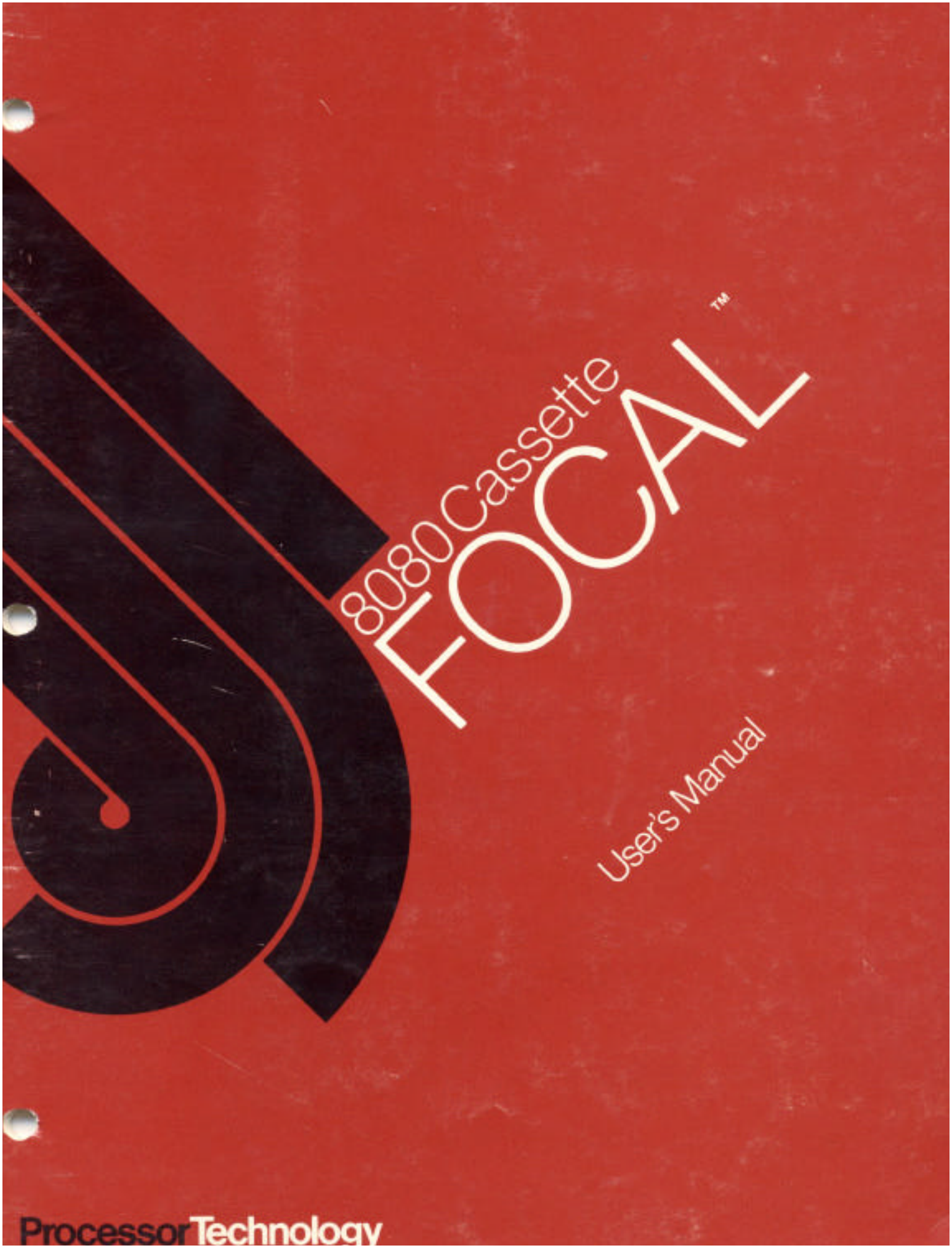
8080 Cassette
FOCAL™
User's Manual
Processor Technology

8080 FOCAL™

User's Manual

Processor Technology Corporation
7100 Johnson Industrial Drive
Pleasanton, CA 94566
Telephone (415) 829-2600

First Printing, April, 1978
Manual Part No. 727025

IMPORTANT NOTICE

TABLE OF CONTENTS

1. INTRODUCTION

FOCAL is the name of a computer language as well as the name of the program which translates and executes programs written in that language. The program, FOCAL, belongs to a class of language processors called "interpreters," and this means that FOCAL, while operating, has complete control of the machine, and thus can assist in storing, editing and running programs. Externally, FOCAL communicates with a user through an input/output device like a teletype. Internally, it divides up memory into three sections containing the FOCAL program itself, the user's stored program and any variables the user may have created. A minimum memory size of 8K is necessary for FOCAL, and additional memory allows FOCAL to store larger programs and more variables. The additional machine requirements are described in the section called "Running FOCAL."

