DTC
Extended Basic

DTC EXTENDED BASIC

VERSION 3.2

DATA TERMINALS & COMMUNICATIONS
1190 Dell Avenue
Campbell, CA 95008
(408) 378-1112

March 29, 1978

TABLE OF CONTENTS

INTRODUCTION

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

DTC BASIC is a programming language both easily understood and simple to use. It serves as an excellent "tool" for applications in the 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 it "commands". Designed specifically as an "interactive" language, you can give a command such as "PRINT 2 + 2", and DTC 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 DTC MICROFILE is "at your fingertips".

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

1) System Configuration
2) Version of BASIC
3) A detailed description of the error
   Include all pertinent information such as a listing of the program in which the error occurred, the data placed into the program and BASIC's printout.

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

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

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

If you are already familiar with BASIC programming, the following section may be skipped. Turn directly to the Reference Material on page 28.

FOR THOSE NEW TO BASIC

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

When the MICROFILE is powered ON, insert the BASIC program diskette into drive zero. Enter BA carriage return after the system level prompt (*). The system will load BASIC and print out the number of free bytes in your system and DTC BASIC 3.2 followed by the BASIC prompt OK.

We recommend that you try each example in this section as it is presented. This will enhance your "feel" for BASIC and how it is used.

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

NOTE: All commands to DTC 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 backspace (Control H), to eliminate the last character. Repeated use of backspace will eliminate previous characters. A rubout will delete a whole word. A Control X will eliminate the entire line that you are typing.

Now, try typing the following:

```
PRINT 10-4 (end with carriage returns, "-" means subtract)
```

DTC BASIC will immediately print:

```
6

OK
```

The print statement you typed in was executed as soon as you hit the carriage return key. BASIC evaluated the formula after the "PRINT" and then typed out its value, in this case 6.

Now try typing in this:

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

DTC BASIC will print:

```
.5       30

OK
```

As you can see, DTC 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 line into columns, each 14 characters wide. The result is a "," causes BASIC to skip to the next 14 column field on the terminal, where the value 30 was printed.

Commands such as the "PRINT" statements you have just typed in are called Direct Commands. There is another type of command called an Indirect Command. Every Indirect Command begins with a Line Number. A Line Number is any integer from 0 to 65529.

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 MICROFILE 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
```

DTC 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 Direct 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
```

DTC BASIC will reply with:

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

OK
```

Sometimes it is desirable to delete a line of a 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
```

DTC 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
```

DTC 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
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
```

DTC BASIC will reply with:

```
OK
```

Now type in:

```
LIST
```

DTC 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
```

DTC BASIC will reply with:

```
ONE THIRD IS EQUAL TO     .333333

OK
```

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

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

NOTE: Numbers are always printed with at least one trailing space

Any text to be printed is always to be enclosed in double quotes.

Try the following examples:

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

     ONE THIRD IS EQUAL TO .333333

     OK
```

B)
```
PRINT 1,2,3

     1     2     3
```

```
OK
```

C) PRINT 1;2;3

```
 1  2  3

OK
```

D) PRINT -1;2;-3

```
-1  2 -3

OK
```

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

The largest number that may be represented in DTC BASIC is $1.70141*10^{38}$ while the smallest negative number is $2.93874*10^{-39}$.

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 of 0 to 999999, it is printed as an integer.

3) If the absolute value of the number is greater than or equal to .1 and less than or equal to 999999, it is printed in fixed point notation, with no exponent.

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

Scientific notation is formatted as follows: SX.XXXXXESTT. (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. It is followed by the decimal point and then the other five digits of the mantissa. An "E" is then printed (for exponent), followed by the sign (S) of the exponent; then the two digits (TT) of the exponent itself. Leading zeroes are never printed; i.e. the digit before the decimal is never zero. Also, trailing zeroes are never printed. If there is only one digit to print after all trailing zeroes are suppressed, no decimal point is printed. The exponent sign will be "+" for 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 key 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 DTC BASIC will place them into:

NUMBER	OUTPUT FORMAT
+1	1
-1	-1
6523	6523
-23.460	-23.46
1E20	1E+20
-12.3456E-7	-1.23456E-06
1.234567E-10	1.23457E-10
1000000	1E+06
999999	999999
0.1	.1
0.01	1E-02
0.000123	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. However, only the first 7 digits are significant, and the seventh digit is rounded up.

```
PRINT 1.23456789012345678901
1.23457
```

```
OK
```

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

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

OK
```

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

If you haven't already guessed, what the program above 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 of three circles. This was accomplished by typing a carriage return to the input statement (thus a blank line). Press the BREAK key to stop the program.

The letter "R" in the program we just used is 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).

Below are some examples of legal and illegal variable names:

LEGAL	ILLEGAL	
TP	TO	(variable names cannot be reserved
PSTG4$	FOR	words however reserved words can
COUNT		be imbedded as in FEND).
PRUN	3X	(Must start with alphabetic char.)

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:

ABS AND AS ASC ATN AUTO BIN$ BKSP BYE CAR CDBL CHR$
CINT CLEAR CLOSE CONT COS CSNG CVD CVI CVS DATA DEF
DEFDBL DEFINT DEFSNG DEFSTR DEL DELAY DELETE DIM EDIT
ELSE END EOF EQV ERASE ERL ERR ERROR EXIT EXP FIELD FN
FOR FRE GET GOSUB GOTO HEX$ IF IMP INP INPUT INSTR INT
KILL LABEL$ LCHR$ LEFT$ LEN LET LINE LINPUT LIST LLINE
LLIST LOAD LOC LOF LOG LPOS LPRINT LSET MAX MID$ MIN
MKD$ MKI$ MKS$ MOD MON NEW NEXT NOT NXTR OCT$ ON OPEN OR

```
OUT     PEEK  POKE  POS  PRINT  PUT  RANDOMIZE  READ  REM     REN
RESTORE  RESUME  RETURN  RIGHT$  RND  ROL  ROR    RSET  RUN  SAVE
SGN  SHL  SHR  SIN  SLEEP  SPACE$  SPC      SQR  STEP  STOP  STR$
STRING$  SWAP  TAB  TAN    THEN  TLOAD  TO  TRIM$  TRIML$  TRIMR$
TROFF TRON  UNLOAD  USING  USR  VAL  VARPTR  WAIT  XOR
```

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

Try the following:

```
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 MICROFILE'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 three things occur:

1) A CLEAR command is typed in

2) A RUN command is typed in

3) 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 MICROFILE and run, it will ask for a value for B. Type any value you wish in. The MICROFILE will then come to the "IF" statement. Between the "IF" and the "THEN" portion of the statement there are two expressions separated by a relation.

A relation is one of the following six symbols:

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

The IF statement is either true or false, depending upon whether the two expressions satisfy the relation or not. For example, in the program we just did, if 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 on your own is the quickest and easiest way to understand how BASIC works. Remember to stop these programs, just press the BREAK key.

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 a 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<=10 we go back to the statement following the "FOR" statement. The overall operation then is the same as with the previous two programs.

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 15 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 for 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 Page 48 for additional information).

"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 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 would try to return; and since there was no "GOSUB" which called the subroutine an RG error would occur. Each "GOSUB" executed in a program should have a matching "RETURN" executed later, 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 statement until we find one that matches the guess.

If more values are read than there are numbers in the DATA statement, 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". DTC MICROFILE 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$="DTC MICROFILE"

OK
PRINT A$
DTC MICROFILE
OK
```

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

Now that we have set A$ to the 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 ("DTC")
13            3

OK
```

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

The number of characters in a string expression may range from 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 "DTC MICROFILE", we might want to print out only the first three characters of A$. We would do so like this:

```
PRINT LEFT$(A$,3)
DTC

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

   D
   DT
   DTC
   DTC
   DTC M
   DTC MI
   DTC MIC
   DTC MICR
   DTC MICRO
   DTC MICROF
   DTC MICROFI
   DTC MICROFIL
   DTC MICROFILE
```

```
OK
```

Since A$ has 13 characters, this loop will be executed with N=1,2, 3,...,12,13. 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
```

```
DTC MICROFILE
TC MICROFILE
C MICROFILE
 MICROFILE
MICROFILE
ICROFILE
CROFILE
ROFILE
OFILE
FILE
ILE
LE
E

OK
```

"MID$" returns a string starting at the Nth position to the end (last character) of A$. 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
```

```
D           DT
T           TC
C           C
             M
M           MI
I           IC
C           CR
R           RO
O           OF
F           FI
I           IL
L           LE
E           E

OK
```

See the Reference Material for more detail 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$="THE "+A$


OK
PRINT B$
THE DTC MICROFILE


OK
```

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

```
C$=LEFT$(B$,3)+"-"+MID$(B$,4,3)+"-"+RIGHT$(B$,10)


OK
PRINT C$
THE-DTC-MICROFILE


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:

```
NUM$="567.8"

OK
PRINT VAL(NUM$)
567.8

OK
NUM$=STR$(3.1415)

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

OK
```

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

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

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

OK
```

If you have an application where a user is typing in a question such as "WHAT IS THE VOLUME OF 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.

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 THROUGH ALL ELEMENTS,
187 REM CHECK TO SEE IF WE EXCHANGED ANY.
188 REM 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,DTC,MICROFILE,RANDOM
230 DATA MONDAY,"***ANSWER***","FOO"
240 DATA COMPUTER,FOO,ELP,MILWAUKEE,SEATTLE,SAN FRANCISCO
```

ENTERING PROGRAMS

Programs may be typed directly into BASIC at the terminal or they may be composed with the EDITOR Program and "TLOADed". Either way the following rules apply.

1. All program lines (as opposed to command or direct statement lines) begin with a unique statement number between 1 and 65529.

2. Statements may be entered in any order, but they are stored by BASIC in numerically increasing order.

3. Lines may be up to 255 characters in length, including the statement number. Leading blanks are preserved.

4. Multiple BASIC statements may appear on the same line. They are separated by ":".

5. Comments may appear at the end of a line. Comments are preceeded by a single quote ('), which is equivalent to ":REM...".

6. Spaces may delimit words. Spaces cannot appear within key words or names (eg: GOTO, not GO TO).

7. To execute BASIC programs, load the systems disk containing BASIC in Drive 0. When the prompt "*" occurs, enter "BA" carriage return.

```
DTC MICROFILE

*BA

18793 BYTES FREE
D.T.C. BASIC 3.2

OK
```

8. The prompt OK indicates BASIC is loaded and ready to be used.

COMMANDS

Command Syntax:

In the descriptions below, and throughout this manual, certain notations are used to help describe command and statement formats.

1. Anything in capital letters stands for itself, that is something you type in directly. For instance, GOTO, BYE, CLEAR, NEW.

2. Anything in lower case letters stands for an argument or parameter of the specified type to be supplied as necessary. If the lower case name has a "$" following it, the parameter must be a string or a string expression. For instance: "filenames$" is a string or string expression representing a file name (e.g. "MASTR"); "disk" is a numeric value or expression representing a disk number.

3. Optional fields are shown surrounded by "{" and "}".

NAME	EXAMPLE	PURPOSE/USE
AUTO {N,I}	AUTO 100,10	The AUTO command numbers the input program line automatically beginning with the line number specified. Each successive statement is incremented by the value specified. Default values are 10,10.
BYE		Exits BASIC and returns to the DTC MICROFILE monitor. Any open files are closed.
CAR {width}		Sets the width of the output line (the point at which a carriage return is automatically inserted by BASIC) to the specified value. The size must lie in the range 15 to 225. If a size of 0 is specified, lines are presumed to be infinitely long. That is, carriage returns are never inserted. This mode is

especially useful when plotting at the terminal. However, in this case the POS function will return inacurate results for true positions in excess of 225.

CAR without an argument resets the carriage width to the default value of 132 characters. CAR applies to the terminal and line ports only. Lines on disk are always infinitely long.

CLEAR {space}

CLEAR deletes all variables, but preserves the program in memory. If an argument value is given, it sets the amount of space to be used by strings and string variables to the number of bytes specified. If no argument is given, the amount of string space remains unchanged. When BASIC is first started, the string area size is 1000 bytes.

CONT

Continues program execution after a BREAK is typed or an END or STOP statement is executed. You cannot continue after any error, after modifying your program, or before your program has been run. You can, however, execute direct statements that print or change the value program variables. CONT always continues at the next statement to be executed. If you wish to continue at some other statement, use a direct GOTO statement. If an INPUT statement has been stopped (by typing a blank line), CONT will resume execution with that same input statement.

