M T U B A S I C L A N G U A G E I N T E R P R E T E R

R E F E R E N C E M A N U A L

FEBRUARY, 1982 REV. B

TABLE OF CONTENTS

1. INTRODUCTION

At the heart of MTU BASIC is the standard Microsoft BASIC interpreter for the 6502. This interpreter has been extensively modified to add new commands and adapt the existing ones which must interact with the operating system, such as LOAD and SAVE. The remaining commands have not been changed, and should operate identically to other implementations of Microsoft BASIC.

One of the new features contained in MTU BASIC is that it doesn't communicate with the keyboard and display directly. Instead it communicates via CODOS channels. This means that BASIC doesn't know if it is being driven by the user from the console device, or from text in a file or from some other input device. It also doesn't know where its output is being directed. This makes it simple to run MTU BASIC from a CODOS job file.

To directly support the use of the function keys and legend display, the LEGEND command and reserved variable KEY have been added. These make it simple to write function key driven applications.

Perhaps the most significant feature added is the ability of MTU BASIC to accept up to 8 external command enhancement programs, which we call Library Programs. Each of these programs adds a set of its own commands to BASIC's standard set, as well as the routines to execute the new commands. This provides a much more satisfactory interface between machine language and BASIC than the standard MCALL and USR type commands available in other BASICs.

2. GETTING STARTED WITH BASIC

2.1 STARTUP PROCEDURE

If your computer does not have MTU BASIC loaded and running, follow the procedures in chapter 4 of the SET UP AND INSTALLATION section of this manual. When your I/O device types "READY", you are ready to use MTU BASIC. Note that all commands to BASIC end with a carriage return. The carriage return tells BASIC that you have finished typing the command.

The method for describing the syntax of various commands and arguments in the CODOS section of this manual is also used with MTU BASIC. Angle brackets enclose words describing the kind of entry required. Square brackets enclose optional arguments or symbols. Ellipsis are used to indicate an arbitrary number of repetitions of the previous argument(s). Symbols not enclosed in angle brackets are literal symbols which must be typed exactly as shown. Braces are used to enclose each of several mutually exclusive choices, only one of which may be selected.

MTU BASIC is entered by executing a command with the following syntax:

BASIC [<top of memory>] [,<terminal width>]

where

<top of memory> = the first high memory location not to be used by BASIC expressed as a decimal number. Defaults to 48460 (BE00 hex).

<terminal width> = the number of printing characters that may be output before an automatic carriage return occurs. Defaults to 192.

If the top of memory parameter is specified, both the current top of memory and the default top of memory will be set to the specified location. If not specified, both default to 48460 (BE00 hex). The difference between these two tops of memory is that the current top of memory will be moved down as Library Programs are linked into BASIC. The current top of memory will move back up to the default top of memory when the libraries are freed. At no time will BASIC use any memory at or above the default top of memory, not even for library programs. To change the default top of memory you must exit and reenter BASIC with the BASIC command.

The terminal width parameter controls how many printing characters may be output before an automatic carriage return occurs. Based on the terminal width, the last legal tab stop is determined. Since the Console device under CODOS does its own automatic carriage return, it isn't necessary to specify 80 as the terminal width parameter. The terminal width defaults to 192.

2.2 PRINT STATEMENT

Now, try typing in the following:

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

BASIC will immediately print:

```
6
READY.
```

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

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

BASIC will print:

```
 .5          30
READY.
```

As you can see, BASIC can do division and multiplication as well as subtraction. Note how a comma was used in the PRINT command to print two values instead of just one. The comma divides the line into fields, each 10 characters wide. The comma causes BASIC to skip to the next field on the line, where the value 30 was printed.

2.3 PROGRAMS AND LINE NUMBERS

Commands such as the PRINT statement 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 may be any integer from 0 to 63999. Try typing in the following lines:

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

Notice that BASIC did not print either of the values. That is because BASIC recognizes by the line numbers that these lines are to be saved and executed as a sequence later. A sequence of indirect commands is called a program. BASIC saves indirect commands in memory. When you type RUN, BASIC executes the program, beginning with the lowest-numbered line. If we type RUN now, BASIC will type out:

```
5
-1
READY.
```

In the above example, we entered lines 10 and 20 in numerical order. However, it makes no difference in what order you enter indirect statements. BASIC always puts them into correct numerical order according to line number. To see a listing of the complete program currently in memory, type LIST. BASIC will reply with:

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

Sometimes it is desirable to delete a line of a program altogether. This is accomplished by typing the line number to be deleted, followed only by a carriage return. Type in the following:

```
10
LIST
```

BASIC will reply with:

```
20 PRINT 2-3
READY.
```

We have now deleted line 10 from the program. There is no way to get it back except to retype it. To insert a new line 10, type in 10 followed by the new line. Type in the following:

```
10 PRINT 2*3
LIST
10 PRINT 2*3
20 PRINT 2-3
READY.
```

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

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

BASIC will reply with:

```
10 PRINT 3-3
20 PRINT 2-3
READY.
```

It is not recommended that lines be numbered in small increments. It may become necessary to insert a new line between two existing lines. An increment of 10 between line numbers is usually 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 type in a new one, be sure to type NEW first. This should be done in order to prevent a mixture of the old and new programs.

Type in the following:

```
NEW
```

BASIC will reply with:

```
READY.
```

Now type in:

```
LIST
```

BASIC will reply with:

```
READY.
```

2.4 NUMBERS, OPERATORS, AND STRINGS

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

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

BASIC will reply with:

```
ONE THIRD IS EQUAL TO          .333333333
READY.
```

As explained earlier, including a comma in a PRINT statement causes it to space over to the next field before the value is printed. Using a semicolon instead of a comma causes the value to be printed immediately, without spacing to the next field. Note that numbers are always printed with at least one trailing space. Any text to be printed must always be enclosed in double quotes. Try the following examples:

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

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

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

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

We will digress for a moment to explain the format of numbers in BASIC. Floating point numbers are stored internally to over nine digits of accuracy. When a number is printed, only nine digits are shown. Any number may also have an exponent.

The largest number that may be represented in MTU BASIC is 1.70141*10E38, while the smallest positive number is 2.93874*10E-39.

When a floating point number is printed, the following rules are used to determine the exact format.

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

2. If the absolute value of the number is an integer in the range 0 to 999999999, 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 999999999, it is printed in fixed notation, with no exponent.

.01

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

Scientific notation is formatted as follows:

```
SX.XXXXXXXXESTT.
```

Each X is an integer, 0 to 9. The leading S is the sign of the number; a space for a positive number and a minus sign for a negative number. One non-zero digit is printed before the decimal point. This is followed by the decimal point and then the other eight digits of the mantissa. An E is then printed (for exponent), followed by the sign (S) of the exponent; then the two digits (TT) of the exponent itself. Leading zeros are never printed; i.e., the digit before the decimal is never zero. Also, trailing zeros are never printed. If there is only one digit to print after all trailing zeros 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, zeros are not suppressed in the exponent field. The value of any number expressed thus is the number to the left of the E times 10 raised to the power of the number to the right of the E.

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

The following are examples of various numbers and the output format BASIC will use:

NUMBER	OUTPUT
+1	+1
-1	-1
6523	6523
-23.460	-23.46
1E20	1E+20
-12.3456789E-7	-1.23456789E-06
1.23456789E-10	1.23456789E-10
1000000	1E+06
999999999	999999999

A number input from the terminal or a numeric constant used in a BASIC program may have as many digits as desired, up to a maximum of 40. However, only the first 9 digits are significant in MTU BASIC. The final digit is rounded up.

```
PRINT 1.23456789
READY.
```

So far we have used several different operators in order to inform BASIC of the calculations we wish to perform. Whenever a combination of these operators is used, it is necessary to know which operations are to be performed first. As in standard algebra, we can either specify which operations have the highest priority, or we can rely on BASIC's precedence of operators. That precedence is as follows:

Priority of Operations

1. Parentheses -- any expression enclosed in parentheses is always evaluated first.

2. Exponentiation

3. Negation

4. Multiplication and Division (of equal priority)

5. Addition and Subtraction (of equal priority)

6. Relational operators (all of equal priority)

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

7. Logical Operators in the order NOT, AND, then OR.

2.5 INPUT AND GOTO STATEMENTS

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
READY.
```

Here's what is happening. When BASIC encounters the INPUT statement, it types a question mark on the terminal and then waits for a response. When you type in 10 followed by a carriage return , execution continues with the next statement in the program, with the variable R set to 10. In executing line 20, BASIC prints 314.159.

As you can see, the program calculates the area of a circle with radius R. To calculate the area of different circles, we could keep rerunning the program over for each value of R. But, there's an easier way to do this; simply add another line to the program as follows:

```
30 GOTO 10
RUN
?10
314.159
?3
28.37431
?4.7
69.3977231
?
READY.
```

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. This could have gone on indefinitely, but we decided to stop after calculating the area for three circles. This was accomplished by typing a carriage return to the input statement (thus a blank line).

2.6 VARIABLES

The letter R in the program we just executed was termed a variable. A variable name consists of one or two characters, and the first character must be a letter. The second character may be a letter or a number (0-9). Any characters after the

first two are ignored.

Variable names may not be the same as BASIC reserved words, nor may they contain BASIC reserved words. Reserved words are those words used as BASIC commands, statements, or functions. For example, TO would be an illegal variable name and FEND would be illegal because it contains the reserved word END.

Here is a list of MTU BASIC reserved words which must not be contained in variable names:

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

Integer variables are very useful for saving memory space. A floating point number requires 5 bytes of memory space for storage. One byte stores the exponent of the number, and the other four bytes are used to store the mantissa. BASIC requires only 2 bytes of memory to store integers, because of their smaller range +32767 to -32767.

Integer variables are distinquished from numeric variables by a "%" after the variable name. You may assign values to an integer variable in the same way that you assign values to floating point variables. For example:

```
X% = 6
D4% = 5*3
```

If you assign a non-integer value to an integer variable, the number will be truncated that is, any value to the right of a decimal point will be lost. For example:

```
B% = 4.883
PRINT B%
4
READY.
```

During the program execution, when BASIC encounters a calculation to be done with an integer variable, these steps are followed:

1. BASIC retrieves the variable from memory.
2. The number is converted into its floating point equivalent.
3. The calculation is performed.
4. The number is converted back into an integer.
5. The result is stored in memory.

If you use integer values in a program and wish to get accurate results from calculations, you must be very careful to ensure that values are not truncated. When you define a variable as an integer, be certain that it will never receive a floating point number that you may wish to recall. Although integer variables can

save significant amounts of memory space, their use may cause some loss of execution speed.

2.7 LET STATEMENT

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

Try the following examples:

```
A=5
READY.
PRINT A,A*2
5     10
READY.
LET Z=7
READY.
PRINT Z,Z-A
7     2
READY.
```

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 memory to store data. The values of variables are thrown away and the space in memory used to store them is released when one of four things occurs:

1. A new line is typed into the program or an old line is deleted
2. A CLEAR is executed
3. A RUN is executed
4. A NEW is executed

Another important fact is that if a variable is encountered in an expression 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 expression. For example:

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

2.8 REM STATEMENT

REM is short for remark. This statement is used to insert comments or notes into a program. These comments can help avoid confusion--especially when you write long programs. They are also helpful to anyone else who reads your program.

REM statements often note the purpose of a program and describe the techniques used in accomplishing that purpose. REM statements are useful only to humans. No information is passed to BASIC through the notes in a REM statement. When a REM is encountered during execution of a BASIC program, everything on the rest of the line is ignored.

2.9 IF AND THEN STATEMENTS; RELATIONAL OPERATORS

Suppose we want to write a program to find out 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 and THEN statements do just that. Try typing in the following program. (Remember, type NEW first.)

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

When this program is typed into the computer and run, it will ask for a value for B. Type any value you wish. The computer will then come to the IF statement. Between the IF and THEN portion of the statement there are two expressions separated by a relational operator. A relational operator is one of the following six symbols:

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

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 entered, if 0 were 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 were typed in for B. Since 1=0 is false, the IF statement would be false and the THEN clause would not be executed. 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 receives 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 20, if A is smaller than B, A<=B is true so we go to line 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 sample programs several times. Remember, to stop these programs, just give a carriage return to the input request. Then try writing a few programs of your own using the IF-THEN statement. Writing programs of your own is the quickest and easiest way to understand how BASIC works.

2.10 FOR AND NEXT STATEMENTS

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, and the form is SQR(x). (For more information about functions, see 1.11.) We could write the program as follows:

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

This program will do the job. But imagine how time-consuming this would be if we wanted to make a table of 100 square roots, or 1000 square roots! The inefficient method above could be improved by using the IF statement 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 ten-statement program above. Let's look at how it works. Line 10 has a LET statement which sets the value of the variable N to 1. Line 20 prints N and the square root of N using its current value. Line 30 is a LET statement that sets the value of N to N+1. In a LET statement, an equals sign means "to be replaced with".

Thus, the first time line 30 is executed, N becomes 2. At line 40, since N now equals 2, N $\leq$10 is true, so the THEN portion branches 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 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. But notice that the first method we used would require 1000 statements to create a table of 1000 square roots, whereas the final example could do it with just 3 lines.

At line 10, N is set to equal 1. Line 20 causes the value of N and the square root of N to be printed. At line 30 we see a new type of statement. The NEXT N statement causes 1 to be added to N, and then if N =10, the program returns to the statement following the FOR statement. The overall effect is the same as with the previous program. Notice that the variable following the FOR is exactly the same as the variable after the NEXT.

It is also possible to add a STEP clause to the FOR statement to change the increment of the FOR loop variable.

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

This tells BASIC to start with N=10 and to add 2 to N each time, instead of 1 as in the previous program. If no STEP is given in a FOR statement, BASIC assumes that 1 is to be added each time. The STEP can be followed by any expression.

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. For example:

```
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" the I-loop.