The user controls FOCAL by typing a line of characters followed by a carriage return. The input line can be a command to FOCAL which it must execute immediately, or it could be a program line to be stored for later execution. These types of input lines can be intermixed as there is no interference between them. An input line, which is to be stored as a program statement must begin with a number identifying its location within the program. A line without a number is not stored but executed immediately. FOCAL determines the order of execution for lines in the stored program from their line numbers, not the order they were typed, which makes adding or replacing lines during a debugging session very easy. The following sequences, listed as they were input, will both result in the same sequence of calculations.

```
*1.0  SET A=1
*2.0  SET B=A*3.2
*2.21 SET C=A+B
*4.0  SET X=A+C/2

*2.21 SET C=A+B
*1.0  SET A=1
*4.0  SET X=A+C/2
*2.0  SET B=A*3.2
```

The digits to the left of the decimal point in a line number make up what is called the GROUP NUMBER, and can be used in some statements to identify a block of statements. The digits to the right of the decimal are known as the STEP NUMBER and these have no special significance; step numbers can be assigned values in the range 01 to 99; 0 and 00 are illegal. Groups of statements will later be referenced by certain commands by a group number followed by a zero step number. The reader should note that there is no single line in the program with a zero step number; this number refers to the entire block having that group number.

Most computer systems draw a distinction between commands and statements. Commands are input lines given to a program, usually called an "operating system," which controls the entire machine. Statements, on the other hand, are lines written in a strictly defined language, and these are interpreted by a program subordinate to the operating system; this program could be a "compiler," an "assembler" or an "interpreter." FOCAL is unusual in that it has the functions of both an operating system and an interpreter, and this gives it an enormous flexibility. It can handle both statements from the language FOCAL and commands of a supervisory nature. This distinction between operating systems and interpreters, statements and commands is further weakened because FOCAL allows statements from the language to be executed immediately much like commands in other machines. This manual will put little emphasis on the differences between commands and statements; in fact, the terms will be used almost interchangeably.

At any given instant FOCAL will be in one of three states or operating modes READY, EXECUTION or PROGRAM INPUT. FOCAL enters the READY mode after it finishes the last command given to it, and as it enters this state it issues an asterisk (*) to the user's terminal. It remains in this mode until the user has typed an input line followed by a carriage return. In the short sequences given on the first page of the introduction, the asterisks were supplied by FOCAL and the user typed the remainder of the line shown. The carriage return, needed at the end of every input line, forces FOCAL to leave the ready mode and enter the EXECUTION mode to perform the actions specified by the input line. This may mean that it only has to store the line as part of the program under construction. The input line could also make FOCAL execute the input line as

though it were a part of a program; it could ask FOCAL to perform some supervisory function, or it could have FOCAL execute the stored program.

The last mode, PROGRAM INPUT, occurs when FOCAL encounters and executes a special instruction, ASK, which will be described in full later. At that time FOCAL issues a question mark to the terminal and waits, as in the READY mode, until the user enters a number followed by a carriage return. The difference between this mode and READY is that the user is "talking" to his program through FOCAL. The execution mode is automatically re-entered after the input is completed.

1.1 GO and QUIT Commands

The stored program begins execution when FOCAL is given the GO command. This execution begins at the lowest line number in the stored program and will proceed from there to higher line numbers as the program logic allows. The running program can be stopped by the user, by FOCAL or by the program itself; regardless of the reason, control will pass to the "READY" mode of FOCAL at the conclusion of the run.

The program can stop itself with the QUIT statement, which can be stored anyplace, any number of times, throughout the stored program. The instant this statement is executed the program stops and returns control to FOCAL which will issue a ready prompt to signal the end of the program. The program is not altered in any way by running so that it can be immediately re-run if desired. Examples of the GO and QUIT statements appear all throughout this manual.

In the event the program begins acting in some undesirable way, the user can halt the program by typing "MODE" (this can also be done by hitting the "CNTL" key and the "@" simultaneously). FOCAL should respond with the READY prompt; if not, the program has managed to annihilate part of the program FOCAL, and FOCAL will have to be reloaded.

FOCAL will stop a program, if an error is discovered, or the program executes its highest numbered line and does not jump back into the rest of the program. In this latter case, FOCAL tries to find a higher numbered line, and failing it returns to the ready mode with an asterisk. Errors terminate a job with question marks and an error code which can be deciphered with the table given in a later section.

2. CONVENTIONS

2.1 Numbers

All numbers in FOCAL are internally treated as floating point numbers occupying four memory bytes apiece. The largest number which can be represented by FOCAL is 3.6 times 10 to the 38th power, and the smallest non-zero positive number is 2.7 times 10 to the minus 39th power. This same range applies to negative numbers as well.

The accuracy for a number anywhere in this range is limited to approximately 7 decimal places making 850.0000 equivalent to 850.00003. Any number can be given to FOCAL as an integer (without a decimal point) as a floating point number (a number containing a decimal), or as a number in scientific notation. Numbers in the scientific notation format consist of a mantissa and an exponent; the mantissa is written in decimal form followed by an E, followed by the exponent value. In the scientifically formatated number, -7.2E-11, the number -7.2 is the mantissa and -11 is the exponent. The value of this number is -7.2 times 10 to the -11th power or -.000000000072. Any form of number input may be signed (+ or -) or unsigned.

In FOCAL the following numbers are equivalent:

```
   700.3240
   700.32403
     7.003240E2
     7.00324E+2.0
     7.00324E01.0
    70.0324E01.0
700324.0E-3.0
      .0700324E4
```

All numbers printed by FOCAL can contain up to 7 decimal digits (excluding the sign).