DELETE DELETE linel

Deletes program line numbered "linel".

DELETE -line2

Deletes all program lines up to and including line2.

	DELETE.	Deletes the current line.
	DELETE linel-line2	Deletes program lines linel through line2 (inclusive).
EDIT	EDIT line	Allows intra-line editing of a program line. EDIT is automatically entered if a syntax error is encountered during program execution. The EDIT command is described in detail on page 97.
EXIT		Exits BASIC and returns to the DTC Microfile monitor. This command is identical to the BYE command.
LIST	LIST linel	Lists line "linel" of the program if there is one.
	LIST or LIST -	Lists the entire program.
	LIST linel-	Lists all lines beginning at linel
	LIST.	Lists the current line.
	LIST -line2	Lists all lines up to (including) line2.
	LIST linel-line2	Lists from linel to line2, inclusive.
LLIST		This command is identical to the LIST command, and takes the same parameters. The listing, however, is directed to the data coupler (line) EIA port on the Microfile rather than to the terminal.
LOAD filename${,disk {R}} LOAD"Filename"{,disk{,R}}		Load a compressed BASIC program file (file type B). The file name may be any legal string or string expression. The second parameter

is the disk number and may be any expression that evaluates to a legal disk number.

LOAD A$,1,R
or
LOAD"PAYRO",1,R

If the disk parameter is not present, the file is loaded from disk 0. If ",R" is present the program is automatically run, and the variables from the previous programs are retained. If the ,"R" is not present, the variables are erased and BASIC returns to the command level. Any previous program is erased when LOAD is performed. If the file name is a literal instead of a string, then the name must be enclosed in quotes. The user files which were open with the previous program will remain open and positioned.

MON string$

MON takes one parameter, a string expression. This string is passed to the Microfile monitor as if it had been typed in from the terminal. (Example: MON "FI D0") This allows you to use some of the commands of the DTC Microfile from within BASIC, without exiting from BASIC.

No restriction is placed on the Microfile command that can be sent. However, if the program loaded by the Microfile monitor is too large, it might not be possible to return to BASIC. Programs run by the MON command must reside in the area below address 2800H. Monitor commands that can be run in this way are: CWL, PR, LA, SP, FI, RN, MD, PA, RAT, RAL, and the system commands HT, HL, SA, RU, and LO.

NEW

Clears all variables, erases all program lines, and closes all files. In other words, allows you to start fresh.

REN	{start {,increment}}	Renumbers your BASIC source program. The first value given is the starting number and the second is the increment to be used to get the next line number from the current one. If no increment value is given, a value of 10 is used. If no starting value is given, a value of 10 is used. Line numbers used within GOTO, GOSUB, THEN, ELSE, and RESTORE are also changed to reflect their new values.
RUN	{statement -number}	Runs the program. All variables are cleared before the program is run. If a statement number is given, RUN will start at the specified statement. A running program may be stopped by depressing the BREAK key at the terminal.
SAVE	SAVE filenames$ {,disk {,P}} SAVE filenames$ {,disk,T{linel-line2}}	Saves the current program on the disk specified, under the name specified by the string file$. The "T" forms of SAVE save the program as a type T file, which can be be printed or edited by the Microfile editor. The other forms of SAVE save the program as a type B file which contains a compressed form of the program. Type B files are much faster to load and require less disk space. If a file already exists of the specified name and type, SAVE will erase the old copy before saving the new copy. The P form of SAVE saves the file in a "Protected" mode. Such a program, when subsequently loaded, cannot be LISTed, EDITed, or SAVEd. Also, any commentary material (such as REM text) is deleted when the program is saved, making it somewhat shorter.
SLEEP		This command shuts off all disk motors until a disk activity, MON request or BYE restarts them.

TLOAD filename$
{,disk {,R}}
or
TLOAD "Filename"{,disk{,R}}

TLOAD works like LOAD, except that the file to be loaded must be type T. Also, TLOAD does not erase the current program so that it may be used to compose a single program from several files, each containing say, some subroutines. Lines are not renumbered during a TLOAD, so line number conflicts will cause replacement of the old line by the new line, just as though the lines had been typed in from the terminal.

TLOAD of programs with literal names requires quote marks to be added with the filename. The variables will be saved if the R parameter is given with the command. The ",R" option causes the program to be run after loading, otherwise the terminal returns to command level.

TRON
TROFF

Turns tracing of a program on and off. This is a very useful feature during debugging because you can see the exact sequence of statements that were executed. When tracing is turned on, each time a new program line is started, that line number is printed enclosed in "[]". No spaces are printed between statement numbers on a line.

STEP

STEP is like TRON except that execution pauses after typing the statement number and before the statement is executed. Typing any non-break character allows the statement to execute. To exit the STEP function, depress break then enter TROFF carriage return.

UNLOAD dn(dn...)

This command will eject the specified disks if the disk drive is so equipped. Unload without parameters will eject all disks.

EXPRESSIONS

BASIC allows both arithmetic and string expressions. The rules are given below.

String Operators

String Concatenation

"AB" + "CD" = "ABCD"

OPERATORS

SYMBOL	SAMPLE STATEMENT	PURPOSE/USE
=	A=100 LET Z=2.	Assigns a value to a variable. The LET is optional
-	B=-A	Negation. Note that O-A is subtraction, while -A is negation.
^	130 PRINT X^3	Exponentiation (equal to X*X*X in the sample statement) 0^0=1 0 to any other power = 0 A^B, with A negative and B not an integer gives an error.
*	140 X=R*(B*D)	Multiplication
/	150 PRINT X/1.3	Division
+	160 Z=R+T+Q	Addition
-	170 J=100-I	Subtraction
\	180 J=5\2	Integer Division (Truncates, so 5\2 = 2)

MOD 190 J=A MOD 4 Integer Modulus

RULES FOR EVALUATION EXPRESSIONS:

1) Operators of higher precedence are performed before operators 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 evaluation 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 PARENTHESES	
2)	^ or ^	EXPONENTIATION
3)	NEGATION	-X WHERE X MAY BE A FORMULA
4)	* /	MULTIPLICATION AND DIVISION
5)	\	INTEGER DIVISION
6)	MOD	MODULUS
7)	+ -	ADDITION AND SUBTRACTION
8)	MIN and MAX	MINIMUM/MAXIMUM OF TWO FACTORS
9)	SHL,SHR,ROL,ROR	SHIFTS LEFT, RIGHT - ROTATE LEFT, RIGHT
10)	RELATIONAL	= EQUAL

OPERATORS (equal precedence for all six)	<> NOT EQUAL
	< LESS THAN
	> GREATER THAN
	<= LESS THAN OR EQUAL
	>= GREATER THAN OR EQUAL

(These 6 below are logical operators)

11)	NOT	LOGICAL AND BITWISE "NOT" LIKE NEGATION, NOT TAKES ONLY THE FORMULA TO ITS RIGHT AS AN ARGUMENT
12)	AND	LOGICAL AND BITWISE "AND"
13)	OR	LOGICAL AND BITWISE "OR"
14)	XOR	A EXCLUSIVE OR B
15)	EQV	A EQUIVALENT TO B
16)	IMP	A IMPLICATION WITH B

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 formula after the IF is not equal to 0. That is to say, IF X THEN... is equivalent to IF X<>0 THEN...

SYMBOL	SAMPLE STATEMENT	PURPOSE/USE
=	10 IF A=15 THEN 40	Expression Equals Expression
<>	70 IF A<>0 THEN 5	Expression Does Not Equal Expression
>	30 IF B>100 THEN 8	Expression Greater Than Expression
<	160 IF B<2 THEN 10	Expression Less Than Expression

<=,=<	180 IF 100<=B+C THEN 10	Expression Less Than Or Equal To Expression
>=,=>	190 IF Q=>R THEN 50	Expression Greater Than or Equal To Expression
AND	2 IF A<5 AND B<2 THEN 7	If expression 1 (A<5) AND expression 2 (B<2) are both true, then branch to line 7.
OR	IF A<1 OR B<2 THEN 2	If either expression 1 (A<1) OR expression 2 (B<2) is true, then branch to line 2.
NOT	IF NOT Q3 THEN 4	If expression "NOT Q3" is true (because Q3 is false), then branch to line 4. NOTE: NOT -1=0 (NOT true=false)
XOR	IF A XOR B THEN 60	If A and Not B or B and Not A then the expression is true.
EQV	IF A EQV B THEN 12	If A and B or Not A and Not B then the expression is true.
IMP	IF A IMP B THEN 30	The expression of implication is true if Not A or B.

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

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

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

The following truth table shows the logical relationship between bits:

OPERATOR	ARG. 1	ARG. 2	RESULT
AND	1	1	1
	0	1	0
	1	0	0
	0	0	0
OR	1	1	1
	1	0	1
	0	1	1
	0	0	0
NOT	1	-	0
	0	-	1
XOR	1	1	0
	0	1	1
	1	0	1
	0	0	0
EQV	1	1	1
	1	0	0
	0	1	0
	0	0	1
IMP	1	1	1
	0	1	1
	1	0	0
	0	0	1

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

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

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

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

4 AND 2=0	4 equals binary 100 and 2 equals binary 10, so the result is binary 0 because none of the bits in either argument match to give a 1 bit in the result.
4 OR 2=6	Binary 100 OR'd with binary 10 equals binary 110, or 6 decimal.
10 OR 10=10	Binary 1010 OR'd with binary 1010 equals binary 1010, or 10 decimal.
-1 OR -2=-1	Binary 1111111111111111 (-1) OR'd with binary 1111111111111110 (-2) equals binary 1111111111111111, or -1.
NOT 0=-1	The bit complement of binary 0 to 16 places is sixteen ones (1111111111111111) or -1. Also NOT -1=0.
NOT X	NOT X is equal to -(X+1). This is because to form the sixteen bit two's complement of the number, you take the bit (one's) complement and add one.
NOT 1=-2	The sixteen bit complement of 1 is 1111111111111110, which is equal to -(1+1) or -2.

The operators MIN and MAX return the numeric data which meets the operator type. For example 3 MIN 7=3 or PRINT A MAX B. The greater number will be printed out using the MAX operator. The MIN function returns the lesser of the two.

The shift operators will perform shifts on 16 bit integers by the following operators:

A SHL B	shifts A left B bits, zeros shifted into right end.
A SHR B	shifts A right B bits, the sign bit is extended to the right.
A ROL B	rotates A left B bits, bits shifted off the left end cycle into the right end.

A ROR B rotates A right B bits, bits shifted off the right end cycle into the left end.

The manipulation is against the integer in a 16 bit register. One example: PRINT 5 SHL 2 which gives 20. A program to discover the bit action which occurs, may have SHL,SHR,ROL and ROR interchanged for the study, is as follows:

```
10 INPUT A
20 INPUT B
24 PRINT BIN$(A)
30 PRINT BIN$(A SHL B)
40 GOTO 10


OK
RUN
? 16
? 2
10000
1000000
```

VARIABLE AND CONSTANT TYPES

There are four types of values used in EXTENDED BASIC programming:

NAME	SYMBOL	# OF BYTES/VALUE
STRINGS (0 to 255 characters)	$	3
INTEGERS (must be -32768 and =< 32767)	%	2
DOUBLE PRECISION (exponent: -38 to +38) 16 digits	#	8
SINGLE PRECISION (exponent: -38 to +38) 6 digits	!	4

The type a variable will be is explicitly declared by appending one of the four symbols listed above to its name. Otherwise, the first letter of the variable is used to look into the table that indicates the default type for that letter. Initially (after CLEAR, after NEW, or after modifying a program), all letters are defaulted to SINGLE PRECISION.

The following four statements can be used to modify the DEFAULT table:

STATEMENT	DEFAULTS VARIABLE TO
DEFINT r	INTEGER
DEFSTR r	STRING
DEFDBL r	DOUBLE PRECISION
DEFSNG r	SINGLE PRECISION

These are described further in the section on Statements.

TYPING OF CONSTANTS

The type that a particular constant will be is determined by the following:

1) If it is more than 7 digits or "D" is used in the exponent, then it will be DOUBLE PRECISION.

2) If it is >32767 or <-32768, a decimal point (.) is used, or an "E" is used, then it is SINGLE PRECISION.

When a + or * or a comparison is performed, the operands are converted to both be of the same type as the most accurate operand. Therefore, if one or both operands are double precision, the operation is done in double precision (accurate but slow). If neither is double precision but one or more operands are single precision floating point, then the operation will be done in single precision floating point. Otherwise, both operands must be integers and the operation is performed in integer representation.