2.11 ARRAYS -- DIM STATEMENT

An array is a table of values. The name of this table, which is called the array name, can be any legal variable name, such as "A". The array name A is distinct and separate from the simple variable A, and both can be used in a single program. To identify an element of the array, a subscript is added in parentheses to the array name. A(I) represents the Ith element in the array A.

BASIC must be told how much space to allocate for the entire array A. This is

done with a DIM (dimension) statement, for example,

```
DIM A(15)
```

In this example, space is reserved for the array A to have elements from A(0) to A(15). Array subscripts always start with zero. Therefore, an array dimensioned at 15 has space for 16 elements.

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

The following program illustrates the use of arrays. It asks you to input 8 numbers, which it then sorts into ascending order.

```
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 sort is accomplished by comparing adjacent numbers. If they are not ordered, their positions in the array are reversed. If any pairs are reversed, the variable "F" is incremented, and, at line 150, BASIC is told to repeat the comparison process. If all the numbers are in order, lines 160 through 180 print out the sorted list. Notice that a subscript can be any expression, as illustrated in line 110. Arrays may also be dimensioned as integer arrays. If all elements of an array can be represented by integers, a significant amount of memory space can be saved. An integer array is dimensioned exactly like a numeric array, except that a "%" must follow the array name. For example:

```
DIM A%(9)
```

This array will have 10 elements, each one an integer. As with other integer variables, BASIC will perform all computations in floating point arithmetic. See 1.5.

2.12 INTRINSIC FUNCTIONS

BASIC provides functions to perform a number of standard mathematical operations. These functions would be time-consuming for you to calculate and code in BASIC, especially if you used them more than once in your program. A BASIC function has a three- or four-letter call name followed by an argument in parentheses. A function may be used anywhere in a progam unless you have deleted it during initialization.

2.13 USER-DEFINED FUNCTIONS -- DEF STATEMENT

You may also define your own functions when you require a reasonably simple calculation at more than one place in your program. The DEF statement alllows you to create a function and use it in the same way that you would use any of BASIC's intrinsic functions. A legal user-defined function name begins with FN followed by any acceptable BASIC variable name. A function need be defined only once and can be defined anywhere in the program. For example:

```
DEF FNA(S)=S*S
```

The dummy argument must be a simple variable, but the expression after the equals sign may contain the argument variable or any other program variable. If, after defining the above function, BASIC encountered a line such as:

```
20 PRINT FNA(4)+5
```

BASIC would respond with:

```
21
READY.
```

First 4*4 was calculated using the formula we defined above. The 5 was added to this value.

2.14 SUBROUTINES -- GOSUB AND RETURN

A program may perform the same action in several different places. If this is expressed by a relatively simple calculation, you can probably use a BASIC function or a user-defined function to perform it. However, often it is a more complicated series of calculations requiring many lines of programming. In such cases the "GOSUB"and "RETURN" statements can be used to avoid entering the same lines over and over. When a "GOSUB" statement is encountered, BASIC branches to a specified line number. BASIC keeps track of the point at which it branched from the main program, and when a "RETURN" is encountered, BASIC goes back to the statement following the last executed "GOSUB". 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 "YOU MUST ENTER AN INTEGER. TRY AGAIN."
130 GOTO 100
140 RETURN
```

This program asks for two numbers which must be integers and then prints the sum of the two. The subroutine in this program consists of lines 100 to 130. The subroutine asks for a number, and if it is not an integer, asks for another number. It will continue to ask for numbers until an integer is entered.

The main program prints "WHAT IS THE NUMBER" and then calls the subroutine to determine if the number is an integer. 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. The program then prints "WHAT IS THE SECOND NUMBER" and the second value is entered when

the subroutine is called again. 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 the "STOP" statement. It causes the program to stop execution at line 90. If the STOP statement were not included in the program, we would "fall into" the subroutine at line 100. It would then ask that another number be input. When program execution reached line 130, the subroutine would try to return to the main body of the program. But since it was never called by a GOSUB, a return without GOSUB error in 140 would occur. Each GOSUB executed in a program must have a corresponding RETURN to be executed later. The opposite also 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 indicating the line at which the STOP was encountered.

2.15 READ, DATA, AND RESTORE STATEMENTS

Suppose that a program required that the same list of numbers be entered into the program every time it was run. BASIC contains two statements for accomplishing this purpose, the "READ" and "DATA" statements. These same statements also make it easy to change the list of numbers whenever necessary. Consider the following example:

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

When the program is run and the READ statement is encountered, the effect is similar to that of 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 be used. DATA is always read sequentially in this manner, and there may be any number of DATA statements in a program.

The above program plays a game whose object is to guess one of the numbers contained in the DATA statements. For each guess that is typed in, the program reads through all of the numbers in the DATA statements until one is found that matches the guess. If more values are read than there are numbers in the DATA statement, an out of data (OD) error occurs. Line 40 checks to see if -999999 was read. This number is used as a flag to indicate that all of the data has been read. If -999999 is read, then the guess given was incorrect.

Before returning to line 10 to make another guess, we need to make certain that the READSs will begin with the first piece of data again. This is the function of the "RESTORE" statement. After the RESTORE is encountered, the next piece of data

read will be the first piece in the first DATA statement.

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

2.16 STRINGS AND STRING FUNCTIONS

A list of characters is referred to as a "string". HELLO, MTU BASIC, and THIS IS A TEST are all strings. Like numeric variables, string variables can be assigned specific values. String variable names are distinquished from numeric variable names in that they have a"$" as the last character of the variable name. For example:

```
A$="MTU BASIC"
READY.
PRINT A$
MTU BASIC
READY.
```

This example assigns the string value "MTU BASIC" to the string variable A$. Note that a character string to be assigned to a variable must be enclosed in quotes.

Now that A$ has a string value, it is possible to use various string functions to learn more about that value. String functions are similar to numeric functions, and are called and passed arguments in the same way. One of these functions is the "LEN" or length function. When called, it returns an integer equal to the number of characters in a string. For example:

```
PRINT LEN(A$),LEN("HELLO")
9          5
READY.
```

The number of characters in a string 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, in much the same way that numeric variables are initialized to 0. 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. For example:

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

Another way to create the null string is by using quotation marks with nothing between them, i.e., Q$="". Setting a string variable to the null string can be used to free up the string space previously used by a non-null value of that string variable.

It is often desirable to be able to access parts of a string and manipulate those parts. The string A$ was previously defined. If we wished to see only some of its contents we might use:

```
PRINT LEFT$(A$,3)
BASIC
READY.
```

"LEFT$" is a string function which returns a string composed of the leftmost characters of its string argument. Another example might read:

```
FOR N=1 TO LEN(A$):PRINT LEFT$(A$,N):NEXT N
M
MT
MTU
MTU
MTU B
MTU BA
MTU BAS
MTU BASI
MTU BASIC
READY.
```

Since A$ has 9 characters, the loop will be executed with N=1,2,3,...,8,9. The first time through only the first character will be printed. The second time, the first two characters will be printed, etc. You should also note that the FOR..NEXT loop has been entered on one line. A colon (:) may be used to separate multiple statements on a line, as was done here.

There is a similar string function called "RIGHT$" which returns the right N characters from a string expression. Try substituting RIGHT$ for LEFT$ in the previous example and note the result.

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

```
FOR N=1 TO LEN(A$):PRINT MID$(A$,N):NEXT N

MTU BASIC
TU BASIC
U BASIC
 BASIC
BASIC
ASIC
SIC
IC
C
READY.
```

MID$ returns a string starting at the Nth position of A$ to the end (last character) of A$. The first position of the string is position 1 and the last possible position of a string is position 255. Very often it is desirable to extract only the Nth character from a string. This can be done by calling MID$ with three arguments. The third argument specifies the number of characters to return. For example:

```
FOR N=1 TO LEN(A$):PRINT MID$(A$,N,1),MID$(A$,N,2):NEXT N

M   MT
T   TU
U   U
     B
B   BA
A   AS
S   SI
I   IC
C   C
READY.
```

Strings may also be concatenated (joined together) through the use of the "+"

operator. Try the following:

```
B$="HELLO"+" "+A$
READY.
PRINT B$
HELLO MTU BASIC
READY.
```

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$,5)+"-"+MID$(B$,7,3)+"-"+RIGHT$(B$,5)
READY.
PRINT C$
HELLO-MTU-BASIC
READY.
```

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

```
S$="567.8"
READY.
PRINT VAL(S$)
567.8
READY.
S$=STR$(3.1415)
READY.
PRINT S$,LEFT$(S$,5)
3.1415  3.14
READY.
```

"STR$" can be used to perform formatted I/O on numbers. A number may be converted to a string and then LEFT$, RIGHT$, and MID$ can be used to reformat the number as desired. STR$ can also be used to find out how many print columns a number will require. For example:

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

VAL is useful in many different applications. If a program accepted a question from a user such as:

"WHAT IS VOLUME OF A CYLINDER OF RADIUS 5.36 FEET, OF HEIGHT 5.1 FEET?"

VAL could be used 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 quite similar to the one given in 1.10 that sorted a numeric list. Strings may also be elements of arrays. These are dimensioned exactly like numeric arrays, except that the variable used as the array name must be a string variable (having last character "$"). For instance, DIM A$(10,10) creates a string array of 121 elements, eleven rows by eleven columns (rows 0 to 10 and columns 0 to 10). Each string array element is a complete string, which can be up to 255 characters in length.

```
100 DIM A$(15): REM ALLOCATE SPACE FOR STRING MATRIX
110 FOR I=1 TO 15;READ A$(I):NEXT I
115 REM READ IN STRINGS
120 F=0:I=1
125 REM SET EXCHANGE FLAG TO ZERO AND SUBSCRIPT TO ONE
130 IF A$(I)<=A$(I+1) THEN 180
135 REM DON'T EXCHANGE ORDERED ELEMENTS
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 EXCHANGE OF ELEMENTS
180 I-I+1:IF I<15 GOTO 130
185 REM ONCE PASS IS MADE THRU ALL ELEMENTS,
187 REM CHECK TO SEE IF WE EXCHANGED ANY
188 REM IF NOT, SORT COMPLETED.
190 IF F THEN 120:PRINT A$(I):NEXT I:REM PRINT LIST
200 FOR I=1 TO 15:PRINT A$(I):NEXT I:REM PRINT LIST
210 REM STRING DATA FOLLOWS
220 DATA APPLE,DOG,CAT,BASIC,MTU,COMPUTER
230 DATA MONDAY,"***ANSWER***","FOO"
240 DATA RANDOM,BISCUIT,BELLEVUE,SEATTLE,BYTE,BIT,SUGAR
```

2.17 OUTPUTTING TO A PRINTER

Outputting to a printer requires two steps. You must assign a channel to the Printer device (assuming you have configured one into CODOS). Then you simply output to that channel and the characters should appear on the printer. This can be done very easily while in BASIC using the CIL Library.

If you want to output to the printer without using the CIL, you should use the CODOS ASSIGN command to assign an unused channel to the printer. This must be done prior to entering BASIC. Then after starting BASIC, you can switch your output to the printer by executing an OUTCHAN command specifying the channel to which you assigned the printer. At this point, all ouput which normally went to the display screen will go to the printer instead. This will continue until another OUTCHAN command is given, or a BASIC error occurs. Refer to the OUTCHAN command for further details.

3.1 SPACE OPTIMIZATION

The following hints will help you save memory space.

1. Use multiple statements per line. There is a small amount of overhead (5 bytes) associated with each line in the program. Two of these five bytes contain the line number of the line in binary. This means that no matter how many digits you have in your line number, (min number is 0, max number is 63999), it takes the same number of bytes. Putting as many statements as possible on a line will cut down on the number of bytes used by your program.

2. Delete all unnecessary spaces from your program. The statement

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

   uses three more bytes than

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

   and all spaces between the line number and the first non-blank character are ignored.

3. Delete all REM statements. Each REM statement uses at least one byte plus the number of bytes in the comment text. For instance, the statement

```
130 REM THIS IS A COMMENT
```

   uses 24 bytes of memory. In the statement

```
140 X=X+Y:REM UPDATE SUM
```

   the REM uses 14 bytes of memory including the colon before the REM.

4. Use variables instead of constants. Suppose you use the constant 3.14149 ten times in your program. If you insert a statement 10 P=3.14149 into the program, and use P instead of 3.14149 each time it is needed, you will save 41 bytes. This will also result in improved execution speed.

5. The END statement is entirely optional at the end of a program.

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

7. Use GOSUBs to execute sections of program statements that perform identical actions.

8. Use the zero elements of arrays; for instance A(0), B(0,X). Simple (non-matrix) numeric variables like V use 7 bytes; 2 for the variable name, and 5 for the value. Simple non-matrix string variables also use 7 bytes, 2 for the variable name, 1 for the length, and 2 for a pointer. Two bytes are unused. Array variables use a minimum of 12

12 bytes. Two bytes are used for the variable name, two for the size of the array, two for the number of dimensions, and two for each dimension along with four bytes for each of the array elements.

String variables also use one 1 byte of string space for each character in the string. This is true whether the string variable is a simple string variable like A$, or an element of a string array such as Q1$(5,3). When a new function is defined by a DEF statement, 6 bytes are used to store the definition.

9. Reserved words such as FOR, GOTO or NOT, and the names or intrinsic functions such as COS, INT AND STR$ take up only one byte of program storage. All other characters in programs use one byte of program storage each. When a program is being executed, space is dynamically allocated on the stack as follows:

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

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

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

10. Use integer variables. Integer variables require 2 to 4 fewer bytes for storage (depending on the version of BASIC) than floating point variables.

3.2 TIME OPTIMIZATION

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

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

2. Use variables instead of constants. It takes more time to convert a constant to its floating point representation than it does to fetch the value of a simple or matrix variable. This is especially important within FOR...NEXT loops or other code that is executed repeatedly. THIS IS PROBABLY THE MOST IMPORTANT SPEED HINT BY A FACTOR OF 10!

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

4. Omit the index variable from NEXT statements. NEXT is somewhat faster than its equivalent NEXT I because no check is made to see if the variable specified in the NEXT is the same as the variable in the most recent FOR statememt.

3.3 DERIVED FUNCTIONS

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