2.2 Variables

A variable is a uniquely named storage location having an associated arithmetic value. Its name consists of a sequence of letters and/or numbers, the first character being any letter other than F. (Names beginning with F are assumed to be function names.)

FOCAL variable names are unique in their first two characters only. Thus, the variable names SA, SAM and SAMMY refer to the same storage location.

```
LEGAL NAMES          FOCAL RECOGNIZES

 KNOCK                    KN
 PROFIT                   PR
 COST1                    CO   (Thus COST1 and COST2
 COST2                    CO    are the same!)

ILLEGAL NAMES        REASON

3ARM                 First character must be
                     be alphabetic.
FOOT                 First character must
                     not be F.
```

To facilitate storage of large amounts of information, FOCAL allows variables to be subscripted. This feature will be described later.

2.3 Evaluating Expressions

FOCAL, which is a contraction of FOrmula CALculator, allows the user to construct arithmetic expressions or formulas using the following symbols:

```
^     EXPONENTIATION
*     MULTIPLICATION
/     DIVISION
+     ADDITION
-     SUBTRACTION
```

8080 FOCAL is similar to DEC FOCAL in that the evaluation of arithmetic expressions proceeds according to standard operator priority. This priority follows the table above with the " ^" operation having the highest priority.

Occasionally it becomes necessary, due to the complexity of an expression, to NEST parts of the equation in parentheses. Just as a single pair of parentheses reorder the sequence of calculations in the above example, the "sub-expression" within parentheses can be reordered by separating its parts with parentheses. For instance,

```
*SET   X=CA*A+B*C+B/2.6
*SET   X=(CA*A+B)*(C+B/2.6)
*SET   X=(CA*(A+B))*((C+B)/2.6)
```

all contain legal expressions. Each will, however, use a different sequence of multiplications, additions, etc., which will produce a different value for the variable "X". The internal sequence of steps for evaluating the last of the above examples would be:

```
1. ADD A TO B.  PUT SUM IN TEMPORARY LOCATION "1" (TEMP)
2. ADD C AND B.  SUM TO TEMP "2"
3. DIVIDE VALUE IN TEMP2 by 2.6 AND STORE QUOTIENT IN TEMP2
4. MULTIPLY CA BY TEMP1.  PRODUCT STORED IN TEMP 1
5. MULTIPLY TEMP1 BY TEMP2.  PRODUCT STORED IN X
```

The level of a nest, or "Level number," is equal to the number of left parentheses minus the number of right parentheses found to the left of the term in question. FOCAL, as shown, evaluates the terms with the highest level number first, and works down from there. Any level of nesting is allowed, as long as the statement occupies only one line.

2.4 Math Functions

FOCAL provides eleven standard math functions along with a user-defined assembly language function. A function is a routine internal to FOCAL which performs an arithmetic calculation on a value called an "argument," which is given to it. This argument must be enclosed in parentheses immediately following the functions's name. FOCAL's generality permits this argument to be a constant, variable or expression. The following is a list of the function names recognized by FOCAL (throughout this list, the character "X" represents the argument):

```
FUNCTION    USE

FABS(X)     ABSOLUTE VALUE
FSGN(X)     "SIGN" OF X.  VALUE RETURNED IS -1 WHEN X
            IS NEGATIVE, 0 WHEN X=0, AND +1 WHEN X>0
FITR(X)     INTEGER PART OF X
FRAN(X)     RANDOM NUMBER BETWEEN .5 and 1.0 WITH RANDOM SIGN
FATN(X)     ARC TANGENT
FEXP(X)     EXPONENTIAL -- E^X
FLOG(X)     NATURAL LOG
FSIN(X)     SIN OF X
FCOS(X)     COSINE OF X
FSQT(X)     SQUARE ROOT OF ABSOLUTE VALUE OF X
FHYS(X)     HYPERBOLIC SIN
FUSR(X)     USER DEFINED.  WHEN UNDEFINED, THIS
            RETURNS THE VALUE OF THE ARGUMENT UNCHANGED
```

In a statement of the form SET Y= FSIN(PHI), the variable Y is given the computed value of the sine of the angle, "PHI". All of FOCAL's trigonometric functions assume that their arguments are given in radians (FATN returns a radian value from minus pi to + pi). To convert degrees to radians simply divide by 57.29579.

2.5 Command Line Interpretation

FOCAL allows and even encourages the programmer to put more than one statement on an input line. The additional commands (statements) need to be separated by semicolons, and in one case, the "FOR" statement, the entire line must be ended with a semicolon. This multiple statement line feature can be used in both program store mode and immediate mode.

FOCAL also allows, for efficiency's sake, abbreviated commands; thus, for example, "SET," "GO" and "QUIT" could all be written as "S," "G" and "Q." Internally all commands are identified by their first letter only, so that "SHAKE", "GASP" and "QUAKE" could be used for "SET," "GO" and "QUIT." All of these statements and their functions will be described shortly.

To simplify the command recognition process, the FOCAL language has been constructed with a mild emphasis on blanks. In the commands to be discussed in the upcoming sections, all must begin with an easily recognized word (or abbreviation) like "SET," "GO," "ASK" and "Q." This word, called a KEYWORD, must be followed by at least one blank. This is a common source of error for people new to the language. Line numbers in stored programs also require a trailing blank for the same reason.

