEN FOR SYNERTEK BASIC

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

- 1) Matrix subscripts. Some BASICs use " " and " " to denote matrix subscripts. Synertek 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 BASIC: DIM A\$(J).

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

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

<u>OLD</u>	<u>NEW</u>
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:

<u>OLD</u>	<u>NEW</u>
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 and C to zero.

In Synertek 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 rubouts at the end of each line, instead of the three per line recommended for use with Synertek BASIC.

To get around this, try to use the tape feed control on the Teletype to stop the tape from reading as soon as 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 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 G

### BASIC/MACHINE LANGUAGE INTERFACE

Synertek BASIC provides two forms of the USR function to provide the BASIC program with access to assembly language subroutines written by the user.

The first format of the USR function is the one most commonly seen in formal BASIC definition, though not necessarily, the most convenient. The format is:

#### USR (I)

This format of USR assumes that the address of the subroutine to be called has previously been placed in location(s) 000B and 000C of page zero with the POKE command, 000B containing the low order address, 000C containing the high order address. The parameter I is passed to the subroutine as a 16-bit signed integer in the [A,Y] register pair. Any result of the subroutine call is assumed to be returned as a 16-bit signed integer in the [A, Y] register pair.

The second format of the USR function allows the user to specify the subroutine address in the call, and also allows the passing of multiple parameters. The format is:

#### USR(I,J,...,Z)

This format of USR assumes that I is the address of the subroutine to be called, and that J through Z are the parameters to be passed to the subroutine. Each parameter starting with J is passed to the subroutine as a 16-bit signed integer that has been placed on the stack. The last parameter is the list is placed in the [A, Y] register pair instead of on the stack. No POKE'ing of page zero is necessary. Any result of the subroutine call is assumed to be returned as a 16-bit signed integer in the [A, Y] register pair.

Regardless of the format, the USR function should use an RTS to return to BASIC, or a JMP to a routine that will put the result in the [A, Y] register pair as a 16-bit signed integer value before returning.

To use separately loaded machine language routines, it is necessary to reserve some RAM space. This is done by replying appropriately to the MEMORY SIZE? question.

Example: To reserve the last 256 bytes of RAM on a 4K system, answer as follows:

```
MEMORY SIZE? 3840
```

BASIC will use only the first 3840 bytes of RAM, leaving 256 bytes available for your routines.

Example: The SYM-1 SUPERMON program contains a subroutine, OUTBYT, which will output a two digit hexadecimal representation of the accumulator when called. Its address is 82FA. It could be called in either of the following ways.

1. POKE &"000B", & "00FA"  
POKE & "000C", & "0082"  
X=USR (& "FF00")

FF will be printed.

2. X=USR (& "82FA", & "FF00")

FF will be printed.