If the result of an integer + or * is too big to be an integer, the operation will be done in single precision and the result will be single precision. Division (/) is done the same as the above operation, except it is never done at the integer level. If both operands are integers, the operation is done as a single precision divide.

The operators AND, OR, NOT, \, and MOD force both operands to be integers before the operation is done. If one of the operands is >32767 or <-32768, an overflow error will occur. The result of these operations will always be an integer. (Except -32768\-1 gives single precision).

No matter what the operands to ^ are, they will both be converted to single precision. The functions SIN, COS, ATN, TAN, SQR, LOG, EXP, and RND also convert their arguments to single precision and give the result as such, accurate to 6 digits.

Using a subscript >32767 and assigning an integer variable a value too large to be an integer gives an overflow error.

TYPE CONVERSION

When a number is converted to an integer, it is truncated (rounded down). For example:

```
I%=.999          A%=-.01
PRINT I%         PRINT A%
0                -1
```

I% will perform as if the INT function was applied.

When a double precision number is converted to single precision, it is rounded off. For example:

```
D#=77777777
I!=D#
PRINT I!
7.77778E+07
```

No automatic conversion is done between strings and numbers. See the STR$, NUM, ASC, and CHR$ functions for this purpose.

CONSTANTS AND CONVERSIONS

Binary, Hexadecimal and Octal constants and conversions to decimal equivalents may be accomplished. The conversions are specified from the respective base with the ampersand followed by a capital letter indicating the input form. Examples are as follows: (&Bnnn Binary, &Onnn Octal and &Hnnn Hexadecimal)

```
PRINT &B101101001001
2889
OK
PRINT &O174 OR &174
124
OK
PRINT &H3FA
1018
```

The O for Octal may be omitted and the input form is then assummed to be octal. A sample program for the conversion of hexadecimal to decimal is as follows:

```
10 READ N
20 PRINT VAL ("&H"+TRIML$(STR$(N)))
30 DATA 651,12,14270
40 GOTO 10
```

The conversion of decimal numbers to their respective Binary, Octal or Hexadecimal equivalent may be done using the following forms: BIN$(nnn), OCT$(nnn) and HEX$(nnn).

Examples are:

```
OK
PRINT BIN$(7630)
1110111001110
OK
PRINT OCT$(345)
531
OK
PRINT HEX$(931)
3A3
OK
```

A simple program:

```
10 READ N
20 PRINT HEX$(N)
30 DATA 265,306,2179
40 GOTO 10
```

BASIC STATEMENTS

Just as a command may be used as a program statement, so can program statements be used as "commands". When entered without a statement number, a statement line, which may have multiple statements on it, is executed immediately.

Most statements are described below. Disk I/O statements are described in the section on DISK I/O.

DATA x1 {,x2...}
x1 is a numeric or string constant

The DATA statement provides values to be used by the READ statement. It is not itself executable and if encountered in the program flow is simply skipped. All statements taken together in a program define the data block for READ. They need not be contiguous. The RUN command resets the data block pointer so that the next READ statement will get the first value from the first DATA statement. The pointer is then advanced by READ until all data is read.

String constants may be quoted or unquoted. Quoted strings terminate at the ending quote (or end of statement, if that occurs first). Unquoted strings terminate at a comma or end of statement. Strings may not contain a quote (") within them.

DEF FNa {(V1{V2,...})} - Function Definition

The user can define functions like the built-in functions (SQR, SGN, ABS, etc.) through the use of the DEF statement. The name of the function is "FN" followed by any legal variable name, for example: FNX, FNJ7%, FNKO, FNR2$. User defined functions are limited to one statement. A function may be defined to be any expression. The variable names V1, V2, etc. are called formal parameters. When the function is used, the formal parameter names are replaced by the actual expression values in the call. So that:

```
DEF FNX2 (A,B) = SQR(A^2+B^2)
```

when called as:

```
R = FNX2 (25,P*Q+R)
```

would be evaluated as:

```
R = SQR((25)^2 + (P*Q+R)^2)
```

Any variables having the same name as formal parameter names are not affected. User functions may call other user functions.

If the same variable names used in the argument list are used in the call, then they must be in the same sequence. The program below shows the results of using the same variables in the call as the original definition, but out of sequence.

```
10 DEF FNA(A,B)=SQR(A^2+B^2)
30 PRINT "INPUT X":INPUT X
40 PRINT "INPUT Y":INPUT Y
45 LET H=FNA(X,Y)
50 PRINT H
65 PRINT "INPUT B":INPUT B
70 PRINT "INPUT A":INPUT A
75 LET P=FNA(B,A)
80 PRINT P
85 END
```

```
OK
RUN
INPUT X
? 5
INPUT Y
? 6
 7.81025 - (CORRECT)
INPUT B
? 5
INPUT A
? 6
 7.07107 - (INCORRECT)
```

DEF USRn = address

Users may also call separately compiled assembly language subroutines. The DEF USR statement tells BASIC the starting address of the subroutine. "n" may be any digit from 0 - 9, thereby defining 10 user routines. USR

is equivalent to USR0. The address given is the true address for values in the range <= address <=32767. For true addresses between 32768 and 65535 the complement address must be given (true address - 65536).

DEFDBL r{-s} -double precision
DEFINT r{-s} -integer
DEFSNG r{-s} -single precision
DEFSTR r{-s} -string

In these four statements the default type of all variables beginning with letters in the range r to s inclusive is changed. These four statements also allow multiple ranges if required for efficiency. Specification of only those variables necessary for definition within the range of variables may be stated e.g. DEFINT A,E,I-O,U. The statement has no effect on variables specified with an explicit type suffix (#,%,!,$). Initially after CLEAR, RUN, NEW, or any program modification, DEFSNG A-Z is assured. Care should be taken when using these statements since variables referred to without type indicators may not be the same after the statement is executed. It is recommended that these statements be used only at the start of a program, before any other statements are executed.

Example:

```
 10 I%   = 1 'Integer
 20 I!   = 2 'Single precision
 30 I#   = 3 'Double precision
 40 I$   ="ABC"'String
 50 PRINT  I 'Gives 2
 60 DEFINT I
 70 PRINT  I 'Gives 1
 80 DEFSTR I
 90 PRINT  I 'Gives ABC
100 DEFDBL I
110 PRINT  I 'Gives 3
```

DIM V1{(S1{S2,...})}
{,V2{(S3{,S4...})}...}

Allocate space for matrices. Matrices can have up to 255 dimensions (subscript positions). V1, V2,... are variable names; S1, S2,... are expressions giving the maximum value of the subscript in that position. The minimum subscript value at any position is 0, so for example

```
DIM A(3),B$(2,5)
```

creates two arrays. A has 4 elements, A(0), A(1), A(2), and A(3). B has 18 elements, from B(0,0) through B(2,5). If a matrix is not explicitly dimensioned with a DIM statement, it is assumed to be dimensioned (10).

Unlike some versions of BASIC, it is not necessary to dimension strings. The statement

```
DIM C$(30)
```

does not create one string of 30 characters, but sets up an array of 31 strings, each of which can be up to 255 characters long.

ELSE (SEE IF)

END

Terminates program execution without printing a BREAK message (see STOP). CONT after an end statement causes program execution to resume at the statement after the END statement. END can be used anywhere in the program, and is optional. An END is assumed to follow the last line of the program.

ERASE array{,array...}

```
ERASE J%
ERASE X%,I/#
ERASE A$
ERASE D/#,NMS%
```

Eliminates an array. If no such array exists an "ILLEGAL FUNCTION CALL" error will occur. ERASE must refer to an array, not an array element [ERASE B(9) would be ILLEGAL]. The space the array is using is freed up and made available for other uses. The array can be dimensioned again, but the values before the ERASE are lost.

FOR var=start TO ending value {STEP increment}

The variable (var) is set equal to the starting value expression following the equal sign. Then statements are executed until a NEXT is encountered. When NEXT is executed the increment value is added to "var" and the result tested against the ending value. If

the increment was postive and the new value is <= the ending value or the increment is negative and the new value is >= the ending value the program jumps back to the first statement following the FOR, otherwise it continues in line.

The index variable must be an integer or single precision variable. Integer variables are much faster to execute. If the STEP clause is absent, an increment of +1 is used.

The expressions for starting and ending values are executed only once, at the entry to the loop. All loops are executed at least one time, even in cases like

```
FOR I = 1 TO 0 STEP + 5
```

FOR loops may be nested to any depth (limited only by the available memory). The only requirement is that all active FOR loops must have different index variables. An attempt to begin a FOR loop with an index variable the same as an already active loop has the effect of terminating the active loop as well as any loops nested under it.

GOSUB statement number — Branches to the specified statement. When a RETURN is encountered control resumes at the statement following the GOSUB. GOSUB may be nested to any depth. RETURN always returns to the innermost active GOSUB in the nest.

GOTO statement number — Branches to the statement specified. Note that GOTO is one word.

```
IF test GOTO statement number
IF test THEN statement {:statement...} -
  {ELSE statement{:statement...}}
```

If the test expression is true (non-zero), statements are executed following THEN. If false, statements are executed following ELSE. The only restriction is that all of

the THEN-clause or ELSE-clause statements must fit on the same line. Any of the statements may also be IF-THEN-ELSE statements. In the case of nested IF-THEN and IF-THEN-ELSE statements, an ELSE is assumed to belong to the nearest IF on its left that does not already have an ELSE.

The clause "...THEN GOTO statement number..." may be replaced by "...GOTO statement number..." or "...THEN statement number...". Likewise, "...ELSE GOTO statement number..." may be replaced by "...ELSE statement number...".

INPUT {prompt$;} var1{,var2...}

Requests data from the terminal (to be typed in). Each value must be separated from the preceeding value by a comma (,). The last value typed should be followed by a carriage return. Only constants may be typed in as response to an INPUT statement, and the type (numerical string) of the constant must match the type of the variable that is to receive it. If a prompt string is specified, it is typed out preceeding the type in. If no prompt is specified, BASIC prompts with "? ". If more data was requested than was typed in, a prompt of "?? " is typed. If more data was typed in than was requested, the extra will be ignored. Strings may be quoted or not, as in the DATA statement.

If a carriage return alone is typed in response 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} variable = expression

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

LINE INPUT var$

Requests data from the terminal, without a prompt. A prompt will be added if enclosed in quotes. The entire line, up to the carriage return, is returned as a string to var$. In this form of input, typing carriage return alone returns a blank string and does not stop the program.

LINPUT {prompt$;} var1 {var2...}

Requests data from the auxiliary line to be typed in. In all other respects it behaves like INPUT from the terminal.

LLINE INPUT var$

Works just like LINE INPUT, except that the line is input from the auxiliary line or second terminal port on the MICROFILE.

LPRINT value {,value} {;value}...

See PRINT. This works like PRINT except that the output is directed to the auxiliary line or second terminal port of the MICROFILE. LPRINT USING string$; {value;{,value...}}

NEXT {var{,var}}

NEXT marks the end of the FOR loop with the matching variable name. If no value is given, the innermost FOR is matched. This is the fastest form. All FOR loops nested under the matched one are closed. NEXT with multiple variables may be used to match multiple FOR statements, eg:

NEXT V,W is equivalent to

NEXT V: NEXT W

ON expression GOTO stmt. no. {,stmt. no. ...}
ON expression GOSUB stmt. no. {,stmt. no. ...}

Performs a GOTO or GOSUB to the i'th statement number in the list, depending on the value of the expression. If the value is 1, the first statement number is used. If 2, the second statement {,stmt. no. ...} number is used, etc. A value of 0 or a value greater than the number of elements in the list causes the ON statement to be ignored. A value <0 or >255 is an error.

INP(I) 25 PRINT INP(I)

Gives the status of (reads a byte from) input port I. Result is =>0 and <=255.

OUT port, byte

Puts the type out on MICROFILE port "port". Both port and byte must be >=0 and <=255. This can be used to set the front panel display lights as follows:

The front panel line display port number is 66. The byte format is:

bits: x x x x v v v v

x(1) select digit 1
x(2) select digit 2
x(3) select digit 3
x(4) select digit 4

v(1,2,3,4)
digit 0-9,A,B,C,D,E,F,
(hexadecimal)

Digit values 0-9 give the digits 0-9. A value of 10 gives A, 11 gives B, 12 gives C, 13 gives D, 14 gives E, and 15 gives F. Multiple digit places can be selected simultaneously, so that

OUT 66, (8+4+2+1)*16+0 will clear the display to 0

OUT 66, (8)*16+10 will set the leftmost digit to "A".