Most of the FOCAL language statements will expect a sequence of characters following the keyword and its blank. The form and content of this part of the statement will depend on the command in question. Those commands not requiring more than the keyword will ignore anything between the keyword and the next semicolon or carriage return. Telling FOCAL to "*QUIT YER COMPLAININ"' will only cause it to QUIT. Telling it to -*GO TO HEAVEN" (?) will cause an error, because "TO HEAVEN" is not the line number expected by the "G" or "GOTO" command. For the same reasons, "GO TO 5.1" will force an error because "TO 5.1" is not a legal line number. This command will be described later.

3. THE SET COMMAND

The most fundamental command in FOCAL is the SET command. In its general form it looks like:

```
*<line number> SET <variable>  =  <expression>
```

This command can also be used in the immediate mode by omitting the line number.

The variable names (X, Y, Z and D in the example below) are defined by FOCAL and are given values (25.1, -6.88, etc.). A memory location is associated with the variable name. If the specified variable name has not yet been encountered by FOCAL, a new memory location is set aside and is associated with the name.

```
*10.4  SET Z=12.01
*10.5  SET X=25.1
*10.6  SET Y=-6.88
*10.7  SET D=FSQT((X*X)+(Y*Y)+(Z*Z))
*GO
*
```

After the program has been run and FOCAL returns to the ready mode, the memory location for the variable D (from above) contains the value of the expression FSQT((X*X)+ (Y*Y)+ (Z*Z)).

The execution of the SET command produces the same results whether the command was stored and executed, or executed in immediate mode. It is often very effective to use the command in both ways during a series of runs with a program. Before the RUN command is given, the controlling values for the problem can be set or defined allowing the program to be very general. The following illustrates a typical sequence of runs using this feature.

```
*1.1   SET A=FSQT(B/C+B^3)*D                          INPUT THE
*1.2   SET . . . . . . . . . .                        PROGRAM
* .
* .
* .
(generalized program)
* .
* .
* .
*99.9   QUIT
*SET    B=10.1; SET C=12.77; SET-D=60                 SET PARAMETERS
*GO                                                   RUN THE PROGRAM
(results)                                             EXAMINE RESULTS
*SET    B=10.6; SET C=11; SET D=100                   MODIFY PARAMETERS
*GO                                                   RERUN THE PROGRAM
(results)                                             EXAMINE
```

4. INPUT/OUTPUT COMMANDS

4.1 The Type Command

Every FOCAL program must contain at least one TYPE statement if it is to produce printed results. The TYPE statement prints values of variables, text strings and results of expressions. These can be combined using commas to separate the items into a list. The following example shows several TYPE statements and their resultant printout:

```
*SET A=1;SET B=2;SET AB=-6
*TYPE A,#
 1.000000
*TYPE A,B,AB,#
 1.000000 2.000000-6.000000
*TYPE "QUOTATION MARKS START AND END TEXT"
QUOTATION MARKS START AND END TEXT:TYPE A,#
 1.000000
*TYPE !,"NOTE HOW THE # AND ! WORK",#
NOTE HOW THE #  AND ! WORK
*TYPE A, !,B,#
 1.000000

 2.000000
*TYPE %5.02,AB,"hi",12345.5456
 -6.00hi2345.55:
```

The examples are in immediate mode where their results were immediately visible. The only modification for program storage would be the addition of line numbers. It is important in the examples to watch how the special characters comma, #, ! and " are used.

The % begins a field description of the form %w.0d which describes how numbers should be printed. The w is the width to be used (maximum no. of digits), and the d is the number of these digits which are to appear after the decimal point. The 0 is required. In the example above, note that truncation occurs if the number exceeds the field width. (Six digits were retained because there is room for a sign.) The field description remains in effect for all TYPE statements until another field description is seen.

The most readable output is usually made by combining text with printed values. The program can thus identify a value as well as print its value. The following shows this feature used in program store mode.

```
*1.1   SET G=32;SET T=5;SET D=.5*G*(T^2)
*2.1   TYPE "FOR ACCELERATION",G,"AND TIME",T,"SECONDS",#
*2.51  TYPE "AN OBJECT FALLS",D,"FEET.",#
*GO
FOR ACCELERATION 32.00000 AND TIME 5.000000 SECONDS
AN OBJECT FALLS 400.0000 FEET.
#
```

Labeling results is a very good programming practice usually ignored by beginning programmers. It does require more time and effort, but this is more than offset by the amount of clarity added to the code and its output. For programs which could he stored for any amount of time or for lengthy programs, any kind of documentation is very helpful and this labeling with TYPE statements is a very good form of documentation.

If a $ appears in the list of things to print (not in quotes) FOCAL will print out all the variables in use and their corresponding values. The $ terminates the print list, that is, anything following it won't be printed.