APPENDIX H  
ASCII CHARACTER CODES

<u>DECIMAL</u>	<u>CHAR.</u>	<u>DECIMAL</u>	<u>CHAR.</u>	<u>DECIMAL</u>	<u>CHAR.</u>
000	NUL ↑ <sup>Q</sup>	043	11 +	086	5A V
001	SOH ↑ <sup>R</sup>	044	12 ,	087	5B W
002	STX ↑ <sup>B</sup>	045	13 -	088	5C X
003	ETX ↑ <sup>C</sup>	046	14 .	089	5D Y
004	EOT ↑ <sup>D</sup>	047	15 /	090	5E Z
005	ENQ ↑ <sup>E</sup>	048	16 0	091	5F [
006	ACK ↑ <sup>F</sup>	049	17 1	220 092	60 \
007	BEL ↑ <sup>G</sup>	050	18 2	093	61
008	BS ↑ <sup>H</sup>	051	19 3	094	62 f
009	HT ↑ <sup>I</sup>	180 052	20 4	095	63 ↑
010	LF ↑ <sup>I</sup>	053	21 5	096	64 < SHIFT-RETURN Σ
011	VT ↑ <sup>R</sup>	054	22 6	097	65 a
012	FF ↑ <sup>L</sup>	055	23 7	098	66 b
013	CR ↑ <sup>M</sup>	056	24 8	099	67 c
014	SO ↑ <sup>N</sup>	057	25 9	100	68 d
015	SI ↑ <sup>O</sup>	058	26 :	101	69 e
016	DLE ↑ <sup>P</sup>	059	27 ;	230 102	70 f
017	DC1 ↑ <sup>Q</sup>	190 060	28 <	103	71 g
018	DC2 ↑ <sup>R</sup>	061	29 =	104	72 h
019	DC3 ↑ <sup>S</sup>	062	30 >	105	73 i
020	DC4 ↑ <sup>T</sup>	063	31 ?	106	74 j
021	NAK ↑ <sup>U</sup>	064	32 @	107	75 k
022	SYN ↑ <sup>V</sup>	065	33 A	108	76 l
023	ETB ↑ <sup>W</sup>	066	34 B	109	77 m
024	CAN ↑ <sup>X</sup>	067	35 C	110	78 n
025	EM ↑ <sup>Y</sup>	068	36 D	111	79 o
026	SUB ↑ <sup>Z</sup>	069	37 E	240 112	p
027	ESCAPE ↑ <sup>L</sup>	070	38 F	113	q
028	FS ↑	071	39 G	114	r
029	GS ↑ <sup>J</sup>	200 072	40 H	115	s
030	RS ↑ <sup>A</sup>	073	41 I	116	t
031	US ↑ <sup>K</sup>	074	42 J	117	u
160 032	SPACE	075	43 K	118	v
033	!	076	44 L	119	w
034	"	077	45 M	120	x
035	#	078	46 N	121	y
036	\$	079	47 O	250 122	z
037	%	080	48 P	123	±
038	&	081	49 Q	124	π
039	'	210 082	50 R	125	τ
040	(	083	51 S	126	~
041	)	084	52 T	127	DEL
170 042	*	085	53 U		

LF=Line Feed

FF=Form Feed

CR=Carriage Return

DEL=Rubout

CONTROL KEY ↑



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 preceding 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 the KTM-2 and most other 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 BASIC's CHR\$ function.

APPENDIX I  
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
- 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

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

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

First Printing: December 1978

Revised: July 1980

Page 1: The following should be corrected as shown:

You have received one ROM as your BASIC language. This ROM is designed to run in your Synertek SYM-1 using the SUPERMON monitor.

Insert the ROM marked 02-0058A into socket U21. Before applying power, the following on-board jumpers must be changed.

<u>Remove the following Jumpers</u>	<u>Add the following Jumpers</u>
B-2	B-1
F-5	F-4
L-12	K-11, 12, 13, & 14
M-13	

If you have a program in socket U23 running at address C000-DFFF, then it must be relocated so as to not conflict with BASIC.

Page 23: The following line should be corrected:

WAS: The listing can be control--C'd (BASIC will finish listing the current line).

SHOULD BE: The listing can be interrupted by pressing the BREAK key (BASIC will finish listing the current line).

#### WRITE PROTECT BUG

When a warm-start entry to BASIC is made, system RAM (A600-A67F) remains write protected. This will cause any calls to monitor routines to fail. It will also cause BASIC SAVE and LOAD to fail.

To get around this problem with MON-1.0, after a warm-start perform a call to ACCESS to remove write protect.

```
.G 0           Warm-Start BASIC
OK
A=USR("&8B86",0)  Un-Write Protect System RAM
OK
```

To get around this problem with MON-1.1, after a warm-start do a "save A" with the recorder off. After this "dummy" save operation, all save and LOAD operations will function properly.

Page G-1: In order to return a 16 bit value, load the accumulator with the high order 8 bits, and the y-register with the low order 8 bits. Then perform a JMP \$D14C.

If no value is to be returned, a RTS may be used instead of JMP D14C.

#### HEXADECIMAL CONSTANTS

Hexadecimal constants in the range of 0000-7FFF and 8001-FFFF may be used as the first argument of the USR statement. In the PEEK and POKE statements, the permissible range of 0000-7FFF only. Use decimal constants for other values.