POKE location, byte

The POKE statement stores the byte (0<=byte<=255) in the memory location specified. For locations between 0 and 32767, give true location. For true locations between 32768 and 65535, give the complement location (true location- 65536).

PRINT value {,value} or {;value} ...

Prints the value of expressions on the terminal. If the list of values does not end with a comma (,) or a semicolon, (;), then a carriage return/line feed is put out after all the values have been printed. A value may be any expression, including string expressions, or literal strings (enclosed in quotes). 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, the printer is spaced to the start of the next 14 column field (until the carriage is at column 14, 28, 42, 56, 70,...). If printing would continue past the end of line as defined by the CAR command, a carriage return/line feed is inserted at the end of line point.

The special functions SPC and TAB may only be used within a print statement. SPC (i) will print i spaces at the terminal (0<=i<=255). TAB(i) moves the printer to column i (0<=i<=255). 0 is the leftmost column number. If infinite lines are being used (CAR 0), TAB positions past column 255 may be inaccurate.

PRINT USING string$; {value {,value...}}

The PRINT USING statment can be used in situations where a specific output format is desired. This situation might be encountered in applications such as printing payroll checks or an accounting report. See Page 85 for a complete description of this command.

READ var{,var}

READ is exactly like INPUT, except that the data comes from DATA statements within the program (see DATA). Attempting to read beyond the end of data will result in an error.

REM comments

Allows the user to put comments in the program. REM statements are not executed, but can be branched to. A REM statement is terminated only by end of line, not by ":".

RESTORE {statement number}

Allows the re-reading of DATA statements. After a RESTORE, the next piece of data read will be the first value listed in a DATA statement at or beyond the specified statement number. If no statement number is specified, the first statement of the program will be used.

RETURN

Causes a return to the statement after the most recently executed GOSUB.

STEP (See FOR)

STOP

Causes a program to stop execution and to enter command mode. The message

BREAK IN LINE nnnn

is printed. Programs may be continued after a STOP. (See END).

SWAP var1, var2

Exchanges the values of two variables. Both must be of the same type (integer, single precision, double precision, or string). Either or both may be array elements.

THEN

(See IF)

TO

(See FOR)

WAIT port, mask {,select}

This statement reads the status of MICROFILE input port "port", exclusive OR's "select" with the status, and then AND's the result with "mask" until the result of that is non-zero. All three parameters must be >=0 and <=255. This should only be used to support special hardware of your own, as most other MICROFILE input ports are monitored by the system.

INTRINSIC FUNCTIONS

ABS(X) 120 PRINT ABS(X)	Gives the absolute value of the expression X. ABS returns X if X>=0, -X otherwise.
ASC(S$)	Returns the ASCII numeric value of the first character of the string expression S$. The parity bit is removed. An error will occur if S$ is the null string.
ATN(X) 10 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).
CDBL(X)	Converts the argument to a double precision number.
CHR$(I)	Assigns the I'th character of the ASCII set.
CINT(X)	Converts the argument to an integer number.
COS(X) 20 PRINT COS(X)	Gives the cosine of the expression X. X is interpreted as being in radians.
CSNG(X)	Converts the argument to single precision.
CVD(S$)	See DISK I/O
CVI(S$)	See DISK I/O
CVS(S$)	See DISK I/O
EOF(I)	See DISK I/O

EXP(X) 15 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) 70 PRINT FRE(0)

Gives the number of memory bytes currently unused by BASIC. Memory allocated for program.

FRE(S$)

STRING space is not included in the count returned by FRE. To find the number of free bytes in STRING space, call FRE with a STRING argument. (See FRE under STRING FUNCTIONS).

INSTR (I%,S1$,S2$)

or

A function has been provided to search one string for an occurrence of another string as a substring. This is the INSTR function. It takes one of two forms.

INSTR (S1$,S2$)

In the first form the string S1$ is searched for the substring 2$ starting at character position I%. The second form is identical, except that the search starts at character 1 of S1$. INSTR returns the character's position of the first occurrence of S2$ in S1$. If S1$ is null, 0 is returned. If S2$ is null then I% is returned, unless I% > LEN(S1$) in which case 0 is returned.

INT(X) 10 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
```

LCHR$(I)

If I=0, this function reads the next character from the auxiliary line or second terminal port of the MICROFILE.

All 8 bits are returned. This function reads without interpreting or attempting line editing. If I>0, LCHR$ reads one character from file I (See DISK I/O).

LEFT$ (S$,I)
30 PRINT LEFT$(S$,I)

Gives the leftmost I characters of the string expression S$. If I<=0 or >255 an FC error occurs.

LEN(S$)

Gives the length of the string expression S$ in characters (bytes). Non-printing characters and blanks are counted as part of the length.

LOC(I)

See DISK I/O

LOF(I)

See DISK I/O

LOG(X) 160 PRINT LOG(X)

Gives the natural (Base E) logarithm of its argument X. To obtain the Base Y logarithm of X use the formula LOG(X)/LOG(Y). Example:

7 = LOG(7)/ LOG(10).

LPOS(X)

Gives the current column position of the auxiliary line or second terminal printer. X is a dummy variable and is ignored. For true positions in excess of 255 (infinite lines) LPOS will give erroneous results.

MID$ (S$,I)
330 PRINT MID$(S$,I)

MID$ called with two arguments returns characters from the string expressions S$ starting at character position I. If I>LEN(I$), the MID$ returns a null (zero length) string. If I <=0 or >255, an FC error occurs.

MID$(S$,I,J)=A$
120 PRINT MID$(S$,10,6)=A$

MID$ with three arguments that returns a string composed of the characters of A$ starting with the Ith character of S$ and continuing for J characters of A$. Nulls are returned for non-existent locations and lengths in the string.

MID$(S$,I,J)
340 PRINT MID$(S$,I,J)

MID$ called with three arguments returns a string composed from the characters of the string expression S$ starting at the Ith character for J characters If I>LEN(S$), MID$ returns a null string. If I or J <=0 or >255, an FC error occurs. If J specifies more characters than are left in the string, all characters from the Ith on are returned.

MID$(T$,I,J)=MID$(A$,K,L)
510 MID$(T$,14,3)=MID$(A$,8,3)
520 PRINT T$

MID$, three function argument that causes T$ to be changed beginning at the Ith character of T$ and continuing for J characters. The characters changed inside T$ are from A$ beginning at the Kth character of A$ and continuing for L characters.

MKD$ (X)

See DISK I/O

MKI$ (I)

See DISK I/O

MKS$ (X)

See DISK I/O

NXTR(I)

See DISK I/O

PEEK (I)

The PEEK function returns the contents of memory address I. The value returned will be >=0 and <=255. For addresses greater than 32767, call PEEK with the complement address (I - 65536).

POS(X)

Gives the current column position of the terminal's printer. Otherwise this is like LPOS.

RIGHT$(S$,I)
30 PRINT RIGHT$(S$,I)

Gives the rightmost I characters of the string expression S$. When I<=0 or >255 an FC error will occur. If I>=LEN(S$) then RIGHT$ returns all of S$.

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.

RND(-PEEK(8197)) is a good random number preset, since memory cell 8197 is constantly changed by a system timer routine.

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).

SPACE$(nn)

Returns string of spaces equal to number specified.

SPC(I)

Prints I spaces. See PRINT.

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.

STR$(X)90 PRINT STR$(X)

Gives a string which is the character representation of the numeric expression X. For instance, STR$(3.1)= "3.1".

STRING$(String,nn)

Returns the first character of string specified, the length equal to nn. Also a specific character can be specified by

an alternate method: PRINT STRING$(CHR$(65),50) which returns a string of fifty A's.

TAB(I)

Moves the terminal printer to column I. See PRINT.

TAN(X) 200 PRINT TAN(X)

Gives the tangent of the expression X. X is interpreted as being in radians.

USRn(Z)

Calls a user's assembly language subroutine with an argument, Z. Any of 10 different routines may be called, by setting n to 0,1,...,9 (e.g.: USR0, USR5). The starting address of the user's routine must have been defined in a DEF USRn statement. Details of the calling sequence are given on page 103.

VAL(S$) 20 PRINT VAL(S$)

Returns the string expression S$ 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 a minus (-) sign, a digit or a decimal point (.) then zero will be returned.

ERROR AND BREAK DETECTION AND PROCESSING IN BASIC 3.2

BASIC 3.2 has 42 defined error messages which occur as a result of errors in programming or operations with input data. The standard error and break routines print out the abbreviated error message and leave the program in the BASIC command mode. (OK is typed). The user may now detect errors and provide his own error message and recovery action and continue in the program.

Commands for ERROR and BREAK Detection and Processing:

ON ERROR GOTO nnn	This statement must be given in program prior to the occurrence of any errors. nnn equals line number of the error test.
ON BREAK GOTO nnn	This statement must be given in program prior to a situation where a break may be given by the operator.
ERR=nn and ERL=nn	The error and error line variables where ERR=nn is equal to the expected error number and ERL=nn is the expected line number of the program statement. The ERL statement is optional in the trap subroutine.
ON ERROR GOTO 0 and ON BREAK GOTO 0	The on error go to zero statement is used whenever the error trap subroutine has not correctly specified the error. It restores system handling of errors.
RESUME RESUME NEXT RESUME nnn	The RESUME statement must occur in the error trap routine and the error handling routine to return to the program. An omission of the resume statement causes a NO RESUME statement to be output. The resume without specification causes a return to the program line where the error occurred. The resume next statement causes program execution to begin at the first program statement after the error line. The resume nnn allows program operation to continue at the specified line.

Rules For Using the Error and Break Detection Processing

1. The BASIC error and break handling routines must be preceded by the ON ERROR GOTO nnn or ON BREAK GOTO nnn statement prior to program execution where an error can occur. The "nnn" is equal to the program line number of the error or break processing trap. See the examples for clarification.

2. The error or break processing trap must use an error number for the logical test of the error. The line that the error occurred on may be given to further isolate the error. The program must not execute the error or break trap unless there is an error.

3. The error trapping subroutine must contain the ON ERROR GOTO 0 statement after all of the desired error and break possibilities have been tested to allow the system to print out the error that was not found in the subroutine.

4. The error trapping subroutine must contain a RESUME statement that points to the ON ERROR GOTO 0. The error handling subroutine must also contain a RESUME statement at its completion to return to the program.

MULTIPLE ERROR TRAP EXAMPLE

A separate statement for error and break must be given as shown in line 10 and line 20 below. The four lines 100 to 130 comprise the error trapping. The error and break test specify an error line number therefore the RESUME statement can be numbered. When the error trap maybe entered from multiple points in the program, omit the ERL variable and use the RESUME in the error handling routine without a line number. Resume next could be used depending on the error and the recovery technique. The break and error detection processing allows a more complete description of errors to program users.

LIST

```
10 ON ERROR GOTO 100
20 ON BREAK GOTO 100
30 FOR I=1 TO 5000
40 NEXT I
50 INPUT A
60 INPUT B
70 PRINT A/B
90 END
```

```
100 IF ERR=0 AND ERL=30 OR ERL=40 THEN 200
110 IF ERR=12 AND ERL=70 THEN 250
120 RESUME 130
130 ON ERROR GOTO 0
200 PRINT "YOUR BREAK ACKNOWLEDGED PLEASE BE PATIENT"
210 RESUME 30
250 PRINT "DIVISION BY ZERO NOT LOGICAL IN THE MICROFILE"
260 PRINT "PLEASE RE-ENTER YOUR FACTORS"
270 RESUME 50


OK
RUN
YOUR BREAK ACKNOWLEDGED PLEASE BE PATIENT
? 23
? 00
DIVISION BY ZERO NOT LOGICAL IN THE MICROFILE
PLEASE RE-ENTER YOUR FACTORS
? 34
? 6
 5.66667

OK
```

ERROR TRAP AND RECOVERY EXAMPLE

The example below shows how useful the error handling can be. Program line 100 checks for error 27, FILE NOT FOUND, and provides a suitable error message and a return to the input line for another try. The example also shows how TLOAD overlays when RUN is part of the load command.