A very powerful use of the TYPE statement comes from its ability to print the result of whole expressions. This means your computer can be used as a super calculator which understands variables. This capability is generally used in the immediate mode as shown below. The variables used are the same as those stored for use by the program.

```
*TYPE 5^2*16.,#
400.0000
*TYPE FSQT(2*D/32),#
5.000000
*TYPE "TIME TO FALL",D,"FEET IS",FSQT(D/16),"SECONDS.",#
TIME TO FALL 400.0000 FEET IS 5.000000 SECONDS.
*SET D=144
*TYPE"TIME TO FALL",D,"FEET IS",FSQT(D/16),"SECONDS.",#
TIME TO FALL 144.0000 FEET IS 3.000000 SECONDS.
```

4.2 The Ask Command

Input to a FOCAL program is handled by the ASK command. It is used in stored programs to define or redefine the values of program variables. The command can contain a text string and a list of variables. No expressions may appear in the ASK command although they are allowable responses to the command. When executing an ASK, FOCAL issues a question mark to request a value. The value, or expression, for which FOCAL can compute a value, must be followed by a carnage return. FOCAL then issues a question mark for the next variable to be defined, and so on. Text is printed as encountered in the command. In use this looks like:

```
*70.6   ASK "DEFINE STARSHIPS X,Y,Z COORDINATES",X,Y,Z,#
*70.61  ASK "DEFINE X,Y,Z FOR KLINGON SHIP",XK,YK,ZK,#
*70.65  SET XD=XK-X; SET YD=YK-Y; SET ZD=ZK-Z
*70.66  SET DIST=FSQT((XD*XD)+(YD*YD)+(ZD*ZD))
*70.69  TYPE "DISTANCE TO ENEMY IS",DIST,"LIGHT YEARS",
*GO
DEFINE STARSHIPS X,Y,Z COORDINATES?4?5?6
DEFINE X,Y,Z FOR KLINGON SHIP?9?4?1
DISTANCE TO ENEMY SHIP IS 7.141429 LIGHT YEARS
```

The ASK command also has provisions to allow a defined value to remain unchanged. The user can type the ESCAPE key in response to the question mark, and the corresponding variable will be unchanged. Typing a carriage return will result in the variable being set to 0. Typing an expression (which may even contain variable and functions) will cause FOCAL to evaluate the expression and assign the resulting value to the variable in question. ASK will continue to issue question marks for the remaining variable in its list. Although the user follows each entry with a carriage return, a line is advanced during an ASK command only when a colon or exclamation point appears in the statement.

5. BRANCH COMMANDS

The computer's ability to alter the sequence of commands it will execute is known as branching. This very powerful ability is represented in FOCAL by three commands: GOTO, IF and JUMP. These can alter the program flow rather than executing statements in their numeric order. The computer can send control to a program line number specified in the command. These commands differ in that GOTO always transfers control to the single statement number given to it, while JUMP and IF transfer to one of a number of possible statements based on a test.

5.1 The GOTO Command

In the following example, the GOTO statement sends control back to a statement that counts the number of times it has been executed. Readers new to programming are strongly advised to follow the example and the results closely.

```
*1.1  SET N=0
*1.3  SET N=N+1
*1.6  TYPE "LOOP NUMBER =",N,#
*1.7  GOTO 1.3
*GO
LOOP NUMBER = 1.000000
LOOP NUMBER = 2.000000
.
.
```

Unfortunately, this program never ends, and the programmer will never see the READY asterisk from FOCAL. Program segments which repeat are called "loops." The program shown above is an example of an "infinite loop". To escape such a loop, type MODE and the program will stop.

GO (GOTO) starts a program at the line number specified or at the lowest line number in the program, if no line number follows. The GOTO statement can also be used in the immediate mode to transfer control to the program. In the next example, the instruction GOTO 1.8 passes control to statement 1.8, and starts the loop with the loop counter already equalling 12.

```
*1.1       SET N=0
*1.3       SET N=N+1
*1.8       TYPE "LOOP NUMBER =",N,#
*1.9       GOTO 1.3
*SET N=12; GOTO 1.8
LOOP NUMBER = 12.00000
LOOP NUMBER = 13.00000
LOOP NUMBER = 14.00000
LOOP NUMBER = 15.00000
```

5.2 The IF Statement

In the above example, the program will again cycle indefinitely, since it has no condition for ending itself. For this reason, FOCAL includes the IF command which transfers control CONDITIONALLY. The basic form of an IF statement is;

```
*<line number> IF (<expression>) L1,L2,L3
```

where L1,L2, and L3 represent statement numbers, and the expression, always enclosed in parentheses, stands for a single variable or arithmetic formula containing variables.

When FOCAL encounters an IF statement, and the value in parentheses is negative, the control is transferred to the first statement number in the list. If the value is zero, control goes to the second, and if it's greater than zero, it transfers to the third. FOCAL recognizes abbreviated forms of the `IF' statement containing one or two statement numbers rather than three. Should the IF statement only contain 2 statement numbers in its transfer list, control will be given to the statement following the IF statement when the value is greater than zero. Similarly, when an IF statement contains only one statement, a value greater than or equal to zero will have control transferred to the next sequential command. These different styles of IF statements are shown in the following examples.

```
*1.2  IF (A-B) 1.5,1.4,1.3
*1.3  TYPE "A IS GREATER THAN B"; QUIT
*1.4  TYPE "A IS EQUAL TO B"; QUIT
*1.5  TYPE "A IS LESS THAN B"; QUIT
.
.
.
*22.1  IF (MONEY) 22.28, 22.28; TYPE "YOU STILL HAVE
       FUNDS",#
*22.3  DO 24.0; GOTO 15.4
.
.
.
*40.6  IF (II) 40.8; DO 70.0; GOTO 40.6
*40.8  TYPE "II IS FINALLY NEGATIVE. GOODBYE",#
*40.9  QUIT
```

Note in the above that a space always separates the IF and the open parenthesis mark. These examples are shown only to exercise the various aspects of the IF statement. They are not meant to be working parts of a single program.