```
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)/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
```

3.4 CONVERSION OF BASIC PROGRAMS

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

1. Array Subscripts - Some BASIC's use "[" and "]" to denote array subscripts. MTU BASIC uses "(" and ")".

2. Strings - A number of BASIC's force you to dimension (declare) the length of strings before you use them. You should remove all dimension statements of this type from the program. In some of these BASICs, a declaration of the form DIM A$(I,J) declares a string matrix of J elements each of which has a length I. Convert DIM statements of this type to equivalent ones in BASIC: DIM A$(J). MTU BASIC uses LEFT$, Right$ and MID$ to take substrings of strings. Other BASIC's use A$(I) to access the Ith character of the string A$, and A$(I,J) to take substring of A$ from character position I to character position J. Convert as follows:

```
OLD           NEW
A$(I)         MID$(A$,I,1)
A$(I,J)       MID$(A$,I,J-I+1)
```

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

```
OLD           NEW
A$(I)=X$      A$=LEFT$(A$,I-1)+X$+MID$(A$,I+1)
A$(I,J)=X$    A$=LEFT$(A$,I-1)+X$+MID$(A$,J+1)
```

3. Multiple assignments. Some BASICs allow statements of the form:

```
500 LET B=C=0.
```

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

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

4. Some BASICs use a backslash instead of ":" to delimit multiple statements per line. Change the backslash to ":" in the program.

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

4. THE LIST OF STANDARD COMMANDS, STATEMENTS, FUNCTIONS AND OPERATORS

The following tables list the standard command, program statements, operators that are always available when using MTU BASIC. Additional specialized commands and functions (such as for graphics or advanced disk I/O) may be temporarily added on through use of the Library Facility. Please refer to the LIB command description and to the separate Library Program manuals for more information about this unique MTU BASIC feature.

The reserved variables shown are predefined by MTU BASIC. They may be used in an expression to obtain the associated value. However, a BASIC program may not store a new value it the reserved variables.

COMMANDS	PROGRAM STATEMENTS	I/O STATEMENTS	ARITHMETIC FUNCTIONS	STRING FUNCTIONS
BYE *	DEF	DATA	ABS	ASC
CLEAR	DIM	GET	ATN	CHR$
CONT	END	INPUT	COS	LEFT$
EDIT *	FOR	OUTCHAN *	EXP	LEN
ENTER *	GOSUB	PRINT	FRE	MID$
FRELIB *	GOTO	READ	FN	RIGHT$
LIB *	IF...GOTO		INT	STR$
LIST	IF...THEN		LOG	VAL
LOAD	LEGEND *		PEEK	
NEW	LET		POS	
RUN	MCALL		RND	
SAVE	NEXT		SGN	
	ON...GOSUB		SIN	
	ON...GOTO		SQR	
	POKE		TAN	
	REM		USR	
	RESTORE			
	RETURN			
	STOP			
	TONE *			
	WAIT			

ARITHMETIC OPERATORS	RELATIONAL OPERATORS	STRING OPERATORS	LOGICAL OPERATORS
+ (add.)	= (equals)	+ (concat.)	AND
- (sub.)	< (less than)		NOT
- (neg.)	> (greater than)		OR
* (mult.)	<= (less or equals)		
/ (div.)	>= (greater or equals)		
^ (exp.)	<> (not equal)		

RESERVED VARIABLES
ST
KE(Y) *

* - MTU added command, program statement, or reserved variable.

5. MTU BASIC COMMANDS

This section describes all of the standard commands found in MTU BASIC. They are called commands instead of program statements because they are usually executed after BASIC has typed "READY.". However, there is no restriction on using these commands within a BASIC program. It should be noted that a few of these commands will terminate program execution when they are finished.

5.1 THE BYE COMMAND

PURPOSE: To exit BASIC and return to the CODOS Monitor.

SYNTAX: BYE

ARGUMENTS:

none

DISCUSSION:

Upon start up of MTU BASIC certain CODOS parameters are modified to better suit the running of BASIC. In order to properly return to the CODOS Monitor, these parameters must be changed back to their original values. This function is performed by the BYE command.

EXAMPLES:

BYE

will return to CODOS, restoring the modified CODOS parameters.

NOTES:

1. When a BYE command is executed, three CODOS parameters are restored to their normal values. First the CODOS error handling routine is set to return to the CODOS Monitor after printing its message. Second, CNTL-C handling is set to return to the CODOS Monitor if a CNTL-C is entered on the keyboard. And finally, the default file extension is set back to "C".

2. The BYE command is the only proper way to return to the CODOS Monitor from BASIC. Pressing the INT key will return you to CODOS, but will leave the parameters mentioned in NOTE 1 unrestored.

3. Once you have returned to the CODOS Monitor with the BYE command or INT key, you may reenter BASIC by executing GO 0. This assumes you haven't disturbed the BASIC interpreter machine code or BASIC's page zero.

5.2 THE CLEAR COMMAND

PURPOSE: To clear all variables; reset FOR and GOSUB state, and RESTORE data.

SYNTAX: CLEAR

ARGUMENTS:

none

DISCUSSION:

The CLEAR command essentially resets BASIC without disturbing the BASIC program, or halting its execution. All variables will be cleared. Also, all FOR..NEXT loops and GOSUBs are cleared. And finally, the DATA pointer for READ statements is RESTOREd.

EXAMPLES:

```
A=1: CLEAR: PRINT A
```

will print 0, since the contents of variable A were cleared.

NOTES:

1. Inserting, deleting or modifying a line of a BASIC program will cause an automatic CLEAR to be performed.

5.3 THE CONT COMMAND

PURPOSE: To continue program execution after it has be stopped by a CTL-C or a STOP statement.

SYNTAX: CONT

ARGUMENTS:

none

DISCUSSION:

The CONT command is used to continue execution of a program after it has been halted by a CTL-C or STOP statement. It cannot be used to continue after an error, after modifying your program, or before the program has be run.

When the executing program is halted, a "BREAK IN LINE " message will be printed with the line number of program line where the halt occurred. At this point you may freely examine the contents of any variables. After this you may continue the execution of the program if you wish.

EXAMPLES:

```
CONT
```

will continue execution of the program.

5.4 THE EDIT COMMAND

PURPOSE: To assist in modifying existing program lines.

SYNTAX: EDIT <line number>

ARGUMENTS:

<line number> = the line number of the line you wish to edit.

DISCUSSION:

The EDIT command makes it easier to modify existing program lines. Executing an EDIT command will cause the specified line to be automatically entered for you, the same as if you retyped the line from the keyboard. The line will appear on the display with the text cursor at the end of the line. At this point you may make any changes you like, even to the line number. All line editing features in the console device may be used, such as INSERT, DELETE, CTL-B, etc. If you wish, you may enter a whole new line or command. The EDIT command simply saves you the trouble of retyping the specified line yourself.

EXAMPLES:

```
EDIT 1000
```

will present line 1000 on the display ready for editing.

NOTES:

1. If the specified line does not exist, an UNDEF'D STATEMENT ERROR will be given.

5.5 THE ENTER COMMAND

PURPOSE: To enter an ASCII BASIC program from a file or device.

SYNTAX: ENTER <file name>

ARGUMENTS:

<file name> = a string expression which evaluates to the desired file name.

DISCUSSION:

The ENTER command is provided to help make BASIC programs more transportable. It is easier to transport programs in ASCII since an editor can be used to convert the program before ENTERing it into MTU BASIC.

EXAMPLES:

```
ENTER "OTHERPROG.T:1"
```

will enter the program found in OTHERPROG.T on drive 1.

```
ENTER NM$
```

will enter the program in the file specified by NM$. If a drive is not specified in NM$, then the default drive is used. If an extension is not specified, it defaults to ".B".

NOTES:

1. The ENTER command doesn't actually enter the program itself. It simply assigns channel 0 to the specified file, then switches BASIC's input channel from 1 to 0. BASIC now reads characters from channel 0, unaware that the characters are coming from a file and not the console device. BASIC will continue to read from channel 0 until EOF is encountered. At this time the input channel is switched back to channel 1.

2. If an ENTER command is executed in a running program, the INPUT and GET commands will read data from the assigned file. If EOF is not reached when the program execution stops, BASIC will continue ENTERing from the file until EOF is reached. This can be dangerous since numeric data will be interpreted as new BASIC program lines and possibly overwrite existing program lines in memory.

3. If the program being ENTERed contains Library commands, that Library must be LIBed in before executing the ENTER command. This is the only way BASIC would know to tokenize the Library commands.

4. CTL-C may be used to abort the entering of a program. All lines previously entered by the ENTER command will remain in your program. Also, aborting the ENTER command will leave channel 0 assigned to the file than was being entered. At this point you should manually free channel 0. This could be done by executing the following three commands:

```
BYE
FREE 0
GO 0
```

5.6 THE FRELIB COMMAND

PURPOSE: To "unlink" all the libraries currently "linked" to BASIC.

SYNTAX: FRELIB

ARGUMENTS:

none

DISCUSSION:

This command is used to return the active command set to just the standard commands. This may be in preparation for LIBing a new set of libraries.

EXAMPLES:

FRELIB

will unlink all libraries, and return the active command set to just the standard commands and statements.

NOTES:

1. If executed as a direct command, the top of memory will be set back to the default memory top, and a CLEAR command is executed. This returns the amount of free memory to the original amount established when you started up BASIC.

2. If executed from a running program the top of memory and variables are left undisturbed. You may load other Libraries without disturbing the variables as long as none of the Libraries load below the current top of memory. This allows a program to "overlay" Libraries to make more room for the program and data. When the program stops, you will have to execute another FRELIB in order to set the top of memory back to the default.

5.7 THE LIB COMMAND

PURPOSE: To load external Library programs and link them to BASIC.

SYNTAX: LIB <file name> [, <file name>] ...

ARGUMENTS:

⟨file name⟩ = a string expression which evaluates to the desired Library Program name.

DISCUSSION:

The LIB command allows the standard set of BASIC commands to be expanded by linking external Library Programs into BASIC. Up to eight Library Programs may be linked at one time. If the LIB command is executed with no parameters, the currently linked libraries are listed out in the order in which their keyword lists will be scanned. Sometimes this order is important as will be explained later.

In addition to linking the new commands into BASIC, the LIB command performs some memory management when actually loading the Library program into memory. When linking more than one Library program, the programs might easily compete for the same address space. The LIB command has a simple solution to this problem. A Library file is allowed to contain multiple copies of the program, each assembled to run at a different location. When loading a Library program, the LIB command will search the Library file in sequence until a program is found which fits in the allocatable space and doesn't conflict with another Library. Here, the allocatable space consists of the 24K of memory just below the default top of memory established during the start up proceedure.

The LIB command will link each Library program as its name is encountered in the command line. Once linked, it will continue with the next name if one is present. If a Library is found to be already linked, it will not be relinked. Linking of that Library is skipped, and the version of that Library already linked is left as is. If the LIB command can't find a program in the file which it can load, a CAN'T LOAD LIBRARY ERROR is given.

The order in which the keyword lists will be scanned is the reverse of the order in which they were linked. In other words, the keyword list of the last Library linked is the first one scanned. As stated in the first paragraph, this order can be important. As each BASIC line is entered, the line is scanned to see if any character sequences match any of the currently available keywords. If a match is found, those characters are replaced by a token so the command may be executed. The order that the library commands are scanned can affect how statements are tokenized. Here is an actual situation where the order is important.

The IGL Library contains a command called SMOVE, and the VGL Library contains a command called MOVE. If VGL is linked after IGL, then the VGL's keywords will be scanned before IGL's. This means that in a line containing an SMOVE command, the characters "MOVE" will be matched and tokenized as a VGL command. The leftover "S" will cause a SYNTAX ERROR when the command is executed. To be able to execute an SMOVE command in this situation, the IGL must be linked after the VGL. This will allow the "SMOVE" characters to be matched and tokenized before the "MOVE" part can be tokenized incorrectly as a MOVE command.

EXAMPLES:

```
LIB "CIL"
```

will link the Library program found in the CIL.Z file on the default drive. The commands in this Library will be scanned prior to any libraries already linked.

```
LIB "KGL:1","MYLIB.X"
```

will link the program in the KGL.Z file on drive 1, followed by the program in MYLIB.X on the default drive.

If the above LIB commands account for the only libraries currently linked, then the command:

```
LIB
```

will print the following list.

```
MYLIB.X
KGL.Z
CIL.Z
```

NOTES:

1. If a CAN'T LOAD LIBRARY ERROR occurs, it means that the LIB command was unable to load one of the Library programs specified due to memory conflicts. If there was more than one argument specified in the command you need to find out which one couldn't be loaded. This is determined by executing a LIB command with no arguments. The first Library listed is the last one to be successfully linked. The argument following this one in the original command line is the Library that couldn't be loaded.

This error will most likely occur if one of the Libraries in the list is already linked, and thus is not reloaded. The usual solution is to execute a FRELIB command, then trying to execute the LIB command again. The following sequence of commands will illustrate this error and its solution.