```
OK
LIST


10 ON ERROR GOTO 100
20 MON "f1"
30 INPUT "Input program name to be TLOADED in BASIC: ";A$
40 TLOAD A$,0,R
50 STOP
100 IF ERR=27 AND ERL=40 THEN 130
110 RESUME 120
120 ON ERROR GOTO 0
130 PRINT "THE NAME ";A$" IS NOT A T TYPE FILE,PLEASE RESPECIFY."
140 RESUME 30

OK
```

```
RUN
$FL$ P
$BA$ P   PRG2 B   SYSCO T   $FS$ P   PRG1 T


Input program name to be TLOADED in BASIC:PRG2
THE NAME PRG2 IS NOT A T TYPE FILE,PLEASE RESPECIFY.
Input program name to be TLOADED in BASIC:PRG1
THIS IS A TLOAD BASIC PROGRAM FILE
END OF THE PROGRAM


OK
LIST


5 GOTO 200
10 ON ERROR GOTO 100
20 MON "fl"
30 INPUT "Input program name to be TLOADED in BASIC: ";A$
40 TLOAD A$,0,R
50 STOP
100 IF ERR=27 AND ERL=40 THEN 130
110 RESUME 120
120 ON ERROR GOTO 0
130 PRINT "THE NAME ";A$" IS NOT A T TYPE FILE,PLEASE RESPECIFY."
140 RESUME 30
200 PRINT "THIS IS A TLOAD BASIC PROGRAM FILE"
210 PRINT "END OF THE PROGRAM"
220 END


OK
```

DEFINING A NEW ERROR STATEMENT

This program shows how to define a new error number. In line 35 if T$ is not equal to MID$ of A$ as specified this is called error 43. The program checks the label of the disk against the password. The maximum error number that can be used is 255. The UNPRINTABLE ERROR message will occur for an error number that is not defined.

```
LIST


10 ON ERROR GOTO 100
20 INPUT "INPUT THE PASSWORD  ";A$
30 T$=MID$(LABEL$(0),9,4)
35 IF (T$=MID$(A$,9,4)) THEN 40 ELSE ERROR 43
40 PRINT "THE DISK IS CORRECT "
```

```
50 END
100 IF ERR=43 AND ERL=35 THEN 200
110 RESUME 120
120 ON ERROR GOTO 0
200 PRINT "I AM SORRY THE PASSWORD IS WRONG, TRY AGAIN"
210 RESUME 20


OK
RUN
INPUT THE PASSWORD SYSTEMS DISC WITH BASIC
I AM SORRY THE PASSWORD IS WRONG, TRY AGAIN
INPUT THE PASSWORD SYSTEMS DISK WITH BASIC
THE DISK IS CORRECT
```

LISTING CURRENTLY DEFINED ERRORS IN BASIC

To get a description of an error number in BASIC enter ERROR nn, where nn is the number in question. ERROR 0 to ERROR 42 are currently described in BASIC 3.2

```
ERROR 1
NEXT WITHOUT FOR


OK
ERROR 34
BAD RECORD NUMBER


OK
ERROR 41
OUT OF DISK SPACE
```

Any error not defined causes the UNPRINTABLE ERROR message. For example; ERROR 57, if entered will give the message.

DISK I/O SECTION

In previous material, facilities have been described for reading and writing information to the terminal (INPUT, PRINT) and auxiliary line (LINPUT, LPRINT), and for imbedding information in a program (READ, DATA). These techniques are useful when a small amount of information is required. When more data is needed, disk files are needed.

All files in the system have associated with them a name and type. File names are 1-5 characters in length. The first character cannot be NUL or DEL. File types are single characters:

T - Text file - sequential
R - Random file
B - Compressed BASIC program
P - MICROFILE machine language program

File types T and R may be manipulated by a BASIC user. The SAVE and LOAD commands use file types T and/or B. Type T files are compatible with the EDITOR and may be printed at the terminal.

Some commands require a disk number. The disk number to use is determined by the slot you have inserted the desired floppy disk into. It is not associated with the file itself, as name and type are.

In the commands below, "filename" may be a quoted string or any string expression that yields a 1-5 character name. Lower case letters in the filename are converted to upper case before use. "Disk" is any numeric expression evaluating to a legal number (0,1,...). File type is determined by the type of OPEN request. All sequential user files are type T. All random files are type R.

When a program wishes to read and write data to a disk file, it must first OPEN the file on the appropriate disk in one of several modes. The general form of the OPEN statement is:

OPEN<mode>{#}<file number>,<file name>{,<disk number>}

<mode> is a string formula whose first character is one of the following:

O Specifies sequential mode, positioned to end of file (appending output)

I Specifies sequential mode, positioned to beginning of file (input)
R Specifies random Input/Output mode

Sequential means that the file is a stream of characters that will be read or written in order much like an INPUT statement reads from the terminal and PRINT writes to the terminal. Random files are divided into groups of 128 characters called records. The nth record of file may be read or written at any time. Random files have other attributes that will be discussed in detail later.

<file number> is a formula that evaluates to an integer between one and sixteen and is used to associate the file being OPENed with a number that will be used to refer to the file in later I/O operations.

Examples:

```
OPEN "O",1,"OUTPT",O
OPEN "I",1,"INPUT"
```

The above two statements would open the files. OUTPT for sequential output and the file INPUT for sequential input on disk zero.

```
OPEN M$,N,F$,D
```

The above statement would open the file whose name was in the string F$ in mode M$ as file number N on disk D.

SEQUENTIAL ASCII FILE I/O

Sequential input and output files are the simplest form of disk input and output as they involve the use of the INPUT and PRINT statements with a file that has been previously OPENed.

To use an INPUT to read data from a file instead of the system console, use:

INPUT #<file number>,<variable list>

Where <file number> represents the number of the file, and <variable list> is a list of the variables to be read, as in a normal INPUT statement.

When data is read from a sequential input file using an INPUT statement, no question mark (?) is printed on the terminal. The format of data in the file should appear exactly the same way that it would be typed to a standard INPUT statement to the terminal.

Input and output may be mixed on sequential files. Output will be inserted at the current file position, which may be the middle of a line.

When reading numeric values, leading spaces are ignored, as are carriage returns and line feeds in the file. When a non-space, non-carriage return, non-line feed character is found, it is assumed to be part of a BASIC format number. The number terminates on a space, a carriage return, line feed, or a comma.

When scanning for string items, leading blanks, carriage returns, and line feeds are also ignored. When a character which is not a leading blank, carriage return or line feed is found, it is assumed to be the start of a string item. If this first character is a quote sign (") the item is taken as being a quoted string, and all characters between the first double quote (") and a matching double quote are returned as characters in the string value. This means that a quoted string in a file may contain any characters except double quote, e.g. carriage returns, line feeds and commas.

If the first character of a string item is not a double quote, then it is assumed to be an unquoted string literal. The string returned will terminate on a comma, carriage return or line feed.

Scanning both numeric and string items stops at the start of the next item or at end of line. Subsequent reads pick up where a previous read left off, even in the middle of a line.

In the case of either a quoted or unquoted string item if the length of the string exceeds 255 characters, the string immeiately terminates at 255 characters.

Also for both numeric and string items, if end of file (EOF) is reached when the items are being INPUT, the item will be terminated whether or not a closing quote was seen.

Example of sequential I/O (NUMERIC ITEMS):

```
500 OPEN "O",1,"FILE",0
510 PRINT #1,X,Y,Z
520 RESTORE #1
530 INPUT #1,X,Y,Z
```

```
PRINT #<file number>,<expression list>
```

or

```
PRINT #<file number>,
        USING <string expression>;<expression list>
```

are used to write data to a sequential file. Example of sequential I/O (quoted string items):

```
500 OPEN "0",1,"FILE"
510 PRINT #1,CHR$(34);X$;CHR$(34);
515 PRINT #1,CHR$(34);Y$;CHR$(34);CHR$(34);Z$;CHR$(34)
520 RESTORE #1
540 PRINT #1,X$,Y$,Z$
```

In this example, the strings being output (X$, Y$, Z$) are surrounded with double quotes through the use of the CHR$ function to generate the ASCII value for a double quote. This technique must be used if a string which is being output to a sequential data file contains commas, carriage returns, line feeds, or leading blanks that are significant.

When leading blanks are not significant and no commas, carriage returns or line feeds exist in the strings to be output, it is sufficient to insert commas between the strings being output, as in the following example:

```
500 OPEN "I",1,"FILE"
510 PRINT #1,X$;",";Y$;",";Z$
520 RESTORE #1
530 INPUT #1,X$,Y$,Z$
```

Print lines on files are always assumed to be infinitely long. That is, carriage returns are never automatically inserted.

RESTORE

The format of the RESTORE file statement is:

```
RESTORE #<file number>{#}<file number>...}
```

The first "#" is mandatory in order to distinguish this statement from the RESTORE data statement. RESTORE is used to reposition a file to its start ("rewinding" the file). It may be used to reread a file or to read a file that has just been written.

CLOSE

The format of the CLOSE statement is as follows:

```
CLOSE {#}<file number>{,{#}<file number>...}
```

CLOSE is used to finish I/O to a particular BASIC data file. After CLOSE has been executed for a file, the file may be reOPENed for input or output on the same or a different <file number>. A CLOSE to an unOPENed file has no effect. A CLOSE for a sequential output file writes the final buffer of output. A CLOSE to any OPEN file finishes the connection between the <file number> and the <file name> given in the OPEN for that file, and allows the <file number> to be used again in another OPEN.

A CLOSE with no argument CLOSEs all OPEN files.

NOTE

A FILE with a given name can be OPENed for sequential or random access with only one <file number> at a time.

DELETING DISK FILES

To delete a disk file, the user should use a KILL statement:

KILL {#}<file number>{,{#}<file number>...}

The KILL statement closes a file and deletes the file from the disk.

CLEAR and NEW always CLOSE all disk files automatically.

LINE INPUT

Often it is desirable to read a whole line of a file into a string without using quotes, commas or other characters as delimiters. This is especially true if certain fields of each line are being used to contain data items, or if a BASIC program saved in ASCII mode is being read as data by another program. The facility provided to perform this function is the LINE INPUT statement:

LINE INPUT #<file number>,<string variable>

LINE INPUT from a data file will return all characters up to a carriage return in <string variable>. LINE INPUT then skips over the following carriage return/line feed sequence so that a subsequent LINE INPUT from the file will return the next line. If the line is longer than 255 characters, only the first 255 are returned.

BACKSPACING SEQUENTIAL FILES

The statement:

BKSP {#}<file number>

Causes a sequential file to be backspaced one line. If the current position is in the middle of a line, the file is backspaced to the start of the current line. A BKSP issued when at the beginning of a file has no effect. BKSP may be used to reread a line or to position a file prior to deleting a line.

DELeting Lines

DEL {#}<file number>

This function deletes a line starting at the current file position and continuing through an end of line (CR/LF) or end of file. The sequence:

```
100 LINE INPUT #1,A$
        .
        .
        .


210 BKSP  #1
220 DEL   #1
230 PRINT #1, "NEW LINE"
```

will replace one line with a new one. If much deleting is done, disk space may be inefficiently used. File space may be compactd by copying the file to a new file, then erasing (KILLing) the old file. The new file may be renamed via a MON request if desired.

LOCating Records

The function:

```
LOC (<file number>)
```

will return the current position on a significant file. It counts the number of CR/LF sequences passed over, thus giving the current line number. The first line is numbered 0.

End of File (EOF) Detection

When reading a sequential data file with INPUT statements, it is

usually desirable to detect when there is no more data in the disk file. The mechanism for detecting this condition is the EOF function:

```
X=EOF(<file number>)
```

EOF returns TRUE (-1) when there is no more data in the file and FALSE (0) if there is more. If an attempt is made to INPUT past the end of a data file, an INPUT PAST END error will occur.

Example:

```
100 OPEN "I",1,"DATA",0
110 I=0
120 IF EOF(1) THEN 160
130 INPUT #1,A(I)
140 I=I+1
150 GOTO 120
160.......
```

In this example, numeric data from the sequential input file DATA is read into the matrix A. When end of file is detected, the IF statement at line 120 branches to line 160, and the variable I "points" one beyond the last element of A that was INPUT from the file.

Suppose one wishes to have a program that will calculate the number of lines in a BASIC program file that has been SAVEd and in TEXT mode:

```
10 INPUT "WHAT IS THE NAME OF THE PROGRAM";P$
20 OPEN "I",1,P$,0
30 I=0
40 IF EOF(1) THEN 70
50 I=I+1:LINE INPUT #1,L$
60 GOTO 40
70 PRINT "PROGRAM";P$;" IS ";I;" LINES LONG"
80 END
```

This example uses the LINE INPUT statement to read each line of the program into the "dummy" string L$, which is used essentially just to INPUT and ignore that part of the file.

RANDOM FILE I/O

Previously, we have discussed how data may be PRINTed or INPUT from sequential data files.

However, it is often desirable to access data in a random fashion, for instance to retrieve information on a particular part number or customer from a large data base stored on a floppy disk. If sequential files were used, the whole file would have to be scanned from the start until the particular item was found. Random files remove this restriction and allow a program to access any record from the first to the last in a speedy fashion.

Also, random files transfer data from variables to the disk output records and vice versa in a much faster, more efficient fashion than sequential files.

Random file I/O is more complex than sequential I/O, and it is recommended that beginners try sequential I/O first.

OPENing a FILE for Random I/O

Random I/O files are OPENed just like sequential data files, except the <mode> is R:

```
OPEN<"MODE">,<FILE NUMBER>,<FILE NAME$>[,DISK NUMBER]

  OPEN "R",1,"RAND",0
```

CLOSING Random Files

Random files must be closed when I/O operations are finished, just like sequential files. To CLOSE a random file, use the CLOSE operator as described previously.