5.3 The JUMP Statement

The last of the branch instructions is the JUMP command. This statement is frequently used when a program needs to transfer to one of more than three locations. The general form of this multibranch instruction is

```
*<line number> JUMP (<expression>), L1,L2,L3,L4, . . .,Ln
```

L1 through Ln are the statement numbers much like those in the IF command definition, but there may be as many numbers given as can fit in the command line. EXPRESSION, as before, is any single variable or arithmetic combination of variables. If the value of the expression equals 0, control transfers to the first statement number given. When the value equals 1, the second statement number is chosen, and so on. Should the value contain a fractional part, like 2.37 or 2.98, only the integer part is considered. The values 2.37 and 2.98 would both transfer control to the third statement listed. The following shows this command being used to select a part of a program given some input from the user.

```
*1.2 ASK "1) RIGHT 2) LEFT 3) UP 4) DOWN 5) NO CHANGE",N
*1.4 JUMP (N-1), 10.1,15.34,12.3,65.98,2.02
```

If the computed value of the bracketed variable or expression is less than zero, control goes to the first statement listed and if it's too large for the list, FOCAL sends control to the last statement listed. If the user had responded 5 or larger to the ASK statement above, control would have gone to the statement numbered 2.02.

6. SUBROUTINES

6.1 The DO And RETURN Commands

A subroutine, sometimes called a "routine," is a special sequence of statements with the same group numbers. A group number, as mentioned, is the integer part of a statement's line number and the fractional part is the step number. A subroutine, for instance, could be the sequence of statements between line numbers 52.01 and 52.99. This sequence is "special" because any part of the program can send control to this block of statements and receive it again when the block, the subroutine, has finished. The "sending of control" is known as a subroutine "call" and the process used for a subroutine to return this control to the code which called it is a "return" from subroutine. In FOCAL the subroutine is thrown into execution by the DO statement, and the return from the routine by a RETURN statement.

The DO statement must specify a line number containing the group number for the subroutine to be called and a step number of zero; thus "DO 52.0" is acceptable, whereas "'DO 52" is not. The RETURN statement requires no arguments; it simply returns control to the statement following the DO statement which called it. Note, in the example below, that 1) the same subroutine can be called from many places, and 2) a subroutine may call another subroutine, which may call yet another, which. . .etc.

```
*10.1  DO 20.0; DO 18.0
*10.2  C IF J NOT RIGHT, CALL 20.0
       AGAIN
*10.3  IF (-J) 10.4; K=20*T; DO 20.0
       .
       .
*20.1  F I=1,10; DO 15.0;
*20.6  R
       .
       .
*18.1  DO 20.0; T K,#
*18.2  DO 16.0
*18.3  R ; C THESE EXAMPLES WERE NOT
       TAKEN FROM
*18.4  C A WORKING PROGRAM
```

The next example further exercises the flexibility of the subroutine calling structure. In this example, a subroutine calls ITSELF until a certain condition is satisfied, and then it begins a series of RETURNS while calculating a factorial for a number. A return to itself is made for each call it made to itself; the last return sends control back to the DO statement which originally called this factorial subroutine. This sort of subroutine calling is known as RECURSION. Computer theory buffs should note that the initial variable list is used for all levels of the calling sequence; FOCAL does not dynamically allocate new memory for copies of the variables.

```
*1.1  A N; C ASK FOR THE NUMBER TO USE
*1.2  S NF=1; DO 2.0; T N,"FACTORIAL IS",NF,#;Q

*2.1  C SEE IF N IS GREATER THAN 1. IF SO SUBTRACT
*2.2  C ONE AND CALL THIS ROUTINE AGAIN UNTIL IT IS 1
*2.3  C THIS ROUTINE RETURNS AS MANY TIMES AS IT WAS CALLED
*2.4  C AND THIS CONTROLS THE FACTORIAL CALCULATION
*2.5  IF(N-1) 2.8,2.8 ; S N=N-1; DO 2.0
*2.6  C RETURNS ENTER HERE
*2.7  S N=N+1 ; S NF=NF*N
*2.8  R ; C RETURN FROM LAST CALL
```

If FOCAL runs out of step numbers for a subroutine, thus threatening to continue into the next group of line numbers, it issues the RETURN. This makes it perfectly valid to omit the return statement from a subroutine. This optimizes memory requirements at the expense of a program that becomes more difficult to read.

7. LOOPS

7.1 The FOR Command

Program loops can be constructed in FOCAL with the FOR command. This command executes the remaining statements on its SAME LINE a specified number of times. The number of loops depends upon the numbers given to the FOR command. In its full form, FOR uses 4 values; an index variable, a start value, increment and stop value for the index. These, in order, look like:

```
*12.1 FOR I=1,3,200 ; J=I/2 ; TYPE J;
```

In the above, the values 1, 3, and 200 could have been variable names, and the second and third statements on that line could have been any legal statements in the language. They must be followed by a blank and the ENTIRE LINE must be followed by a semicolon. In the following example,

```
*12.4 FOR II=VN,NN,Q ; TYPE II, FSQT(II),#;
```

the variable II is initially given the value VN. On successive loops its value increases by the amount NN, and when this value exceeds Q control is passed to the next LINE NUMBER. If only 2 values follow the equal sign, it is assumed that the increment has been omitted, leaving only the start and end values for the next II. In this case, the increment is set to 1.0.