```
LIB "IGL"
LIB "CIL","IGL"          - which gives a CAN'T LOAD LIBRARY ERROR
FRELIB
LIB "CIL","IGL"          - which now succeeds in loading both libraries.
```

If this doesn't succeed, you will have to execute another FRELIB command and try linking the Libraries in a different order. You should consult the manual for the particular Libraries to obtain further information about linking them.

2. If a token is executed for a Library program which isn't linked, BASIC will first read its name from the SYSLIBNAM.Z file. An error message is then printed similar to the following:

```
CIL.Z LIBRARY
?NOT LOADED ERROR
```

5.8 THE LIST COMMAND

PURPOSE: To output an ASCII listing of the program in memory.

SYNTAX: LIST <first line> [- [<last line>]]

or LIST <file name> [, <first line> [- [<last line>]]]

ARGUMENTS:

<first line> = the number of the first line to list. Defaults to the first line of the BASIC program.

<last line> = the number of the last line to list. Defaults to the last line of the BASIC program.

<file name> = a string expression which evaluates to the desired file name. In the LIST command this may evaluate to a device name as well.

DISCUSSION:

The LIST command is used to view portions of the BASIC program on the display, or to save an ASCII copy of the program on disk. The ASCII copy can be edited and later loaded back into memory using the ENTER command.

If a file name is not specified in the command, the listing is output to BASIC's output channel. If not set otherwise, this is channel 2.

EXAMPLES:

```
LIST
```

will list all of the BASIC program in memory to BASIC's output channel, which is usually assigned to the console.

```
LIST 1000-
```

will list from line 1000 through the end of the program to BASIC's output channel.

```
LIST "P"
```

will list all of the program to the "P" device.

```
LIST "SUBROUTINE.T:1",200-300
```

will list from line 200 through line 300 to the file SUBROUTINE.T on drive 1.

NOTES:

1. If a file name is specified, it is expected to be a new file. A FILE EXISTS ERROR is given if it is not a new file.

5.9 THE LOAD COMMAND

PURPOSE: To load a BASIC program in memory image (tokenized) format from disk.

SYNTAX: LOAD ⟨file name⟩

ARGUMENTS:

⟨file name⟩ = a string expression which evaluates to the desired file name.

DISCUSSION:

The LOAD command is used to load a memory image copy of a BASIC program that was saved using the SAVE command. Because it is a memory image copy, the program will load very fast. The LOAD command can also be used to chain execution to another program, see note 1.

EXAMPLES:

```
LOAD "THISPROG"
```

will load the program THISPROG.B from the default drive.

```
LOAD NM$
```

will load the program specified by NM$. If a drive is not specified in NM$, then the default drive is used. If an extension is not specified, it defaults to ".B".

NOTES:

1. If a LOAD command is executed from a running program, the specified program will be loaded into memory. If this program is smaller than or equal in size to the previous one, all variables are left intact and execution continues with the first statement of the program just loaded. If the specified program is larger than the previous one, then the variables are cleared and execution halts. This is also what occurs when the LOAD command is executed as a direct command.

5.10 THE NEW COMMAND

PURPOSE: Delete the current program and all variables.

SYNTAX: NEW

ARGUMENTS:

none

DISCUSSION:

The NEW command is used to delete the current program and clear all variables.

EXAMPLES:

NEW

will delete the current program and variables.

NOTES:

1. The NEW command may be executed from a running program, but it will naturally halt execution when the program clears itself.

5.11 THE RUN COMMAND

PURPOSE: To load and run a BASIC program in memory image (tokenized) from disk.

SYNTAX: RUN [<line number>]

or RUN <file name> [,<line number>]

ARGUMENTS:

file name = a string expression which evaluates to the desired file name.

line number = line number at which execution is to begin. This defaults to the first line of the program if not specified.

DISCUSSION:

The RUN command is used to load and begin execution of a BASIC program. Prior to running the program all variables are cleared. Execution then begins with the specified line, or with the first line of the program.

EXAMPLES:

```
RUN "THISPROG"
```

will load the program THISPROG.B from the default drive and begin execution with the first line.

```
RUN "INVENTORY:1",1000
```

will load the program INVENTORY.B from drive 1 and begin execution with line 1000.

5.12 THE SAVE COMMAND

PURPOSE: To save a BASIC program to disk in memory image (tokenized) format.

SYNTAX: SAVE <file name>

ARGUMENTS:

<file name> = a string expression which evaluates to the desired file name.

DISCUSSION:

The SAVE command saves a memory image copy of the BASIC program in memory to disk. Because it is a memory image copy, the program is saved at high speed.

EXAMPLES:

```
SAVE "LEDGER:1"
```

will save the program currently in memory as LEDGER.B on the disk in drive 1.

```
SAVE NM$
```

will save the program as the file specified by NM$. If a drive is not specified in NM$, then the default drive is used. If an extension is not specified, it defaults to ".B".

NOTES:

1. The SAVE command expects the file specified to be a new file. If it is not, a FILE EXISTS ERROR is given.

6. MTU BASIC STATEMENTS

This section describes all of the standard statements found in MTU BASIC. They are called statements, instead of commands, for no other reason than that they are typically used as statements in a program. There are no restrictions against using them in a direct command. Note that manuals for the various Libraries typically don't distinguish between commands and statements. These manuals call everything commands, except for any functions they provide.

Not included in this section are statements which perform some sort of I/O.

6.1 THE DEF STATEMENT

PURPOSE: To define a user defined function.

SYNTAX: DEF FN <function name>(<dummy variable>) = <expression>

ARGUMENTS:

<function name> = a one or two character variable name to represent the name of the function.

<dummy variable> = a one or two character variable name, to be used in the expression to indicate where the passed parameter should be substituted.

<expression> = a numeric expression defining the value to be returned by the function. The expersion may involve the dummy variable along with other variables.

DISCUSSION:

The DEF is used to define your own functions. The user defined function is restricted to one line. The function may be defined by any expresion, but may have only one argument passed to it. Where the dummy variable appears in the expression the value of the passed argument is substituted. This is the only use of the dummy variable, and will in no way affect a "normal" variable with the same name.

Once defined, "FN" plus the user function name may be used in an expression to call the function.

EXAMPLES:

```
10 DEF FN A(V) = V/B+C
```

defines a function which returns the value of the argument, divided by B+C. The function may be called using the term FNA(arg) in an expression.

```
10 DEF FNAC(X) = -ATN(X/SQR(-X*X+1))+1.5708
```

defines the AC function as an inverse cosine function. The function may be called using the term FNAC(arg) in an expression.

NOTES:

1. If another statement follows the DEF statement, it will be executed when the DEF statement has finished defining the user defined function. This following statement will not be executed when the function is called.

2. A user defined function may be redefined during program execution by executing another DEF statement for that function.

6.2 THE DIM STATEMENT

PURPOSE: To create and allocate space for arrays.

SYNTAX: DIM <name>(<exp> [, exp>] ...) [, <name>(<exp> [, <exp>] ...)] ...

ARGUMENTS:

<name> = a variable name to be used for the array name.

<exp> = an expression for the maximum value in a dimension.

DISCUSSION:

The DIM statement allows you to explicitly create arrays of the desired size. When created, all numeric or string array elements are set to zero or a null string, respectively. An array may have up to 255 dimensions. The subscript for each dimension may range from 0 up the the value of the expression given in the DIM statement for that dimension. Since all subscribts start at zero, an array with a dimension of n will have n+1 elements.

EXAMPLES:

```
10 DIM A(3),B$(2,4,4)
```

creates a single dimension floating point array named A, and a string array named B$ having three dimensions.

```
10 DIM X%(2*I)
```

creates an integer array named X% with a single dimension of 2 times the current value of I.

NOTES:

1. If an array has not been explicitly dimensioned when first encountered during program execution, it will be automatically dimensioned with a single dimension of 10.

2. You may dimension an array only once during the execution of a program. Dimensioning an array a second time results in a REDIM'D ARRAY ERROR.

6.3 THE END STATEMENT

PURPOSE: To terminate program execution without the BREAK message.

SYNTAX: END

ARGUMENTS:

none

DISCUSSION:

The END statement is used to terminate program execution and return you to "READY." mode without printing the BREAK message. The END statement may appear anywhere in a program.

EXAMPLES:

```
1000 END
```

will terminate program execution and return to "READY." mode.

NOTES:

1. An END statement is not required as the last statement of a program.

2. Executing a CONT command after and END statement will resume execution on the statement following the END statement.

6.4 THE FOR STATEMENT

PURPOSE: To define the beginning of a FOR..NEXT loop.

SYNTAX: FOR <loop variable> = <exp> TO <exp> STEP <exp>

ARGUMENTS:

<loop variable> = a non-array variable name to be used as the loop variable.

<exp> = a numeric expression.

DISCUSSION:

The FOR statement is used in conjunction with the NEXT statement to create a FOR..NEXT loop. The statements falling between the FOR and a corresponding NEXT statements will be executed a particular number of times as controlled by the FOR..NEXT loop. The expression following the "=" is the initial value assigned to the loop variable. The expression following the "TO" will be used as the termination value for the loop variable. The expression following the "STEP" will be used as the step value for the FOR..NEXT loop. If the step value is not specified, it default to 1. The values of the expressions are computed only once, before the body of the FOR..NEXT loop is executed.

After a FOR..NEXT loop has been set up by the FOR statement, execution continues with the statement following the FOR statement. This statement is actually the beginning of the loop. Program execution will continue until a corresponding NEXT statement is encountered. At this point the step value is added to the loop variable. The resulting value is then compared with termination value. If the step value is positive and the loop variable is less than or equal to the termination value, execution jumps back to the beginning of the loop. If the step value is negative, and the loop variable is greater than or equal to the termination value, execution jumps back to the beginning of the loop. If the loop variable fails the appropriate condition above, execution continues with the statement following the NEXT statement. When this statement is executed, the loop variable will contain the value which caused the loop to terminate. If needed, there may be more that one corresponding NEXT statement within the body of a loop.

A FOR..NEXT loop should always be exited through a NEXT statement, even though BASIC permits you to jump out of body of the loop without error. However, doing this will leave data associated with the loop on the 6502 stack. If done too many times, your program will come to a screeching halt with a OUT OF MEMORY error when the stack runs out of space. Also, nesting may be done as long as stack space permitts. Eighteen bytes of the stack is used for each FOR..NEXT loop which is active.

EXAMPLES:

```
10 FOR I= 1 TO 6.5 STEP .5
```

will set up a FOR..NEXT loop which will execute with values of I equal to 1, 1.5, 2, 2.5, 3, 3.5, 4, 4.5, 5, 5.5, 6.0, and 6.5.

```
10 FOR I= 8 TO 0 STEP -1
```

will set up a FOR..NEXT loop which will execute with values of I equal to 8, 7, 6, 5, 4, 3, 2, 1, and 0.

NOTES:

1. If a step value of zero is specified, the NEXT statement will jump back only if the loop variable is not equal to the termination value. If they are equal, the loop will be exited.

2. Where possible, you should use whole numbers for the step value. Using small fractions as the step size could lead to round off errors if the loop is executed many times. The result is that the number of times the loop is executed may differ slightly from what is exepected.

6.5 THE GOSUB STATEMENT

PURPOSE: To execute a BASIC subroutine.

SYNTAX: GOSUB <line number>

ARGUMENTS:

<line number> = the line number of the first line in the subroutine. The number must be written out; variables or expressions are not allowed.

DISCUSSION:

The GOSUB statement allows you to execute a set of statements as a subroutine. The GOSUB statement will cause execution to jump to the specified line number. Execution continues at this point until a RETURN statement is encountered. At this point execution jumps back to the statement following the calling GOSUB.

For each GOSUB executed there should be a matching RETURN executed, eventually. Failing to RETURN from a GOSUB will leave some data on the 6502 stack. If done too many times you program will come to a screeching halt with an OUT OF MEMORY error when the stack runs out of space. Also, you may nest GOSUBs as long as stack space permits. Five bytes of the stack is used by each GOSUB which is active.

EXAMPLES:

```
10 GOSUB 1000
```

branches to the subroutine which starts at line 1000.

6.6 THE GOTO STATEMENT

PURPOSE: To jump execution to a specified line number.

SYNTAX: GOTO <line number>

ARGUMENTS:

<line number> = the line number of the program line to jump to. The number must be written out; variables or expressions are not allowed.

DISCUSSION:

The GOTO command causes program execution to jump to the specified line number.

EXAMPLES:

```
10 GOTO 1000
```

will cause program execution to jump to line 1000 when this statement is executed.

6.7 THE IF..GOTO STATEMENT

PURPOSE: To test a relation and jump execution to a specified line number if true.

SYNTAX: IF <relation> GOTO <line number>

ARGUMENTS:

<relation> = a relational expression to be evaluated.

<line number> = the line number of the program line to jump to. The number must be written out; variables or expressions are not allowed.

DISCUSSION:

The IF..GOTO statement will evaluate the specified relational expression. If the expression is true, then the GOTO is executed. This will jump execution to the specified line number. If the relational expression is not true, then program execution continues on the next line.

EXAMPLES:

```
10 IF X = Y-1 GOTO 1000
```

will test if the value in X is equal to the value in Y minus 1. If it is, then execution will jump to line 1000. If not, execution continues on the next line.

6.8 THE IF..THEN STATEMENT

PURPOSE: To test a relation and execute a set of statements if true.

SYNTAX: IF <relation> THEN <statements>

ARGUMENTS:

<relation> = a relational expression to be evaluated.

<statements> = one or more BASIC statements, to be executed if relation is true.

DISCUSSION:

The IF..THEN statement will first evaluate the relational exepression. If the expression is true, program execution continues with the statement(s) following the "THEN". If the relational expression is false, execution continues with the first statement on the next line.

EXAMPLES:

```
200 IF X <0 OR X > 20 THEN X = 0: PRINT "X IS OUT OF RANGE":GOTO 100
```

will test if X is less than 0 or greater than 20. If so, then X is set to 0,"X IS OUT OF RANGE" is printed, and execution jumps back to line 100. If X is within the range 0 to 20, then execution continues on the next line.

1. Actually, the IF..THEN statement may test any numeric expression. The THEN clause of an IF statement is executed whenever the expression after the IF does not evaluate to 0. For example, IF X THEN... is equivalent to IF X <>0 THEN.

2. The IF..THEN statement will also accept a line number following the THEN. This form of IF..THEN statement will execute the same as the IF..GOTO statement. For example IF X=1 THEN 1000 will execute the same as IF X=1 THEN GOTO 1000.

6.9 THE LEGEND STATEMENT

PURPOSE: To enter legends into the legend boxes shown at the bottom of the CRT display screen.

SYNTAX: LEGEND <exp> , <legend> [, <legend>] ...

ARGUMENTS:

<exp> = a numeric expression which evaluates to a number from 1 to 8. This number identifies the legend box in which to begin entering the legends.

<legend>= a string expression containing the desired legend. Only the first 8 characters will be placed in the legend box.

DISCUSSION:

The LEGEND statement is used to display legend strings in the legend boxes seen on the screen. The legend boxes are numbered from 1 to 8 starting from the left, with each box having room for eight characters. Up to eight characters of the legend string will be placed in the appropriate box left justified. Any left over space in the box will contain blanks.

The numeric expression specifies the box into which the first legend string is to be placed. Any additional legend strings in the statement will be placed in legend boxes in ascending order after the specified box. If a legend string is a null string, the legend box will be cleared.

EXAMPLES:

```
10 LEGEND 3,"QUIT"
```

will place "QUIT" in the leftmost 4 character of the fourth legend box. The remaining four character positions will be blank.

```
10 LEGEND 1," Enter"," Delete"," Modify"
```

will place " Enter" in legend box 1, " Delete" in legend box 2, and " Modify" in legend box 3. The leading space in each string helps center the legend in the box.

```
10 LEGEND 4,"  HELP","",""
```

will place " HELP" in legend box 4, and clears legend boxes 5 and 6.

NOTES:

1. In the LEGEND command, legend strings will be accepted and drawn until a legend is placed in the eighth legend box. If additional legend strings are specified, a SYNTAX ERROR results.

6.10 THE LET STATEMENT

PURPOSE: To assign a value to a variable.

SYNTAX: LET <variable> = <expression>

ARGUMENTS:

<variable> = the variable to be assigned the value of the expression.

<expression> = an expression of the same type as the variable.

DISCUSSION:

The LET statement is used to assign a value to a variable. The variable may be of any type, including arrays. The only requirement is that the type, numeric or string, of the variable match that of the expression.

Actually, if a BASIC statement does not begin with a keyword, then BASIC assumes that the statement is a LET statement and the characters at the beginning of the statement represent a variable name. This assumption allows you to omit the LET. It also means that if you mistype a command or statement, BASIC will probably try to interpret it as a LET statement.

EXAMPLES:

```
10 LET A = 2 * X
```

will assign variable A the value of 2 times the value of X.

```
10 B$ = "THIS IS IT"
```

assigns the string "THIS IS IT" to the variable B$.

6.11 THE MCALL STATEMENT

PURPOSE: To call (i.e. execute) a machine language routine.

SYNTAX: MCALL <expression>

ARGUMENTS:

<expression> = a numeric expression which evaluates to the decimal address of the machine language subroutine.

DISCUSSION:

The MCALL statement is used to execute a machine language subroutine. The statement simply causes the subroutine to be executed. The MCALL statement itself does not pass or return any parameters.

EXAMPLES:

```
10 MCALL 40960
```

will execute the machine language subroutine at 40960 ($A000 hex).

6.12 THE NEXT STATEMENT

PURPOSE: To declare the end of a FOR..NEXT loop.

SYNTAX: NEXT [<variable name>]

ARGUMENTS:

<variable name> = a non-array variable name used as the loop variable in a FOR..NEXT loop.

DISCUSSION:

The NEXT statement is used in conjunction with the FOR statement to declare a FOR..NEXT loop. When the NEXT statement is encountered, the variable name (if specified) is compared with the variable name of the loop counter in the most recently executed FOR statement. If they match, then NEXT statement will execute, stepping the loop counter variable. Execution of the program will either loop back or continue with the next statement depending on the parameters specified in the FOR statement. Refer to the FOR statement for details on FOR..NEXT loop operation. If the variable names do not match, then a NEXT WITHOUT FOR ERROR is given.

If a variable name is not specified, then no checking is performed. The NEXT statement is assumed to be associated with the most recent FOR statement.

EXAMPLES:

```
100 NEXT I
```

will check if the loop counter variable in the most recent FOR statement was the variable I. If so, the step increment is added to I and execution will either loop back or continue with the next statement.

6.13 THE ON..GOSUB STATEMENT

PURPOSE: To perform a multiway subroutine call based on a specified variable.

SYNTAX: ON <variable> GOSUB <line number> [,<line number>] ...

ARGUMENTS:

<variable> = a numeric variable to be used as the selection variable.

<line number> = the line number of a subroutine. The line number must be written out, variables and expressions are not allowed.

DISCUSSION:

The ON..GOSUB statement is used to perform a multiway subroutine call, based on the value in the selection variable. The selection variable is used to select one of the specified line numbers, if any. If one is selected then a subroutine call is performed to that line number. When the subroutine returns, execution continues with the statement following the ON..GOSUB statement.

The selection variable performs its selection as follows.

SELECTION VARIABLE	ACTION
0 <= sv < 1	Execution falls through ON..GOSUB to next statement. No subroutine is executed.
1 <= sv < 2	Subroutine specified by first line number is executed. Execution continues with statement following ON..GOSUB.
2 <= sv < 3	Subroutine specified by second line number is executed. Execution continues with statement following ON..GOSUB.
Etc.	

If the selection variable selects a line number beyond those specified, execution falls through the ON..GOSUB statement and continues with the statement following the ON..GOSUB. If the selection variable is negative, or greater than 255, then an ILLEGAL QUANTITY ERROR is given.

EXAMPLES:

```
10 ON I GOSUB 1000,2000,3000,4000
```

will execute a subroutine at 1000, 2000, 3000, 4000, or no subroutine, depending on the value of the variable I. For example, if I contains 3 then the subroutine at 3000 will be executed. If I contains 6, then no subroutine is executed. If either case, execution after the ON..GOSUB continues with the statement on the next line.

6.14 THE ON..GOTO STATEMENT

PURPOSE: To perform a multiway jump based on a specified variable.

SYNTAX: ON <variable> GOTO <line number> [,<line number>] ...

ARGUMENTS:

<variable> = a numeric variable to be used as the selection variable.

<line number> = the line number of a statement to jump to. The line number must be written out, variables and expressions are not allowed.

DISCUSSION:

The ON..GOTO statement is used to perform a multiway jump, based on the value in the selection variable. The selection variable is used to selection one of the specified line numbers, if any. If one is selected then execution jumps to that line number.

The selection variable performs its selection as follows.

SELECTION VARIABLE	ACTION
0 <= SV < 1	Execution falls through ON..GOTO to next statement.
1 <= SV < 2	Execution jumps to the first line number specified.
2 <= SV < 3	Execution jumps to the second line number specified.
Etc.	

If the selection variable selects a line number beyond those specified, execution falls through the ON..GOTO statement and continues with the statement following the ON..GOTO. If the selection variable is negative, or greater than 255, then an ILLEGAL QUANTITY ERROR is given.

EXAMPLES:

```
10 ON I GOTO 1000,2000,3000,4000
```

will jump to line 1000, 2000, 3000, 4000, or no jump is performed, depending on the value of the variable I. For example, if I contains 3 then execution jumps to 3000. If I contains 6, then no jump is performed, and execution falls through to the next line. If I contains 1.2, then execution jumps to line 1000.

6.15 THE POKE STATEMENT

PURPOSE: To store a value at a specified memory location.

SYNTAX: POKE <address>,<byte>

ARGUMENTS:

<address> = a numeric expression which evaluates to the decimal address of the desired memory location.

<byte> = a numeric expression which evaluates to the byte you wish to store. Must be a number from 0 to 255.

DISCUSSION:

The POKE statement is used to store a byte value at a specified memory location. If the address argument is not within the range 0 to 65535, or the byte argument within the range 0 to 255, an ILLEGAL QUANTITY ERROR will be given. You should be careful when using this statement, since POKEing the wrong location can have undesireable results.

EXAMPLES:

```
POKE 2048,128
```

will store the value 128 ($80 hex) at the memory location 2048 ($800 hex). Executing this statement will cause INPUT statements to not halt when answered with just a carriage return (see INPUT statement).

```
POKE I+8,V
```

will store the value in V at the memory location I+8.

NOTES:

1. The values may be POKEd into BANK 0 only. You must provide your own machine language routine to store into another BANK.

6.16 THE REM STATEMENT

PURPOSE: To place comment statements in a BASIC program.

SYNTAX: REM <comment>

ARGUMENTS:

<comment> = the group of characters which make up the comment.

DISCUSSION:

The REM statement is used to place comments within a BASIC program without affecting execution. All characters between the "REM" and the end of the program line are treated as a comment. REM statements are skipped over when encountered during execution.

EXAMPLES:

```
1200 PRINT AV: REM AV IS THE AVERAGE

2000 REM THIS SECTION COMPUTES THE INDEX
```

NOTES:

1. REM statements can slow down execution slightly. A small amount of time must be spent skipping over them during execution.

6.17 THE RESTORE STATEMENT

PURPOSE: To restore the DATA pointer used by READ statements to a specified line number.

SYNTAX: RESTORE <line number>

ARGUMENTS:

<line number> = the line number to which the DATA pointer is restored. The line number must be written out, variables and expressions are not allowed.

DISCUSSION:

The RESTORE statement is used to restore the DATA pointer used by READ statements to a specified line number. This line number will then be the point from which READ statements will search for data contained in DATA statements. The RESTORE statement is typically used to reread certain data.

EXAMPLES:

```
RESTORE 1500
```

will restore the DATA pointer to line 500. The next READ statement will take its data from the first DATA statement in, or following this line.

6.18 THE RETURN STATEMENT

PURPOSE: To terminate execution of a subroutine and return to the statement following the most recently executed GOSUB.

SYNTAX: RETURN

ARGUMENTS:

none

DISCUSSION:

The RETURN statement is used to terminate execution of a subroutine. Execution will return to the statement following the most recently executed GOSUB statement.

EXAMPLES:

```
2100 RETURN
```

will return execution to the statement following the most recent GOSUB.

6.19 THE STOP STATEMENT

PURPOSE: To terminate program execution with a BREAK IN LINE number message.

SYNTAX: STOP

ARGUMENTS:

none

DISCUSSION:

The STOP statement is used to terminate execution of a BASIC program with a BREAK IN LINE number message. A CONT command may be used to continue execution with the statement following the STOP.

EXAMPLES:

```
1999 STOP
```

will cause execution to stop, and a BREAK IN LINE 1999 message will be printed.

6.20 THE TONE STATEMENT

PURPOSE: To output a tone via the CB2 signal on the Parallel Output Port.

SYNTAX: TONE [⟨pitch⟩ [, ⟨volume⟩ [, ⟨waveform⟩]]]

ARGUMENTS:

⟨pitch⟩ = an expression which evaluates to a number from 1 to 255. This number is used to set the pitch of the tone.

⟨volume⟩ = an expression which evaluates to a number from 0 to 255. This number is used to set the volume of the generated tone.

⟨waveform⟩ = an expression which evaluates to a number from 0 to 255. This number is used to set the shift register involved with generating CB2 sound.

DISCUSSION:

The TONE command uses the serial output capabilities of the 6522 to generate a tone. This can be used to generate various sounds or play musical notes. Executing a TONE command with parameters will cause the appropriate tone to be generated. Executing a TONE command with no parameters will cause any tone being generated to cease.

EXAMPLES:

```
TONE 237,128,51
```

will set the pitch to 237, the volume to 128, and the waveform to 51. This results in a square wave of the frequency 523 hertz (high C) at a moderate volume level.