```
  CLOSE {#}<file number>{{#},<file number>...}

  CLOSE #1,#7
```

READING & WRITING DATA TO A RANDOM FILE - GET & PUT

Each random file on disk has associated with it a "random buffer", described by a FIELD statement, of 128 bytes. When a GET or PUT operation is performed, data is transferred directly from the buffer to the data file (PUT statement)or from the data file to the buffer (GET statement).

The syntax of GET and PUT is as follows:

PUT {#}<file number>{,<record number>}

PUT #4,1

GET {#}<file number>{,<record number>}

GET #4,2

If <record number> is omitted from a GET or PUT statement, the record number that is one higher than the previous GET or PUT is read into the random buffer. Initially a GET or PUT without a record number will read or write the first record (record 0). The largest possible record number is 4095. If an attempt is made to GET a record which has never been PUT, all zeros are read into the record, and no error occurs.

It is not necessary to maintain contiguous record numbers. A single disk will support about 2000 random records. No sectors are allocated for unused numbers.

MOVING DATA IN AND OUT OF THE RANDOM BUFFER

So far we have described techniques for writing (PUT) and reading (GET) data from a file into its associated random buffer. Now we will describe how data from string variables is moved to and from the random buffer itself. This is accomplished through the use of the FIELD, LSET and RSET statements.

FIELD

The FIELD statement is used to associate some or all of a file's random buffer with a particular string variable. Then, when the file buffer is read with GET or written with PUT, string variables which have been FIELDed into the buffer will automatically have their contents read or written. The format of the FIELD statement is:

FIELD {#} <file number> {,<field size> AS <string variable>...}

FIELD #3,128 AS A$

<File number> is used to specify the file number of the file whose random buffer is being referenced. If the file is not a random file, a BAD FILE MODE error will occur.

<Field size> is used to set the length of the string in the random buffer.

<String variable> is the string which is being associated with a certain number of characters (bytes) in the buffer.

Multiple fields may be associated with string variables in a given FIELD statement. Each successive string variable is assigned a successive field in the random buffer:

```
FIELD #5, 10 AS A$, 20 AS B$, 30 AS C$
```

Thus, the above statement would assign the first 10 characters of the random buffer to the string variable A$, the next 20 characters to B$ and the next 30 characters to the variable C$.

It is important to note that the FIELD statement does not cause any data to be transferred to or from the random buffer. It only causes the string variables given as arguments to "point" into the random buffer.

Often it is necessary to divide the random buffer into a number of sub-records to make more efficient use of disk space. For instance, it might be desirable to divide the 128 character record up into two identical sub-records. To accomplish this, one need only place a "dummy" variable at the start of the FIELD statement to skip over the first sub-record in the record:

```
FIELD #1,64 AS D$, 20 AS NAME$,
        20 AS ADDRESS$, 24 AS OCCUPATION$
```

Then, the dummy variable D$ is used to skip over the first 64 characters in the record. Another way to do this would be to have a variable I that would select whether the first or second sub-record of a record was to be selected:

```
FIELD #1,64*(I-1) AS D$
        20 AS NAME$, 20 AS ADDRESS$, 24 AS OCCUPATION$
```

Here, if the variable I is one, (I-1)*64 = 0 characters will be skipped over, selecting the first sub-record. If I is two, 64 characters will be skipped over, selecting the second sub-record.

Another technique that is very useful is to use a FOR...NEXT loop and a matrix to set up sub-records in the random buffer:

```
1000 FOR I=1 TO 16
1010 FIELD #1, (I-1)*8 AS D$, 4 AS A$(I), 4 AS B$(I)
1020 NEXT I
```

In this example, we have divided the random buffer up into 16 sub-records, each composed of two fields, the first 4-character field in A$(X) and the second 4-character field in B$(X) where X is the sub-record number.

> The FIELD statement may be executed any number of times on a given file. It does not cause any allocation of string space, the only space allocation that occurs is for the string variables mentioned in the FIELD statement. These string variables have a one byte count and two byte pointer set up which points into the random buffer for the specified file.

LSET and RSET

When a GET operation is performed, all string variables which have been FIELDed into the random buffer for that file automatically have values assigned to them. The CVI, CVS and CVD functions may be used to convert any numeric fields in the record to their numeric values.

When going the other way, i.e. inserting strings into the random

buffer before performing a PUT statement, a problem arises. This is because of the way string assignments usually take place. For example:

```
LET A$=B$
```

When a LET statement is executed, the character string assigned to the left hand variable (A$) is created in string space. However, for assignments into the random we don't want this to happen. Instead, we want the string being assigned to be stored where the string variable was FIELDed.

In order to do this, two special assignment statements have been provided, LSET and RSET:

```
LSET <string variable>=<string formula>

RSET <string variable>=<string formula>
```

```
LSET A$=MKS$(V)
RSET B$="TEST"
LSET C$=MKD$(D#)
```

The difference between LSET and RSET determines what happens if the string value being assigned is shorter than the length specified for the string variable in the FIELD statement. LSET left-justifies the string, adding blanks to pad out the right side of the string if it is too short. RSET right-justifies the string, padding on the left. If the string value is too long, the extra characters at the end of the string are ignored.

NOTE

> Do not use LSET or RSET on string variables which have not been mentioned in a FIELD statement, or a SET TO NON DISK STRING error will occur.

LOC, LOF, NXTR

LOC is used to determine what the current (next) record number is for random files. In other words, it returns the record number that will be used if a GET or PUT is executed with the <record number> parameter omitted.

```
LOC (<file number>)

PRINT LOC (1)
15
```

LOC is also valid for sequential files, and gives the current line number of the file.

LOF is used to determine the highest record number ever written to a random file:

```
LOF (<file number>)

PRINT LOF (2)
200
```

An attempt to use LOF on a sequential file will cause a BAD FILE MODE error.

A random file may have large gaps of unused record numbers. It is time consuming to GET every possible record number to check for this. NXTR returns the next index in the file that actually has a real record on disk.

```
NXTR (<file number>)

200 GET #5,NXTR(5)
```

If there are no more real records, NXTR returns 4096.

DELETING RANDOM RECORDS

A random record may be deleted from a file using the statement:

```
DEL {#}<file number>{,<record number>}
```

When a record is deleted, its disk space is returned to the system and any subsequent attempt to read it will return all zeros.

USING NUMERIC VALUES IN RANDOM FILES

As we have seen, data is always stored in the random buffer through the use of string variables. In order to convert between string and numbers, a number of special functions have been provided.

To convert between numbers and strings:

MKI$(<integer value>)	Returns a two type string error if value is not >=-32768 and <=+32767. Fractional part lost.
MKS$(<single precision value>)	Returns a four byte string.
MKD$(<double precision value>)	Returns an eight byte string.

To convert between strings and numbers:

CVI(<two byte string>)	Returns an integer value.
CVS(<four byte string>)	Returns a single precision value.
CVD(<eight byte string>)	Returns a double precision value.

CVI, CVS, and CVD all give an error if the string given as the argument is shorter than required. If the string argument is longer than necessary, the extra characters are ignored.

These functions are extremely fast, as they convert between BASIC's internal representation for integers, single and double precision values and strings. Conventional sequential I/O must perform time consuming character scanning algorithms when converting between numbers and strings.

RANDOM FILE PROGRAM EXAMPLES

```
LIST

10 OPEN "R",1,"RAND",0
```

```
20 FIELD #1, 128 AS F$
30 INPUT R$
40 LSET F$=R$
50 PUT #1,1
60 GET #1,1
70 R$=F$
80 PRINT R$
90 CLOSE #1


OK
RUN
? NOW IS THE TIME FOR ALL GOOD MEN TO COME TO THE AID OF THEIR COUNTRY
NOW IS THE TIME FOR ALL GOOD MEN TO COME TO THE AID OF THEIR COUNTRY


OK
```

```
LOAD "RON1"


OK
LIST


10 FOR I=1 TO 10:  INPUT B$(I) : NEXT I
20 OPEN "R",#2,"RAND",0
30 FOR J=1 TO 10
40 FIELD #2,10*J-10 AS D$, 10 AS A$(J)
45 FOR I=1 TO 10:LSET A$(I)=B$(I):NEXT
50 PUT #2,1
60 NEXT J
70 CLOSE #2
80 END


OK
RUN
? PRNTR
? LINE2
? PLOT
? AUTO LF
? TOF
? SCROLL
? LPI
? PITCH
? VIDEO
? FORMAT


OK
LOAD"RON2"


OK
LIST


10 OPEN "R",#2,"RAND",0
20 FOR J=1 TO 10
30 FIELD #2,10*J-10 AS D$, 10 AS A$(J)
40 NEXT
50 GET #2,1
60 FOR J=1 TO 10
70 PRINT A$(J);
80 NEXT J
90 CLOSE #2
100 END


OK
RUN
```

```
PRNTR     LINE2     PLOT     AUTO LF     TOF     SCROLL     LPI     PITCH
VIDEO     FORMAT

OK
```

PRINT USING

The PRINT USING statement can be employed in situations where a specific output format is desired. This situation might be encountered in such applications as printing payroll checks or an accounting report. Other uses for this statement will become more apparent as you go through the text.

The general format for the PRINT USING statement is as follows:

(line number) PRINT USING <string>; <value list>

The "string" may be either a string variable, string expression or a string constant which is a precise copy of the line to be printed. Characters in the string will be printed just as they appear, with the exception of the formatting characters. The "value list" is a list of items to be printed. The string will be repeatedly scanned until:

1) The string ends and there are no values in the value list.

2) A field is scanned in the string, but the value list is exhausted.

The string should be constructed according to the following rules:

STRING FIELDS

!	Specifies a single character string field. (The string itself is specified in the value list).
\n spaces\	Specifies a string field consisting of 2+n characters. Backslashes with no spaces between them would indicate a field of 2 characters width, one space between them would indicate a field 3 characters in width, etc.

In both cases, if the string has more characters than the field width, the extra characters will be ignored. If the string has less characters than the field width, extra spaces will be printed to fill out the entire field.

Trying to print a number in a string field will cause a TYPE MISMATCH error to occur.

Example:

```
10 A$="ABCDE":B$="FGH"
20 PRINT USING "\  \";B$,A$
30 PRINT USING "!";A$,B$
```

(the above would print out)

```
FGH ABCD
AF
```

Note that where the "!" was used only the first letter of each string was printed. Where the backslashes were enclosed by two spaces, four letters from each string were printed (an extra space was printed for B$ which has only three characters). The extra characters in the first case and for A$ in the second case were ignored.

NUMERIC FIELDS

With the PRINT USING statement, numeric printouts may be altered to suit almost any applications which may be found necessary. This should be done according to the following rules:

Numeric fields are specified by the # sign, each of which will represent a digit position. These digit positions are always filled. The numeric field will be right justified; that is, if the number printed is too small to fill all of the digit positions specified, leading spaces will be printed as necessary to fill the entire field.

The decimal point position may be specified in any particular arrangement as desired; rounding is performed as necessary. If the field format specifies a digit is to precede the decimal point, that digit will always be printed (as 0 if necessary).

The following program will help illustrate these rules:

```
10 INPUT A$,A
20 PRINT USING A$;A
30 GOTO 10
RUN
? ##,12
12
? ###,12
 12
? #####,12
```

```
  12
?##.##,12
12.00
? ###.,12
12.
? #.###,.02
0.020
? ##.#,2.36
2.4
```

+ This sign may be used at either the beginning or end of the numeric field, and will force the + sign to be printed at either end of the field as specified, if the number is positive. If it is used at the end of the field, and the number is negative, a - sign will be forced at the end of the number.

- The - sign when used at the end of the numeric field designation will force the sign to be trailing the number, if it is negative. If the number is positive, a space is printed.

NOTE: There are cases where forcing the sign of a number to be printed on the trailing side will free an extra space for leading digits. (See exponential format).

** The ** placed at the beginning of a numeric field designation will cause any unused spaces in the leading portion of the number printed out to be filled with asterisks. The ** also specifies positions for 2 more digits. (Termed "asterisk fill").

$$ When the $$ is used at the beginning of a numeric field designation, a $ sign will be printed in the space immediately preceding the number printed. Note that the $$ also specifies positions for two more digits, but the $ itself takes up one of these spaces. Exponential format cannot be used leading $ signs, nor can negative numbers be output unless the sign is forced to be trailing.

**$ The **$ used at the beginning of a numeric field designation causes both of the above ** and $$ to be performed on the number being printed out. All of the previous conditions apply, except that **$ allows for 3 additional digit positions, one of which is the $ sign.

, A comma appearing to the left of the decimal point in a numeric field designation will cause a comma to be printed every three digits to the left of the decimal point in the number being printed out. The comma also specifies digit position. A comma to the right of the decimal point in a numeric field designation is considered a part of the string itself and will be treated as a printing character.

^^^^ Exponential Format. If the exponential format of a number is desired in the print out, the numeric field designation should be followed by ^^^^ (allows space for E+-XXX). As with the other formats, any decimal point arrangement is allowed. In this case, the significant digits are left justified and the exponent is adjusted.

% If the number to be printed out is larger than the specified numeric field, a % character will be printed and then the number itself in its standard format. (The user will see the entire number). If rounding a number causes it to exceed the specified field, the % character will be printed followed by the rounded number. (Such a field= .##, and the number is .999 will print %1.00.)

If the number of digits specified exceeds 24, a FUNCTION CALL error will occur.

Try going through the following example to help illustrate the preceding rules. A single program such as follows is the easiest method for learning PRINT USING.

Examples: Type the short program into your machine as it is listed below. This program will keep looping and allow you to experiment with PRINT USING as you go along.

Program:

```
10 INPUT A$,A
20 PRINT USING A$;A
30 GOTO 10
RUN
```

The computer will start by typing a ?. Fill in the numeric field designator and value list as desired, or follow along below.

```
? +#,9
+9
? +#,10
%+10
```

```
? ##,-2
-2
? +#,-2
-2
? #,-2
%-2
? +.###,.02
+0.02
? ####.#,100
100.0
? ##+,2
2+
? THIS IS A NUMBER ##,2
THIS IS A NUMBER 2
? BEFORE ## AFTER,12
BEFORE 12 AFTER
? ####,44444
%44444
? **##,1
***1
? **##,12
**12
? **##,123
*123
? **##,1234
1234
? **##,12345
%12345
? **,1
*1
? **,22
22
? **.##,12
12.00
? **####,1
*****1
```

(note: not floating $)

```
? $####.##,12.34
$  12.34
```

(note: floating $)

```
? $$####.##,12.56
   $12.56
? $$.##,1.23
$1.23
? $$.##,12.34
%$12.34
? $$###,0.23
   $0
? $$####.##,0
    $0.00
? **$###.##,1.23
****$1.23
? **$.##,1.23
*$1.23
? **$###,1
****$1
```

```
? #,6.9
7
? #.#,6.99
7.0
? ##-,2
 2
? ##-,-2
 2-
? ##+,2
 2+
? ##+,-2
 2-
? ##^^^^,2
 2E+00
? ##^^^^,12
 1E+01
? #####.###^^^^,2.45678
 2456.780E-03
? #.###^^^^,123
0.123E+03
? #.##^^^^,-123
-.12E+03
? "#####,###.#",1234567.89
1,234,567.9
```

BASIC ERROR MESSAGES-ALPHABETIC LISTING

(See Page 96 for Numeric Listing)

ERROR		
05	BAD DATA FORMAT	The DATA format does not match the READ statement.
35	BAD DISK NUMBER	The disk number is not 0,1,2 or 3.
28	BAD FILE MODE	Attempt to perform a PRINT to a random file, or to perform a PUT or GET on a sequential file. An OPEN statement where the file mode is not I, O, or R.
36	BAD FILE NAME	A file name of 0 characters (null) or a file name whose first byte was 0 or 377 octal (255 decimal) or a file name with more than 5 characters was used as an argument to LOAD, SAVE or OPEN.
26	BAD FILE NUMBER	An attempt was made to use a file number which specifies a file that is not OPEN.
34	BAD RECORD NUMBER	PUT or GET statement, record number is either greater than allowable maximum (4095) or less than zero.
00	BREAK	Break detected.
16	BREAK DURING STRING COMPACTION	A break was detected during string compaction, type CONT so that operation is finished.
19	CAN'T CONTINUE	Attempt to continue a program after modification or an error, or before it was RUN.
38	DIRECT STATEMENT IN FILE	A direct statement was encountered during a LOAD of a program in TEXT format. The load is terminated.

32 41	DISK FULL or OUT OF DISK SPACE	All disk storage is exhausted on the disk. Delete some old disk files and retry.
12	DIVISION BY ZERO	
24	FIELD OVERFLOW	Attempt to allocate more than 255 characters worth of string variables in a single FIELD statement.
29	FILE ALREADY OPEN	A sequential output mode OPEN for a file was issued for a file that was already OPEN and had never been CLOSED.
27	FILE NOT FOUND	Reference was made in a LOAD, or OPEN statement to a file which did not exist on the disk specified.
33	INPUT PAST END	An INPUT statement was executed after all the data in a file had been INPUT. This will happen immediately if an INPUT is executed or a null (empty) file. Use of the EOF function to detect End of File (EOF) will avoid this error.
25	INTERNAL ERROR	Internal error in BASIC. Report conditions under which error occurred to the DTC software department, along with all relevant data. This error can also be caused by certain kinds of disk I/O errors.
30	I/O ERROR	A disk error occurred during the last I/O statement.
13	ILLEGAL DIRECT	You cannot use an INPUT or DEF FN statement as a direct command.
06	ILLEGAL FUNCTION CALL	The parameter passed to a math or string function was the wrong type or out of range.
42	LOAD FILE VERSION MISMATCH	An attempt was made to LOAD a program

		file (type B) SAved under a previous version of BASIC.
37	MODE-MISMATCH	
01	NEXT WITHOUT FOR	A NEXT statement was encountered but a FOR was active.
21	NO RESUME	An error or break detection processing trap without a RESUME statement.
04	OUT OF DATA	A READ or file INPUT statement was executed but no more data is available.
08	OUT OF MEMORY	Program too large, too many variables, too many FOR loops, too many GOSUBS, too complicated an expression or any combination of the above.
15	OUT OF STRING SPACE	Not enough string space was allocated. CLEAR with a large value.
07	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.
39	PROTECTED FILE	An attempt was made to LIST, EDIT, or SAVE a protected program.
11	REDIMENSIONED ARRAY	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 of 10 because a statement like A(I)=3 is encountered and then later in the program DIM A(100) is found.
22	RESUME WITHOUT ERROR OR BREAK	An error trapping subroutine was entered without an error or break in process.

03	RETURN WITHOUT GOSUB	A RETURN statement was encountered without a previous GOSUB being executed.
31	SET TO NON DISK STRING	An LSET or RSET was given for a string variable which had not previously been mentioned in a FIELD statement.
18	STRING FORMULA TOO COMPLEX	A string expression was too complex. Break it into two or more shorter ones.
17	STRING TOO LONG	An attempt was made by use of the concatenation operator to create a string more than 255 characters long.
10	SUBSCRIPT OUT OF RANGE	An attempt was made to reference a matrix element with a subscript out of bounds or with the wrong number of subscripts.
02	SYNTAX ERROR	The statement is not a legal BASIC statement. Missing parentheses, illegal character, incorrect punctuation, etc. Unless this is a protected program, EDIT is entered for the offending line. The start of EDIT is indicated by the line number typed out. If a function was called (FNx) in the line, the error may lie in the function definition.
40	TOO MANY FILES	An OPEN was attempted for a file number >=8 or >0.
14	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 argument was the wrong type.
09	UNDEFINED STATEMENT	An attempt was made to GOTO, GOSUB, THEN, or ELSE to a statement that does not exist.
20	UNDEFINED USER FUNCTION	Reference was made to a user function with an undefined address.

23	UNPRINTABLE ERROR	An error was detected without an error description to match it from within the existing error statements.

LIST OF ASSIGNED ERROR CODES IN BASIC 3.2

ERROR NUMBER	MESSAGE
0	BREAK
1	NEXT WITHOUT FOR
2	SYNTAX ERROR
3	RETURN WITHOUT GOSUB
4	OUT OF DATA
5	BAD DATA FORMAT
6	ILLEGAL FUNCTION CALL
7	OVERFLOW
8	OUT OF MEMORY
9	UNDEFINED STATEMENT
10	SUBSCRIPT OUT OF RANGE
11	REDIMENSIONED ARRAY
12	DIVISION BY ZERO
13	ILLEGAL DIRECT
14	TYPE MISMATCH
15	OUT OF STRING SPACE
16	BREAK DURING STRING COMPACTION
17	STRING TOO LONG
18	STRING FORMULA TOO COMPLEX
19	CAN'T CONTINUE
20	UNDEFINED USER FUNCTION
21	NO RESUME
22	RESUME WITHOUT ERROR OR BREAK
23	UNPRINTABLE ERROR
24	FIELD OVERFLOW
25	INTERNAL ERROR
26	BAD FILE NUMBER
27	FILE NOT FOUND
28	BAD FILE MODE
29	FILE ALREADY OPEN
30	I/O ERROR
31	SET TO NON-DISK STRING
32	DISK FULL
33	INPUT PAST END
34	BAD RECORD NUMBER
35	BAD DISK NUMBER
36	BAD FILE NAME
37	MODE-MISMATCH
38	DIRECT STATEMENT IN FILE
39	PROTECTED FILE
40	TOO MANY FILES
41	OUT OF DISK SPACE
42	LOAD FILE VERSION MISMATCH

EDIT COMMAND

The EDIT command is for the purpose of allowing modifications and additions to be made to existing program lines without having to retype the entire line each time.

Commands typed in the EDIT mode are, as a rule, echoed. Most commands may be preceded by an optional numeric repetition factor which may be used to repeat the command a number of times. This repetition factor should be in the range 0 to 255 (0 is equivalent to 1). If the repetition factor is omitted, it is assumed to be 1. In the following examples a lower case "n" before the command stands for the repetition factor.

In the following description of the EDIT commands, the "cursor" refers to a pointer which is positioned at a character in the line being edited.

To EDIT a line, type EDIT followed by the number of the line and hit the carriage return. The line number of the line EDITed will be printed, followed by a space. The cursor will now be positioned to the left of the first character in the line.

NOTE: The best way of getting the "feel" of the EDIT command is to try EDITing a few lines yourself. Commands not recognized as part of the EDIT commands will be ignored.

MOVING THE CURSOR

A space typed in will move the cursor to the right and cause the character passed over to be printed out. A number preceding the space (nS) will cause the cursor to pass over and print out the number (n) of characters chosen.

INSERTING CHARACTERS

I Inserts new characters into the line being edited. After the I is typed, each character typed in will be inserted at the current cursor position and typed on the terminal. To stop inserting characters type "escape" (or ALT mode on some terminals).

If an attempt is made to insert a character that will make

the line longer than the maximum allowed (255 characters), a bell will be typed (control G) on the terminal and the character will not be inserted.

(or) A backspace (or underline) typed during an insert command will delete the character to the left of the cursor. Characters up to the beginning of the line may be deleted in this manner, and a backspace will be echoed for each character deleted. However, if no characters exist to the left of the cursor, a bell is echoed instead of a backspace.

The "backspace" character used is the same as specified by the MICROFILE CWL command, and may be changed if desired.

If a carriage return is typed during an insert command, it will be as if an escape and then carriage return was typed. That is, all characters to the right of the cursor will be printed and the EDITed line will replace the original line.

X X is the same as I, except that all characters to the right of the cursor are printed, and the cursor moves to the end of the line. At this point it will automatically enter the insert mode (see I command).

X is very useful when you wish to add a new statement to the end of an existing line. For example:

```
Typed by User         EDIT 50          (carriage return)
Typed by MICROFILE    50 X=X+1:Y=Y+1
Typed by User           X    :Y=Y+1    (carriage return)
```

In the above example, the original line #50 was:

```
50 X=X+1
```

The new EDITed line #50 will now read:

```
50    X=X+1:Y=Y+1
```

H H is the same as I, except that all characters to the

right of the cursor are deleted (they will not be typed). The insert mode (see I command) will then automatically be entered.

H is most useful when you wish to replace the last statements on a line with new ones.

DELETING CHARACTERS

D

nD deletes n number of characters to the right of the cursor. If less than n characters exist to the right of the cursor, only that many characters will be deleted. The cursor is positioned to the right of the last character deleted. The characters deleted are enclosed in backslashes (\). For example:

```
Typed by User          20 X=X+1:REM JUST INCREMENT X
Typed by User          EDIT 20  (carriage return)
Typed by MICROFILE     20   \X=X+1:\REM JUST INCREMENT X
Typed by User              6D (carriage return)
```

The new line #20 will no longer contain the characters which are enclosed by the backslashes.

SEARCHING

S

The nSy command searches for the nth occurance of the character y in the line. The search begins at the character one to the right of the cursor. All characters passed over during the search are printed. If the character is not found, the cursor will be at the end of the line. If it is found, the cursor will stop at that point and all of the characters to its left will have been printed. For example:

```
Typed by User          50 REM INCREMENT X
Typed by User          EDIT 50
Typed by MICROFILE     50 REM INCR
Typed by User          2SE
```

K

nKy is equivalent to S, except that all of the characters passed over during the search are deleted. The deleted characters are enclosed in backslashes. For example:

```
Typed by User          10 TEST LINE
Typed by User          EDIT 10
Typed by MICROFILE     10  \TEST\
Typed by User           KL
```

TEXT REPLACEMENT

C — A character in a line may be changed by the use of the C command. Cy, where y is some character, will change the character to the right of the cursor to y. The y will be typed on the terminal and the cursor will be advanced one position. nCy may be used to change n number of characters in a line as they are typed in from the terminal. (See example below.)

If an attempt is made to change a character which does not exist, the change mode will be exited. Example:

```
Typed by User          10 FOR I=1 TO 100
Typed by User          EDIT 10
Typed by MICROFILE     10 FOR I=1 to 256
Typed by User          2S1      3C256
```

ENDING AND RESTARTING

Carriage Return — Tells the computer to finish and print the remainder of the line. The edited line replaces the original line.

E — E is the same as a carriage return, except the remainder of the line is not printed.

Q — Quit. Changes to a line do not take effect until an E or carriage return is typed. Q allows the user to restore the original line without any changes which may have been made, if an E or carriage return has not yet been typed. "OK" will be typed and BASIC will await further commands.

L — Causes the remainder of the line to be printed, and then prints the line number and restarts EDITing at the beginning of the line. The cursor will be positioned to the left of the first character in the line.

L is most useful when you wish to see how the changes in a line look so that you can decide if further EDITs are necessary. Example

```
Typed by User         EDIT 50
Typed by MICROFILE    50 REM  INCREMENT X
Typed by User         2SM           L
Typed by MICROFILE    50
```

A — Causes the original copy of the line to be restored, and EDITing to be restarted at the beginning of the line. For example:

```
Typed by User         10 TEST LINE
Typed by User         EDIT 10
Typed by MICROFILE    10 \TEST LINE\
Typed by User         10D           A
Typed by MICROFILE    10
```

In the above example, the user made a mistake when he deleted TEST LINE. Suppose that he wants to type "1D" instead of "10D". By using A command, the original line 10 is re-entered and is ready for further EDITing.

IMPORTANT

Whenever a SYNTAX ERROR is discovered during the execution of a source program, BASIC will automatically begin EDITing the line that caused the error as if an EDIT command had been typed. For example:

```
10 APPLE
RUN
SYNTAX ERROR IN 10
10
```

Complete editing of a line causes the line edited to be re-inserted. Re-inserting a line causes all variable values to be deleted, therefore you may want to exit the EDIT command without correcting the line so that you can examine the variable values.

This can be easily accomplished by typing the Q command while in the EDIT mode. If this is done, BASIC will type OK and all variable values will be preserved.

EXPANDED ASSEMBLY LANGUAGE FEATURES

(DEF USR)

BASIC has the facility to call up to 10 different assembly language subroutines, numbered USR0-USR9. (USR is equivalent to USR0).

A statement, DEF USR, has been provided to allow the user to specify the starting address in memory of any of his assembly language routines. Statement:

DEF USR{<digit 0 through 9>}=<numeric formula>

Example:

```
DEF USR1=&100000
DEF USR2=31096
DEF USR9=ADR
```

The numeric formula specifies the starting address of the USR routine specified.

Another important feature is the facility to pass string arguments, integer arguments, single precision arguments to a USR routine. When the USR subroutine is entered, the [H,L] register pair contains a pointer to the floating point accumulator where all arguments are stored, and the A register contains the value type.

The Floating Accumulator (FAC) has the following format:

Single (Arg=4) and Double Precision (Arg=8) Values.

```
[HL]     Exponent
[HL]-1   Sign and bits 2-8 of mantissa
[HL]-2   Bits 9-16 of mantissa
[HL]-3   Bits 17-24 of mantissa
[HL]-4   Low order mantissa bytes [Double Precision]
[HL]-5   Low order mantissa bytes [Double Precision]
[HL]-6   Low order mantissa bytes [Double Precision]
[HL]-7   Low order mantissa bytes [Double Precision]
```

Floating point quantities are always normalized so that bit 1 of the

mantissa is 1. The mantissa is always positive, regardless of the sign. The binary point is to the left of bit 1 (fractional representation). The sign is where bit 1 of the mantissa ought to be.

The exponent is a biased signed value. That is, the indicated exponent is the true exponent (-127 to +127) plus 128. An indicated exponent of 0 means the value is zero.

Integer Values: (Arg=2)

```
[HL]-2   Most significant byte  -  signed 16 bit value
[HL]-3   Least significant byte -  signed 16 bit value
```

String Values: (Arg=3)

```
[HL]-2   Most significant byte  -  address of string
         descriptor
[HL]-3   Least significant byte -  address of string
         descriptor
```

String Descriptors have the following form:

```
[Desc]   String size (0-255)
[Desc]+1 Least significant byte  - address of string
[Desc]+2 Most significant byte   - address of string
```

Upon return, the USR function must leave a value in the FAC of the same type.

LOADING USR ROUTINES

User routines can be loaded either below BASIC or above BASIC. Below BASIC, between 2700H and 27FFH, is the area from which MON commands are executed. Loading routines there precludes use of most MON commands. The "LO" command may be used to load the user subroutine.

Loading above BASIC requires that the stack be moved, if BASIC is not yet running. To load above BASIC with BASIC running:

1. Execute CLEAR <size> with enough size to hold programs.

2. Execute MON "PA 2803 yy xx" where the address xxyyH is the new ceiling for BASIC. It must be in the reversed area and below your programs.

3. Execute CLEAR <string size> to reserve string space with respect to the new ceiling. The parameter MUST be present to force BASIC to reset its stack pointers.

4. Now load the programs with MON "LO file".

The stack is maintained within BASIC, and only a limited amount of space may be available, depending on program and non-string variable size. A direct jump to location 2800H can be used to return BASIC to command mode, and clear all variables.

DERIVED FUNCTIONS

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

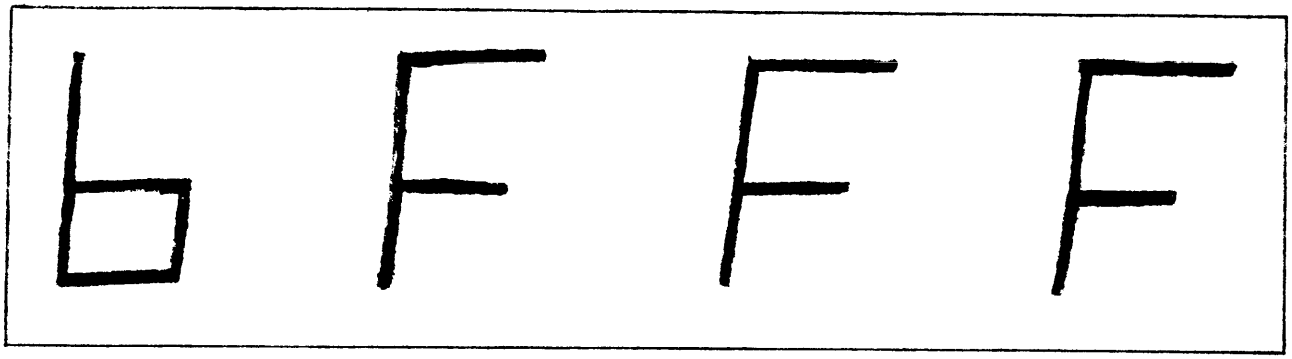
Function	Function Expressed in terms of BASIC functions
SECANT	SEC(X) = 1/COS(X)
COSECANT	CSC(X) = 1/SIN(X)
COTANGENT	COT(X) = 1/TAN(X)
INVERSE SINE	ARCSIN(X) = ATN(X)/SQR(-X*X+1))
INVERSE COSINE	ARCCOS(X) = -ATN(X/SQR(-X*X+1))+1.5708
INVERSE SECANT	ARCSEC(X) = ATN(SQR(X*X-1))+(SGN(X)-1)*1.5708
INVERSE COSECANT	ARCCSC(X) = ATN(1/SQR(X*X-1))+(SGN(X)-1)*1.5708
INVERSE COTANGENT	ARCCOT(X) = -ATN(X)+1.5708
HYPERBOLIC SINE	SINH(X) = (EXP(X)-EXP(-X))/2
HYPERBOLIC COSINE	COSH(X) = (EXP(X)+EXP(-X))/2
HYPERBOLIC TANGENT	TANH(X) = -EXP(-X)/(EXP(X)+EXP(-X))*2+1
HYPERBOLIC SECANT	SECH(X) = 2/(EXP(X)+EXP(-X))
HYPERBOLIC COSECANT	CSCH(X) = 2/(EXP(X)-EXP(-X))
HYPERBOLIC COTANGENT	COTH(X) = EXP(-X)/(EXP(X)-EXP(-X))*2+1

INVERSE HYPERBOLIC SINE	ARGSINH(X) = LOG(X+SQR(X*X+1))
INVERSE HYPERBOLIC COSINE	ARGCOSH(X) = LOG(X+SQR(X*X-1))
INVERSE HYPERBOLIC TANGENT	ARGTANH(X) = LOG((1+X)/(1-X))/2
INVERSE HYPERBOLIC SECANT	ARGSECH(X) = LOG((SQR(-X*X+1)+1)/X)
INVERSE HYPERBOLIC COSECANT	ARGCSCH(X) = LOG((SGN(X)*SQR(X*X+1)+1)/X)
INVERSE HYPERBOLIC COTANGENT	ARGCOTH(X) = LOG((X+1)/(X-1))/2

HELP

If you drop out of BASIC back to MICROFILE's system monitor (the * prompt signifies this condition) you can return to BASIC without losing your program in memory by typing GO 2800. BASIC will respond with OK.

HEXADECIMAL
MICROFILE DISPLAY CONVERSION



Hex.	Dec.	Hex.	Dec.	Hex.	Dec.	Hex.	Dec.
0	0	0	0	0	0	0	0
1	4096	1	256	1	16	1	1
2	8192	2	512	2	32	2	2
3	12288	3	768	3	48	3	3
4	16384	4	1024	4	64	4	4
5	20480	5	1280	5	80	5	5
6	24576	6	1536	6	96	6	6
7	28672	7	1792	7	112	7	7
8	32768	8	2048	8	128	8	8
9	36864	9	2304	9	144	9	9
A	40960	A	2560	A	160	A	10
B	45056	B	2816	B	176	B	11
C	49152	C	3072	C	192	C	12
D	53248	D	3328	D	208	D	13
E	57344	E	3584	E	224	E	14
F	61440	F	3840	F	240	F	15

This chart may be used to convert the Microfile hexadecimal display to a decimal value. Simply add the decimal value of the hexadecimal figure from its associated column with the other column values. For example the display above shows a hex address of BFFF or (b=45056)(F=3840)(F=240)(F=15)=49151

INDEX