Since the FOR command executes only the statements on its same line, it is convenient to use it in conjunction with the DO command. The polynomial graphing given on this page shows this in use.

```
*20.2   S LX=40; S LY=70; S YN=0 S YX=100
*20.3   S XN=0; S XX=100
*20.5   S SX=(XX-XN)/LX ; S SY=(YX-YN)/LY
*20.6   A "DEFINE A,B,C FOR AX^2+BX+C" ,A,B,C
*20.7   T #,"GRAPH Y=AX^2+BX+C, X IS DOWN,"
*20.71  T "Y ACROSS"
*20.9   F X=XN,SX,XX; DO 60.0;
*20.95  Q

*60.1   S  Y=X^2*A+(B*X)+C
*60.3   I (Y-YX) 60.5; S Y=YX; G 60.8
*60.5   I (YN-Y) 60.8; S Y=YN; G 60.8
*60.8   T "I"
*60.83  F J=YN,SY,Y; T "*";
*60.9   T # ;C RETURN
*GO
DEFINE A,B,C FOR AX^2+BX+C?1/80?-1?40
(PRINTS OUT GRAPH HERE)
```

The reader should study the example shown here, and compare this to the fully commented version. It should also be very instructive to run this program as shown, then modify both the program and the data as desired.

The term "loop" refers to any sequence of statements which can be executed repeatedly; the "FOR" statement is only one way of forming a loop. A common way of setting up a loop uses "SET," "IF" and "GOTO" statements in such a way that a counter (or "loop index"), an increment value and a limit are manipulated by the program directly. The following shows the code necessary for such a loop:

```
*23.25  C INDEX IN IS SET 1 INCREMENT LOWER THAN 1ST VALUE
*23.3   SET IN=0
*24.1   C INCREMENT LOOP INDEX. 24.11 IS LOOP START
*24.4   SET IN=IN+IC
*25.1   C TRANSFER OUT OF LOOP WHEN INDEX EXCEEDS LIMIT
*25.2   IF (IN-LIMIT) 27.6
*25.7   C HERE STARTS THE CODE FOR THE BODY OF THE LOOP.
        .
        .
        .
        .
*27.5   GOTO 24.4 ; C FORCE NEXT LOOP
*27.6   C THIS STATEMENT IS OUT OF THE LOOP
*27.7   C REST OF PROGRAM CONTINUES FROM HERE
        .
        .
```

Another example is given in the next section on subscripted variables.

7.2 Subscripted Variables

The variables in a program generally represent the physical entities of the problem under study. The programmer can think of his variable, "T," as containing the current time in seconds or another variable, "SP" as representing a vehicle's speed. This association between variables and their physical meaning is fundamental to any type of computer programming. Quite often, however, several values must be simultaneously associated with a single concept, and thus, the programmer would like to have a single variable name represent these many values. A chess board is a good example of this, since the programmer would like 64 values held for the single board. While it would be possible to assign each of the squares a separate name, FOCAL's subscripted variable feature allows all the squares to be referenced with the same name. A variable which has many values is called an ARRAY, and its separate values may he selected by means of a SUBSCRIPT or INDEX. .A subscript is an integer tag identifying a particular value within an array. It is enclosed in parentheses immediately following the array name. "BD(1)", for example, might be the first square of the board, while "BD(64)" might be the last. Little advantage would be realized were it not for the fact that subscripts themselves can be variable names or even expressions. In other words, any expression can be put into the parentheses following the array name; FOCAL merely calculates the value of the expression, drops any fractional part, and uses the resultant integer to select a single value from the array.