```
TONE N
```

will set the pitch to the value in N. The volume and waveform are not changed.

NOTES:

1. The pitch argument relates to frequency according to the following formula: FREQ = 62500/(PITCH+2). This frequency is the number of complete waveforms that are generated per second.
2. The volume parameter controls the amplitude of the tone with 0 representing the minimum loudness, and 255 representing the maximum.
3. Any parameters not specified in the TONE command are left unchanged.
4. The TONE will cease, and the volume set to 128 when program execution terminates.
5. The following pitches and waveforms will generate square waves of the indicated frequeny and pitch:

PITCH	WVFRM	FREQ	NOTE
140	51	880	A
157	51	784	G
177	51	698	F
188	51	659	E
211	51	587	D
237	51	523	HIGH C

PITCH	WVFRM	FREQ	NOTE
251	51	494	B
140	15	440	A
157	15	392	G
177	15	349	F
188	15	330	E
211	15	294	D
237	15	262	MIDDLE C

6.21 THE WAIT STATEMENT

PURPOSE: To cause a BASIC program to wait for the contents of a specified memory location to reach a certain state.

SYNTAX: WAIT <address> , <and mask> , <exor mask>

ARGUMENTS:

<address> = an expression which evaluates to the decimal address of the location to be tested.

<and mask> = an expression which evaluates to the byte value to be used as the AND mask.

<exor mask> = an expression which evaluates to the byte value to be used as the exclusive OR mask.

DISCUSSION:

The WAIT instruction is used to temporarily suspend BASIC program execution until the content of a specified memory location reaches a certain state. When the content reaches the desired state, program execution continues with the statement following the WAIT statement.

Testing of the memory location is performed as follows. The contents of the location is read. This value is exclusive ORed with the exclusive OR mask. This result is ANDed with the AND mask. If this final result is zero, the test is repeated. If the final result is non-zero, program execution resumes.

The WAIT statement is rarely used. However, it can be useful in synchronizing a BASIC program with certain events. It this case, a register in one of the 6522s will be the memory location tested. Some registers in these ICs have bits which indicate if certain events have occured, such as transition on the CB1 signal.

EXAMPLES:

```
10 WAIT 49133,2,0
```

will wait for the second bit of the memory location at 49133 ($BFED hex) to become a 1. The memory location being tested is the Interrup Flag Register of the SYS1 6522. The bit being tested will be set to 1 whenever the BREAK key is pressed. Thus the statement above will suspend execution of the BASIC program until the BREAK key is pressed.

NOTES:

1. To exit a WAIT statement which has become hung (i.e. testing for a condition which won't occur), you must press the INT key. Then enter GO 0 to return to BASIC. You may use the CODOS monitor at this point, but be aware that certain CODOS parameters have been modified by BASIC (see the BYE command). To correct these parameters, enter GO 0 followed by BYE.

7. I/O STATEMENTS

This section describes the standard program statements that are available in MTU BASIC for input and output. Input is performed by the GET, INPUT, and READ statements. The GET and INPUT statements communicate via BASIC's input channel, normally CODOS channel 1. The READ statement reads data defined within the BASIC program itself using DATA statements.

Output is performed by the PRINT statement. The PRINT command communicates via BASIC's output channel, normally CODOS channel 2. The SPC() and TAB() functions can be used within the PRINT statement to help format the output. The OUTCHAN command allows you to change BASIC's output channel, redirecting it to another channel.

7.1 THE DATA STATEMENT

PURPOSE: To declare data to be read by READ statements.

SYNTAX: DATA <data> [, <data>] ...

ARGUMENTS:

<data> = character sequences representing the desired numeric or string data, optionally enclosed within quotes.

DISCUSSION:

The DATA statement is used to declare data to be read by READ statements. The data items will be read from left to right. If you wish to include leading spaces (blanks), colons (:), or commas (,) in a data item, you must enclose the item within quotes. In addition, any data items enclosed within quotes must be read into a string variable.

EXAMPLES:

```
10 DATA 1,3,-156
```

is a statement declaring data items of 1, 3, and -156. These values will probably be read into numeric variables.

```
10 DATA CAT," NUMBER:"
```

is a statement declaring two string data items. Note that the string " NUMBER:" is enclosed within quotes so it could include a leading space and a colon. The colon will be interpreted as the end of the DATA statement if it were not within quotes.

NOTES:

1. It is recommended, though not required, that DATA statements appear as the first statement in a line. It is also recommended that they appear at the end of a program so they won't take up space in the middle, causing execution of the program to be slowed.

7.2 THE GET STATEMENT

PURPOSE: To input a single character or digit.

SYNTAX: GET <variable> [, <variable>] ...

ARGUMENTS:

<variable> = a variable which is to receive the input character.

DISCUSSION:

The GET command reads data one character at a time from BASIC's input channel, which is usually channel 1. If the variable to receive the character is a string variable, then it is assigned to a string containing the one character If it is numeric, then the character is converted to a number and stored in the variable. If this character is not a numeric digit, a SYNTAX ERROR is given.

EXAMPLES:

If "9" and "X" are the next characters to be read from BASIC's input channel, then the following examples will operate as described:

```
GET A$,B$
```

will set A$ to "9" and B$ to "X".

```
GET N,A$
```

will set N to the value 9, and A$ to "X".

NOTES:

1. If BASIC's input channel is assigned to the console device, as will usually be the case, the GET command will wait for a key to be entered. This may differ from other BASICs which do not wait, but simply return 0 or a null string if a key is not down.

2. If one of the non-ASCII keys is entered, the following values will be returned: This must come back in a string variable

KEY	UNSHIFTED VALUE	SHIFTED VALUE	KEY	UNSHIFTED VALUE	SHIFTED VALUE
f1	128	144	X *	138	154
f2	129	145	÷ *	139	155
f3	130	146	- *	140	156
f4	131	147	+ *	141	157
f5	132	148	ENTER	142	158
f6	133	149	↑	160	176
f7	134	150	←	161	177
f8	135	151	→	162	178
PF1	136	152	↓	163	179
PF2	137	153	HOME	164	180
			DELETE	165	181
			INSERT	166	182

* - numeric pad key.

7.3 THE INPUT STATEMENT

PURPOSE: To input ASCII data.

SYNTAX: INPUT ["⟨prompt⟩";] ⟨variable⟩ [, ⟨variable⟩] ...

ARGUMENTS:

⟨prompt⟩ = a string literal (the enclosing quotes are required) to be used as a prompt message.

⟨variable⟩ = a BASIC variable which is to receive the input data.

DISCUSSION:

The INPUT command will read data from BASIC's input channel, which is usually channel 1. Each INPUT command will first output the prompt if one is specified. The command will then output a "?" and input a line of ASCII from BASIC's input channel. For each variable to receive input, characters will be read from this line until a comma or the carriage return is encountered. At this point the data read so far is stored in the variable. If the variable is numeric, the data is converted first. If there is not enough data to input to all the variables in the command, then another "?" is output and a second line of ASCII is input. This continues until all variables have received data. If there are any leftover characters in the line, then "EXTRA IGNORED" is output.

If an error occurs during the input of the data, "REDO FROM START" is output and the INPUT command is executed again.

If a colon is part of a string and the string is enclosed within quotes, then the colon is treated as a normal character. In all other cases, it is treated as if it was a carriage return instead. For example, if the statement:

```
INPUT NM$,D
```

is executed, then the following responses will give the described results.

```
MYFILE.T:1,10
```

will cause NM$ to become "MYFILE.T" and the INPUT statement will ask for more data.

```
"MYFILE.T:1",10
```

will cause NM$ to become "MYFILE.T:1" and D will be set to 10. If a comma is part of a string that is enclosed within quotes, then the comma will also be treated as a normal character, not a terminator.

EXAMPLES:

```
INPUT "INPUT NUMBER OF ITEMS ";N
```

will print "INPUT NUMBER OF ITEMS ?" and input a numeric quantity into N.

```
INPUT A$,X
```

will print a "?" and input a string followed by a number.

NOTES:

1. Only constants may be input as data such as 100 or "ABC". Expressions are not allowed.

2. The INPUT command may only be executed from a program. If executed as a direct command an ILLEGAL DIRECT ERROR is given.

3. If a carriage return is the only character in an input line then execution of the program halts unless the flag at 2048 decimal (800 hex) is set to 128. If set, then receipt of just a carriage return will set the associated variable to a null string if it is string variable, or 0 if it is a numeric variable.

4. BASIC's input channel may be changed by POKEing the desired channel value at location 2052 decimal (804 hex).

5. If BASIC's output channel has been changed, possibly by an OUTCHAN command, then the prompt, if specified, will not be output.

6. Each character in a prompt will increment the output character count maintained by BASIC. If this count reaches the terminal width, specified when MTU BASIC was first entered, then an auto carriage return occurs. Since no carriage return is output with the prompt, this count must be reset using some other statement, such as a PRINT statement. Failing to reset this character count can cause a mysterious carraige return to occur. Refer to the POS() function for information about controlling the output character count.

7.4 THE OUTCHAN STATEMENT

PURPOSE: To change BASIC's output channel.

SYNTAX: OUTCHAN ⟨channel⟩

ARGUMENTS:

⟨channel⟩ = an expression which evaluates to a number from 0 to 9. The channel through which BASIC sends its output will be set to this number.

DISCUSSION:

Since BASIC performs all its needed I/O through CODOS channels, the user may may direct BASIC's output to any file or device by assigning it to the appropriate channel. The OUTCHAN statement allows BASIC's output to directed to some other channel while the normal output channel, channel 2, remains assigned to the console device.

EXAMPLES:

```
OUTCHAN 3
```

will set BASIC's output channel to channel 3.

NOTES:

1. Commands which output through BASIC's output channel are the LIST and PRINT commands. The "READY." and BASIC error messages will also be output through BASIC's output channel.

2. An "OUTCHAN 2" command is required to set BASIC's output channel back to channel 2.

3. The LIST command will automatically reset BASIC's output channel back to channel 2 when the listing is complete.

7.5 THE PRINT STATMENT

PURPOSE: To output data in ASCII.

SYNTAX: PRINT [<expression>] [, | ; <expression> | TAB(pos) | SPC(num)] ... [, | ;]

ARGUMENTS:

<expression> = an expression which evaluates to the data to be printed.

<pos> = an expression which evaluates to the column in which the printing of the next data item is to begin.

<num> = an expression which evaluates to the number of spaces to output.

DISCUSSION:

The PRINT statement outputs data in ASCII form via BASIC's output channel, normally channel 2. Each numeric expression output will have a preceeding space if it is positive, and a trailing space in all cases.

The number of spaces separating the output for each expression is controlled by the use of the ";" , "," , TAB(), and SPC() functions. The ";" does nothing, the "," tabs to the next 10 column field, the TAB() function tabs to the specified column, and the SPC() function outputs the specified number of spaces. If the PRINT statement ends with something other than a ";", ",", TAB(), or SPC(), then a carriage return will be output at the end of the data.

EXAMPLES:

Given that N = 100 and N$ = "ABC", then the following examples will operate as described:

```
PRINT CHR$(13);123,N$
```

will output a carriage return followed by the string " 123 ". Then it will tab to column 10, output "ABC" followed by a carriage return.

```
PRINT TAB(20);N,N*.1;
```

will tab to column 20 and output " 100 ". Then it will tab to column 30 and output " 10 ".

```
PRINT
```

will simply output a carriage return.

NOTES:

1. If the column specified in the TAB() function has already been passed, no tabbing occurs.

2. If the number of characters output reaches the terminal width specified in the start up procedure, an automatic carriage return is performed. Sometimes a carraige return can seem to come out of thin air because of character count reaching the terminal width. Refer to the POS() function for information about how the character count can be controlled.

7.6 THE READ STATEMENT

PURPOSE: To read data provided in DATA statements.

SYNTAX: READ <variable> [, <variable>]

ARGUMENTS:

<variable> = a BASIC variable which is to receive the read data.

DISCUSSION:

The READ statement is used to read the data declared in DATA statements. The statement will try to read data for each variable that appears in the statement.

The first READ statement executed will read its data from the first DATA statement in the program. Data will be read from this DATA statement, by the first and any other READ statements, until all data in the first DATA statement is exhausted. Reading then continues from the second DATA statement in the program, and so forth. The READ statement will search the program sequentially when it needs to find the next DATA statement. This means the DATA statement may be placed where convenient, as long the is given in the proper sequential order. An attempt to read more data than is provided in the program will result in an OUT OF DATA ERROR.

For information about specifying the data to be read, refer to the description for the DATA statement.

EXAMPLES:

```
100 READ  A,B(1,1),NM$
```

will read a numeric value into the variable A, and array variable B(1,1), and will read a string into variable NM$.

NOTES:

1. When the data read for a numeric variable is not a valid number, a SYNTAX ERROR will be given for the DATA statement in which the invalid data appears.

2. The RESTORE statement can be used to set the line from which the search for the next DATA statement will begin. The next READ statement will read its data from the first DATA statement which appears in or after the specified line. Refer to the RESTORE statement for details.

8. ARITHMETIC FUNCTIONS

MTU BASIC provides certain frequently used arithmentic functions for use in BASIC programs. They all accept one numeric argument and return a single numeric value. In the case of the user defined FN function, you can define those numeric functions not supplied as standard.

8.1 THE ABS FUNCTION

PURPOSE: To return the absolute value of a number.

SYNTAX: ABS(<expression>)

ARGUMENTS:

<expression> = a numeric expression which evaluates to the desired argument.

DISCUSSION:

The ABS function returns the absolute value of the specified number. The absolute value of a number is the magnitude of the number, i.e. same value with sign force to positive.

EXAMPLES:

```
10 PRINT ABS(-1)
```

will print the absolute value of -1, which is 1.

8.2 THE ATN FUNCTION

PURPOSE: To return the arctangent of the specified argument.

SYNTAX: ATN(<angle>)

ARGUMENTS:

<angle> = a numeric expression which evaluates to the argument of which the arctangent is taken.

DISCUSSION:

The ATN function is used to obtain the arctangent of the specified argument. The value returned is an angle given in radians. This value will be within the range PI/2 to -PI/2.

EXAMPLES:

```
10 AG = ATN(1)
```

will assign AG to the arctangent of 1, which is approximately .7854 radians or 45 degrees.

8.3 THE COS FUNCTION

PURPOSE: To return the cosine of a specified angle.

SYNTAX: COS(<angle>)

ARGUMENTS:

<angle> = a numeric expression which evaluates to the desired angle in radians.

DISCUSSION:

The COS function is used to obtain the cosine of a specified angle. This angle should be expressed in radians.

EXAMPLES:

```
10 PRINT COS(3.1416/4)
```

will print the cosine of one fourth of 3.1416 (45 degrees), which is approximately .707.

NOTES:

1. An angle in degrees may be converted to radians by multiplying by 180/3.141592654.

8.4 THE EXP FUNCTION

PURPOSE: To return the value of "e" (2.718281828) raised to a specified power.

SYNTAX: EXP(<power>)

ARGUMENTS:

<power> = a numeric expression which evaluates to the desired power.

DISCUSSION:

The EXP function is used to obtain the value of "e" (2.71828183) raised to the specified power. The maximum power that can be specified without overflow is 87.3365. *

EXAMPLES:

```
10 N=EXP(4)
```

will set N to the value of "e" raised to the fourth power.

8.5 THE FRE FUNCTION

PURPOSE: To return the number of unused bytes avaiable to your program.

SYNTAX: FRE(<dummy>)

ARGUMENTS:

<dummy> = an expression used as a dummy argument which is ignored.

DISCUSSION:

The FRE function returns the number of unused bytes available for program or data storage. The argument is present only because all arithmetic functions reguire a argument. The argument for this function isn't used for anything.

EXAMPLES:

```
10 PRINT FRE(0)
```

will print the number of free bytes available to your program.

8.6 THE FN FUNCTION

PURPOSE: To call a user defined function defined by a DEF statement.

SYNTAX: FN<name>(<expression>)

ARGUMENTS:

<name> = the function name specified in the associated DEF statement.

<expression> = a numeric expression which evaluates to the desired argument.

DISCUSSION:

The FN function is used to called a specified user defined function. This function must have be defined by previously executing a DEF statement for this function. The value returned by the user defined function is determined by the expression given with the DEF statement. Refer to the DEF statement for additional details.

EXAMPLES:

```
2000 A = FN AF(20)*2
```

will assign to A the value of the user defined function AF evaluated with an argument of 20, multiplied by 2.

8.7 THE INT FUNCTION

PURPOSE: To return the largest integer less than or equal to a specified number.

SYNTAX: INT(<expression>)

ARGUMENTS:

<expression> = a numeric expression which evaluates to the argument to take the INT function of.

DISCUSSION:

The INT function is used to obtain the largest integer less than or equal to the specified argument. If the argument is positive, the INT function returns the number with the fractional part removed, i.e. INT(1.1)=1. If the argument is negative, it returns the first integer below the negative value, i.e. INT(-1.1)=2.

EXAMPLES:

```
10 A=INT(N/6)
```

will set A to the number of times that N can be divided evenly by 6.

8.8 THE LOG FUNCTION

PURPOSE: To return the natural (base e) logarithm of the specified argument.

SYNTAX: LOG(<expression>)

ARGUMENTS:

<expression> = a numeric expression which evaluates to the argument of which the natural log is taken.

DISCUSSION:

The LOG function is used to obtain the natural logarithm of an argument. The argument must not be negative or zero, or an ILLEGAL QUANTITY ERROR will be given.

EXAMPLES:

```
10 LG = LOG(10)
```

will assign to LG the natural logarithm of the value 10, which is approximately 2.3026.

NOTES:

1. To obtain the logarithm of another base, use the formula:

logarithm-base-Y(X) = LOG(X)/LOG(Y)

8.9 THE PEEK FUNCTION

PURPOSE: To return the value of the byte at a specified memory location.

SYNTAX: PEEK(<address>)

ARGUMENTS:

<address> = a numeric expression which evaluates to the decimal address of the desired memory location.

DISCUSSION:

The PEEK function is used to obtain the value of a byte at a specified memory location. The address must be within the range 0 to 65535, or an ILLEGAL QUANTITY ERROR will be given. The locations read will always be in BANK 0.

EXAMPLES:

```
10PRINT PEEK(2048)
```

will print the contents of the byte at memory location 2048 ($800 hex).

8.10 THE POS FUNCTION

PURPOSE: To return the number of characters which have been printed on the current line.

SYNTAX: POS(<dummy>)

ARGUMENTS:

<dummy> = an expression used as a dummy argument which is ignored.

DISCUSSION:

The POS function is used to obtain the count of characters which have been output on the current line. This is typically used to determine the position of the cursor on the current line.

This count is derived from characters output through through BASIC's output channel. The count is set to zero whenever a carriage return (ASCII 13), form feed (ASCII 12), or cancel (ASCII 24, CTL-X) character is output. ASCII character codes from 32 to 255 will increment the count by 1. A rub out (ASCII 8) will decrement the count by 1. All other characters leave the count unchanged.

EXAMPLES:

```
10 PRINT CHR$(13);"ABC";: CP = POS(0)
```

will assign to CP the value three, which is the number of characters output on the current line, i.e. since the carraige return.

8.11 THE RND FUNCTION

PURPOSE: To return a random number between 0 and 1.

SYNTAX: RND(<expression>)

ARGUMENTS:

<expression> = a numeric expression which evaluates to the desired argument.

DISCUSSION:

The RND function is used to obtain random values. The value returned will be between 0 and 1, not including 0 or 1. If the specified argument is negative, a new random number is generated, using the argument as a seed to start the random number sequence. Specifying the same argument will cause the same random number sequence to be returned. If the argument is 0, the last random number is returned again. If the argument is positive, a new random number is generated in the current sequence.

EXAMPLES:

```
10 A=RND(-100)*10
```

will assign to A a random number between 0 and 10, starting a random number sequence based on the argument, -100.

```
20 N=RND(1)
```

will assign to N the next random number in the current sequence.

8.12 THE SGN FUNCTION

PURPOSE: To return the sign of a specified argument.

SYNTAX: SGN(<expression>)

ARGUMENTS:

<expression> = a numeric expression which evaluates to the argument of which the sign is taken.

DISCUSSION:

The SGN function will return a value of 1 if the argument is greater than zero, 0 if the argument is zero, and -1 if the argument is negative.

EXAMPLES:

```
10 X=-10:SX = SGN(X)
```

will assign to SX the value -1, since the X will contain the value -10.

8.13 THE SIN FUNCTION

PURPOSE: To return the sine of a specified angle.

SYNTAX: SIN(<angle>)

ARGUMENTS:

<angle> = a numeric expression which evaluates to the desired angle in radians.

DISCUSSION:

The SIN function is used to obtain the cosine of a specified angle. This angle should be expressed in radians.

EXAMPLES:

```
10 PRINT SIN(3.1416/4)
```

will print the sine of one fourth of 3.1416 (45 degrees), which approximately .707.

NOTES:

1. An angle in degrees may be converted to radians by multiplying by 180/3.1415926536.

8.14 THE SQR FUNCTION

PURPOSE: To return the square root of a specified value.

SYNTAX: SQR(<expression>)

ARGUMENTS:

<expression> = a numeric expression which evaluates to the value of which the square root is taken.

DISCUSSION:

The SQR function is used to return the square root of a specified value. The specified value should not be negative, or an ILLEGAL QUANTITY ERROR will be given.

EXAMPLES:

```
10 PRINT SQR(16)
```

will print the square root of 16, which is 4.

8.15 THE TAN FUNCTION

PURPOSE: To return the tangent of a specified angle.

SYNTAX: TAN(<angle>)

ARGUMENTS:

<angle> = a numeric expression which evaluates to the desired angle in radians.

DISCUSSION:

The TAN function is used to obtain the tangent of a specified angle. This angle should be expressed in radians.

EXAMPLES:

```
10 PRINT TAN(3.1416/4)
```

will print the tangent of one fourth of 3.1416 (approximately 45 degrees). The value printed should be close to 1.

NOTES:

1. An angle in degrees may be converted to radians by multiplying by 180/3.1415926536.

8.16 THE USR FUNCTION

PURPOSE: To obtain a floating point value from a machine language subroutine.

SYNTAX: USR(<address> ...)

NOTE: If desired, the machine language subroutine may accept additional arguments following the address argument.

ARGUMENTS:

<address> = a numeric expression which evaluates to the decimal address of machine language subroutine.

DISCUSSION:

The USR function is used to execute a machine language subroutine at the specified address. This machine language subroutine is expected to leave a floating point value in in BASIC's foating point accumulator (FLTACC). This value will be returned as the value of the function.

The machine language routine is resposible for fetching any arguments it requires. These arguments, if present, should be separated by commas. This is easier than you might think. Refer to the section on User Interface Routines in the WRITING A BASIC LIBRARY manual.

EXAMPLES:

The following machine language routine will fetch an argument, convert it to a two byte integer, and return the value of the high byte of that integer.

```
HIBYT   JSR CHKCMV     ; REQUIRE A COMMA AFTER THE ADDRESS, SYNTAX ERROR IF
        LDA #$C0       ;      NOT FOUND
        STA FCHTYP     ; SET TO FETCH A NUMERIC ARGUMENT
        LDA #$40
        STA RETTYP     ; SET TO RETURN THE ARGUMENT AS A TWO BYTE INTEGER
        JSR UFPARM     ; FETCH THE ARGUMENT INTO X AND A REG.
        STX FLTACC+4   ; STORE HIGH BYTE FETCHED IN LOW BYTE OF RETURN VALUE
        LDA #0         ;
        STA FLTACC+3   ; SET HIGH BYTE OF RETURN VALUE TO ZERO
        STA ARGTYP     ; SETUP TO CONVERT THE RETURNED INTEGER TO
        LDA #$80       ;    A FLOATING POINT NUMBER
        STA NUMTYP
        STA RETTYP
        JSR UCONV      ; CONVERT THE INTEGER TO A FLOATING POINT NUMBER
        RTS            ; RETURN TO BASIC
```

For the addresses of these routines and locations, refer to the WRITING A BASIC LIBRARY manual.

If the above code is found in memory at location 40960 ($A000), then the following example will operate as described.

```
10 PRINT USR(40960,N)
```

will print the high byte of the value in N, interpreted as a two byte integer. If N contains 512, the value printed would be 2.

NOTES:

1. Upon return from the machine language subroutine, the value in the floating point accumulator is checked to make sure it is numeric. A TYPE MISMATCH ERROR is given if it is not. To return a string value, this check may be bypassed by popping one return address (i.e. two PLA's) from the top of the 6502 stack.

9. STRING FUNCTIONS

A string, as decribed in Section X.X, is a series of characters. A string may be from 0 to 255 characters in length. A variable name is designated as a string variable by placing a "$" at the end of the name. The functions described in this section return values and information pertaining to strings.

9.1 THE ASC FUNCTION

PURPOSE: To return the ASCII numeric value of the first character in a string.

SYNTAX: ASC(<string>)

ARGUMENTS:

<string> = a string expression of the desired string.

DISCUSSION:

The ASC function returns the ASCII numeric value of the first character in the specified string argument. If the specified string is a null string, an ILLEGAL QUANTITY ERROR will be given.

EXAMPLES:

```
10 PRINT ASC("A")
```

will print the ASCII code for the character "A", which is 65.

```
100 CH=ASC(Z$)
```

will assign to the variable CH the ASCII value for the first character in the string contained in Z$.

9.2 THE CHR$ FUNCTION

PURPOSE: To return a one character string whose single character is the ASCII equivalent of a specified value.

SYNTAX: CHR$(<value>)

ARGUMENTS:

<value> = a numeric expression which evaluates to the desired character value. It must be a number from 0 to 255.

DISCUSSION:

The CHR$() function is used to covert a numeric value to a string containing the equivalent ASCII character. This string will have a length of 1. If the value specified is outside the range 0 to 255, an ILLEGAL QUANTITY ERROR is given. This function is typically used to obtain strings containing characters which are difficult to place directly in program statements , such as control codes.

EXAMPLES:

```
10 A$=CHR$(13)
```

will assign to A$ a string whose contents is the ASCII character equivalent of 13. This character is a carriage return control code.

9.3 THE LEFT$ FUNCTION

PURPOSE: To return a string containing the leftmost portion of another string.

SYNTAX: LEFT$(<string> , <characters>)

ARGUMENTS:

<string> = a string expression for the string to take the leftmost portion of.

<characters> = a numeric expression for the number of leftmost characters to return.

DISCUSSION:

The LEFT$ function will return a string containing the leftmost N characters of the specified string, where N represents the value given in the numeric expression. If the specified string doesn't contain N characters, all of the specified string is returned.

EXAMPLES:

```
100 PRINT LEFT$("ABCDEF",3)
```

will print the leftmost 3 characters in "ABCDEF", i.e. "ABC".

9.4 THE LEN FUNCTION

PURPOSE: To return the length of a specified string.

SYNTAX: LEN(<string>)

ARGUMENTS:

<string> = a string expression of which the length is to be returned.

DISCUSSION:

The LEN function is used to find the length of a specified string. Non-printing characters and spaces are counted as part of the string.

EXAMPLES:

```
10 N=LEN(A$)
```

will set the variable N to the length of the string in A$.

9.5 THE MID$ FUNCTION

PURPOSE: To return a string which is a middle portion of a specified string.

SYNTAX: MID$(⟨string⟩ , ⟨position⟩ , ⟨characters⟩)

ARGUMENTS:

⟨string⟩ = a string expression from which to take a middle portion.

⟨position⟩ = a numeric expression which evaluates to the character position of desired portion.

⟨characters⟩ = a numeric expression which evaluates to length of the desired portion.

DISCUSSION:

The MID$ function is used to obtain a string containing a middle portion of a specified string. If two arguments are specified, the portion returned will start at the specified position and contain the remainder of the specified string.

If three arguments are specified, the portion returned will start at the specified position and contain the number of characters specified. If the number of characters specified is more than the number of remaining characters, then only the remainder of the string is returned.

In either case, a null string is returned if the specified postion is beyond the end of the string.

EXAMPLES:

```
10 PRINT MID$("ABCDEF",4)
```

will print the portion of the string "ABCDEF" that begins with the fourth character and contains the remainder of the string, i.e. "DEF".

```
10 PRINT MID$("ABCDEF",3,2)
```

will print the portion of the string "ABCDEF" that begins with the second character and is two characters in length, i.e. "CD".

9.6 THE RIGHT$ FUNCTION

PURPOSE: To return a string containing the rightmost portion of another string.

SYNTAX: RIGHT$(<string>, <characters>)

ARGUMENTS:

<string> = a string expression for the string of which the rightmost portion is to be returned.

<characters> = a numeric expression for the number of rightmost characters to return.

DISCUSSION:

The RIGHT$ function will return a string containing the rightmost N characters of the specified string, where N represents the value given in the numeric expression. If the specified string doesn't contain N characters, then a null string is returned.

EXAMPLES:

```
100 PRINT RIGHT$("ABCDEF",3)
```

will print the rightmost 3 characters in "ABCDEF", i.e. "DEF".

9.7 THE STR$ FUNCTION

PURPOSE: To return a string containing the ASCII character representation of a specified value.

SYNTAX: STR$(<expression>)

ARGUMENTS:

<expression> = a numeric expression which evaluates to the value to be converted to character representation.

DISCUSSION:

The STR$ function is used to convert a number to a string. If the specified value to convert is positive, then the string will contain a leading blank. If it is negative, the first character will be the minus sign.

EXAMPLES:

```
10 A$=STR$(3.1)
```

will assign the string " 3.1" to A$

```
10 T$="TOTAL="+STR$(T)
```

will assign a string to T$ which contains "TOTAL=" followed by the character representation of the value of T.

9.8 THE VAL FUNCTION

PURPOSE: To return the numeric value of number represented in character form in a string.

SYNTAX: VAL(<string>)

ARGUMENTS:

<string> = a string expression to convert to a numeric value.

DISCUSSION:

The VAL function is used to convert a string into a number (the opposite of the STR$ function). If the first non-space character in the string is not a "+", "-", digit, or decimal point, then the VAL function will return a 0 value.

EXAMPLES:

```
10 N=VAL("-4.5")
```

will assign the value -4.5 to N.

10. BASIC OPERATORS

This section describes the various operators that are available for use in a BASIC expressions. These operators are divided into 4 types, arithmetic, relational, logical, and string expressions, which are described below.

10.1 Arithmetic Operators

The six arithmetic operators available are:

SYMBOL	FUNCTION	EXAMPLE
-	Negation	-51, X=-A
+	Addition	Z=R+T+Q
-	Subtraction	J=100-I
*	Multiplication	X=R*B*D
/	Division	C=X/1.3
^	Exponentiation	X^3

Note that negation is quite separate and distinct from subtraction. When you use the minus sign to negate a variable or a number, BASIC treats such usage as zero minus the variable or number. For example, X=-Y is processed a X=0-Y by BASIC. However, X=10-Y is an example of the arithmetic operation subtraction, and as such takes its usual place in the precedence of operations.

10.2 Relational Operators

Relational operators are used to compare two values. Such comparisons can be combined to form a relational expression. The result of such expressions is value which indicates whether the relation specified in the expression is true or false. Most often, these expressions are found in IF..THEN statements. The six relational operators are:

MATH SYMBOL	BASIC SYMBOL	EXAMPLE	MEANING
=	=	X=Y	X is equal to Y
<	<	X<Y	X is less than Y
≤	<=,=<	X<=Y	X is less than or equal to Y
>	>	X>Y	X is greater than Y
≥	>=,=>	X>=Y	X is greater than or equal to Y
≠	<>,><	X<>Y	X is not equal to Y

The value returned by a relational expression will always be -1 for true, and 0 for false. For example, (4=7) = 0, (4=4) = -1, (7<4) = 0, (7>4) = 1. When applied to strings, the relational operators test alphabetic sequence. Comparison is made on a character by character basis according to the ASCII codes until a difference is found. If the end of one string is reached before a difference is found, the shorter string is considered smaller than the other string.

10.3 Logical Operators

Logical operators are used for bit manipulation and for performing Boolean operations. These three operators convert their arguments to sixteen bit, signed two's complement integers in the range -32768 to +32767. They then perform the specified logical operation on them and return a result within the same range. If the arguments are not in this range, an ILLEGAL QUANTITY ERROR will be given.

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 tables show the logical relationship between bits:

AND

A	B	A AND B
1	1	1
1	0	0
0	1	0
0	0	0

OR

A	B	A AND B
1	1	1
1	0	1
0	1	1
0	0	0

NOT

A	NOT A
1	0
0	1

To determine the results of logical operations on relative expressions, interpret the tables above with 1 as true, and 0 as false. For example, true AND true = true.

10.4 String Operator

There is one string operator. The "+" symbol when used between string arguments represents string concatenation. A new string is created which consists of the characters of the first string followed by the characters of the second string.

10.5 Priority of Operators

The priority of operators controls which operators are executed before others. For example, in the expression: X+Y/Z, is X added to Y before dividing by Z? The answer is no. Division is given higher priority than addition, so Y is divided by Z first, and the result added to X.

When more than one operation is performed in a single formula, BASIC evaluates each portion using the following schedule of priority:

1. Parentheses - any expression enclosed in parentheses is always evaluated first.
2. Exponentiation
3. Negation
4. Multiplication and Division (of equal priority)
5. Addition and Subtraction (of equal priority)
6. Relational operators (all of equal priority)
7. Logical operators in the order NOT, AND, then OR.

If, in any expression, the above rules do not clearly designate the order of priority, then evaluation proceeds from left to right.

11. RESERVED VARIABLES

MTU BASIC uses the following reserved variables. By reserved it is meant that these variables are always defined by the system and cannot be assigned to a value by the BASIC prgoram itself

11.1 THE ST VARIABLE

The ST variable is a reserved variable used by external software to return a status byte to BASIC. When used in a expression, it returns the value of the byte at location 119 decimal (77 hex). This location is interpreted a signed byte, so the values returned will range from -128 to 127. None of the commands in the standard command set affect this byte.

11.2 THE KEY VARIABLE

The KEY variable is used to obtain the status of the 8 function keys. Whenever the vaiable "KEY" or "KE" is used in a expression, the status of the keyboard is read. If no key is down, or a key other than a function Key is down, then 0 is returned. If a function key is down, a number from 1 to 8 is returned. In this case the number returned corresponds to which function key is down, 1 for the F1 key, etc. Once a key has be detected as being down, the KEY variable will return a value of 0 until it is released. The operation of the KEY variable makes it ideal for use in ON...GOTO and ON...GOSUB commands.

After an error occurs, MTU BASIC gives an error message indicating the type of error. BASIC then returns to the READY state. Variable values and the program text remain intact, but the program cannot be continued and all GOSUB and FOR context is lost. The following is a list of the various error messages that can occur, and their possible causes.

BAD SUBSCRIPT

Attempt was made to reference an array element which is outside the dimensions of the array. This error can occur if the wrong number of dimensions is used in an array reference; for instance, A(1,1,1)=Z when A has been dimensioned DIM A(2,2)

CAN'T CONTINUE

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

CAN'T LOAD LIBRARY

The LIB command was unable to load a particular library into memory due to conflicts with libraries already in memory. Library files are permitted to contain multiple copies of a library program, each to load at a different location. Memory allocation routines associated with the LIB command will try to find a copy of a library which won't have any memory conflicts with those already in memory. For more information about dealing with this error, refer to the LIB command.

DIVISION BY ZERO

A division by zero occurred in an arithmetic expression.

FILE DATA

An error occurred in the data read by an INPUT statement and BASIC's input channel had been changed to a channel other than channel 1.

FILE EXISTS

The output file specified with a SAVE or LIST command already exists.

FILE NOT FOUND

The input file specified with a LOAD or ENTER command wasn't found.

FILE TOO LARGE

The input file specified with a LOAD statement is too large to fit into the amount of free memory currently available for BASIC programs.

FORMULA TOO COMPLEX

An arithmetic expression was too complex. Break it into two or more shorter ones.

ILLEGAL DIRECT

An INPUT, GET, or DEF statement was used as a direct command.

ILLEGAL QUANTITY

The parameter passed to a mathematical or string function was out of range. This is commonly due to:

1. a negative array subscript
2. an array subscript greater than 32767
3. LOG function with a negative or zero argument
4. A B with A negative and B not an integer.
5. function calls with an argument out of range.

INTEGRITY

The MTU BASIC interpreter has reached an illegal state. This error should never occur during normal use. If you encounter this error, please note the circumstances in which it occurred and call MTU.

LIST

A line being listed was found to be longer than 255 characters. If some portion of a BASIC program is accidentally overwritten in memory, or on disk, this error could occur when listing the program.

LOAD

An error occurred during the loading of a program. Most likely, the end of the file was encountered prematurely.

NEXT WITHOUT FOR

The variable in a NEXT statement does not match the most recently executed FOR statement. Also, the NEXT statement occurred without a previously executed FOR statement.

NOT A BASIC FILE

The data at the beginning of a file being LOADed was not that of a BASIC program.

NOT A LIBRARY FILE

The data at the beginning of a file being loaded by a LIB command was not that of a library file.

NOT LOADED

A token for a library which is not linked was encountered during program execution, or during the listing of a program. BASIC will attempt to look up the library name in the disk file SYSLIBNAM.Z. If successful, the library name will be printed followed by this error message. If not, only this error message is printed.

OUT OF DATA

A READ statement was executed but all of the DATA statements in the program have already been read. The program tried to read too much data, or insufficient data was included in the program.

OUT OF MEMORY

Program too large, too many variables, too many FOR loops, too many GOSUBs, expression too complicated, or any combination of the above. This may occur if you are using very large arrays. It can also occur if you fail to terminate all FOR..NEXT loops or subroutines. These unterminated FOR..NEXT loops or subroutines gradually build up until the OUT OF MEMORY ERROR occurs.

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 printing any error message.

REDIM'D ARRAY

After an array was demensioned, another dimension statement for the same array was encountered. This error often occurs if an array has been given the default dimension 10 because the array was reference prior to the DIM statement which dimensions it.

RETURN WITHOUT GOSUB

A RETURN statement was encountered without a previous GOSUB statement having been executed.

STRING FORMULA TOO COMPLEX

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

STRING TOO LONG

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

SYNTAX ERROR

Missing parenthesis in an expression, illegal character in a line, incorrect punctuation, etc. This can also occur if commands from a library are entered from the keyboard or ENTERed from a file before the library is loaded with a LIB command.

TOO MANY LIBRARIES

The maximum of eight simultaneously linked libraries has been reached.

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. Also, an argument of an expected type was given a value of a different type.

UNDEF'D FUNCTION

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

UNDEF'D STATEMENT

An attempt was made to THEN, GOSUB, or GOTO a statement which does not exists.

13. SPECIAL CODES AND CHARACTERS

MTU BASIC makes use of a few special control codes and characters. These are the CNTL-C control code, the colon (:), and the question mark (?). Their use is as follows:

CNTL-C Halts execution of a BASIC program, or LIST command. If CNTL-C is pressed during execution of a LIST command, output of the listing will terminate at that point. If CNTL-C is pressed during execution of a program, BREAK IN LINE XXXX is printed, where XXXX is the line number of the next statement to be executed. BASIC then returns to the READY mode.

: A colon is used to separate multiple statements on a line. Colons may be used in direct commands, or in program lines. The only limit on the number of statements per line is the line length. By placing multiple statements per line in a program, you will save memory space, and speed execution. Also, it is not possible to GOTO or GOSUB to the middle of a line.

? The question mark is used as a abbreviation of the reserved word PRINT. For instance, executing ?2+2 is the same as executing PRINT 2+2. This is done to provide a quick way to enter PRINT statements. If not enclosed within quotes, a "?" will always be interpreted as an abbreviation for the reserved word PRINT. If entered in a program line, you will note that "PRINT" is spelled out when that line is later listed.

APPENDIX A

STANDARD KEYWORD LIST

The following is a list of all the keywords in the standard list of BASIC commands, function, and operators with the hexadecimal value of their tokens.

END	80	+	AD
FOR	81	-	AE
NEXT	82	*	AF
DATA	83	/	B0
INPUT	84	^ (exp)	B1
DIM	85	AND	B2
READ	86	OR	B3
LET	87	>	B4
GOTO	88	=	B5
RUN	89	<	B6
IF	8A	SGN	B7
RESTORE	8B	INT	B8
GOSUB	8C	ABS	B9
RETURN	8D	USR	BA
REM	8E	FRE	BB
STOP	8F	POS	BC
ON	90	SQR	BD
WAIT	91	RND	BE
LOAD	92	LOG	BF
ENTER	93	EXP	C0
SAVE	94	COS	C1
DEF	95	SIN	C2
POKE	96	TAN	C3
PRINT	97	ATN	C4
CONT	98	PEEK	C5
LIST	99	LEN	C6
CLEAR	9A	STR$	C7
MCALL	9B	VAL	C8
OUTCHAN	9C	ASC	C9
BYE	9D	CHR$	CA
LEGEND	9E	LEFT$	CB
GET	9F	RIGHT$	CC
LIB	A0	MID$	CD
FRELIB	A1	GO	CE
EDIT	A2		
TONE	A3		
NEW	A4		
(Reserved)	A5		
TAB(	A6		
TO	A7		
FN	A8		
SPC(	A9		
THEN	AA		
NOT	AB		
STEP	AC		