As an illustration, the following sequence of statements, written as a subroutine, counts the number of pieces a chess bishop can threaten from his square. Some initial definitions at the beginning of the program are shown as is the routine itself; the actual calls) from the main body of the program have been omitted.

```
*1.11   C 8 POSSIBLE DIRECTIONS OF MOVEMENT STORED IN DX,DY
        ARRAYS
*1.1    C DEFINE ARRAYS TO BE USED BY LATER SUBROUTINES
*1.12   S DX(1)=1 ; S DX(2)=1 ; S DX(3)=-1 ; S DX(4)=-1
*1.12   S DX(5)=1 ; S DX(6)=-1 ; S DX(7)=0 ; S DX (8)=0
*1.13   S DY(1)=1 ; S DY(2)=-1 ; S DY(3)=1 ; S DY(4)=-1
*1.15   S DY(5)=0 ; S DY(6)=0 ; S DY(7)=1 ; S DY(8)=-1
*1.16   C FIRST 4 USED BY BISHOP. LAST BY ROOKS. ALL BY QUEEN/
        KING
*       .
*       .
*       .
*       .
*       .
*       .
*       .
*70.02  C THIS SUBROUTINE COUNTS THE NUMBER OF OPPOSING PLAYERS
*70.04  C THREATENED BY A KNIGHT AT ROW "NR", COLUMN "NC"
*70.06  C ASSUMES THAT OPPOSING PIECES ARE CODED AS NUMBERS WITH
*70.08  C OPPOSITE SIGNS AND THAT EMPTY SQUARES CONTAIN ZEROS.
*70.10  C "BD" HAS THE ENTIRE BOARD OF 64 SQUARES. "ZAP" COUNTS
*70.12  C THREATS FOR THE CALLING ROUTINE AND "LP" WILL
*70.14  C BE THE LOOP DIRECTION COUNTER INTERNAL TO THIS ROUTINE
*70.16  C START BY ZEROING COUNT AND STARTING DIRECTION INDEX
*70.17  S ZAP=0 ; S LP=0
*70.18  C PUT BISHIP'S VALUE INTO BT TO COMPARE LATER. FROM NR,
        NC
*70.20  S BT=BD(NC-1*B+NR)
*70.22  C START LOOP - RETURN WHEN LP PAST 4
*70.24  SL P=LP+1 ; I (LP-5) 70.26 ; RETURN
*70.26  C SET INITIAL POSITION OF MOVING SQUARE
*70.28  S NX=NR; S NY=NC
*70.30  C LOOP THROUGH NEXT SQUARES ON CHOSEN DIRECTION
*70.32  S NX=NX+DX(LP) ; S NY+DY(LP)
*70.34  C SEE IF YOU'RE STILL ON THE BOARD. 1 < or = to NX,
        NY < or = to 8
*70.36  I (NX) 70.24, 70.24 ; I (9-NX) 70.24,70.24
*70.38  I (NY) 70.24, 70.24 ; I (9-NY) 70.24,70.24
*70.40  C CALCULATE POSITION (INDEX) IN BOARD FOR THIS SQUARE
        (NX,NY)
*70.42  S SQ=BD(NY-1*8+NX)
*70.43  C MULTIPLY BY BISHOP -S VALUE TO CHECK SIGNS
*70.44  S PR=SQ*BT ; I (PR) 70.46,70.32,70.24
*70.45  C FOUND OPPONENT - COUNT IT AS THREATENED. LOOP AGAIN
*70.46  S ZAP=ZAP+1 ; DO 71.0
*70.50  G 70.24; C END OF SUBROUTINE
*71.1   C PRINT OUT THREATS - MONITOR PROGRAM PROGRESS
*71.2   T "PIECE AT ROW",NX,"COLUMN", NY," IS THREATENED BY",#
*71.25  T "PIECE AT", NR,NC,#
*71.3   R
```

FOCAL allows subscripts to have any values from -2047 TO + 2047.

7.3 The COMMENT Statement

FOCAL allows comments to be inserted into a program with the C command. This command requires a line number as any other command in a stored program, but when FOCAL encounters this statement, it simply

skips to the next command. This statement begins with a line number, the letter C, and at least one blank following the C. The rest of the line, up to the semicolon, is ignored by FOCAL. Since these statements have line numbers, branches can be made to them. In this case, comments can be thought of as being "continue" statements (as in FORTRAN).

```
*19.1   C -- POLYNOMIAL GRAPHING PROGRAM --
*19.2   C -- DOCUMENTED VERSION --

*20.1   C PREPARE SCALING VALUES THAT RELATE THE SIZE
*20.12  C OF THE PHYSICAL GRAPH TO THE FUNCTION VALUES
*20.14  C TO BE PLOTTED.
*20.2   S LX=40 ; S LY=70; S YN=0; S YX=100
*20.22  C XN,XX ARE THE MINIMUM AND MAXIMUM X VALUES
*20.24  C YN,YX ARE THE MINIMUM AND MAXIMUM Y VALUES
*20.3   S XN=0; S XXS=100
*20.42  C COMPUTE LENGTH BETWEEN SPOTS ON PLOT BY
*20.44  C COMPARING BOUNDS TO LENGTH AND WIDTH
*20.5   S SX=(XX-XN)/LX ; S SY=(YX-YN)/LY
*20.52  C INPUT PARAMETERS FOR THE POLYNOMIAL
*20.6   A "DEFINE A,B,C FOR AX^2+BX+C ",A,B,C
*20.7   T #, "GRAPH Y=AX^2+BX+C X DOWN, Y ACROSS",#
*20.8   C EACH LOOP OF 20.9 DOES 1 LINE OF PLOT
*20.9   F X=XN,SX,XX; DO 60.0;
*20.95  Q

*60.05  C THIS ROUTINE COMPUTES POLYNOMIAL FOR WHATEVER
*60.07  C VALUE OF X IS PASSED. ITS THEN PLOTTED
*60.09  C (IF POSSIBLE) WITHIN THE DEFINED BOUNDS.
*60.1   S  Y=X^2*A+(B*X)+C
*60.24  C IF Y VALUE T00 LARGE OR T00 SMALL, PLOT ON EDGES
*60.3   I (Y-YX) 60.5; S Y=YX; G 60.8
*60.5   I (YN-Y) 60.8; S Y=N; G 60.8
*60.8   T "I"
*60.81  C PRINT ASTERISKS UNTIL INDEX AS LARGE AS Y
*60.83  F J=YN,SY,Y; T "*";
*60.88  C FINISH THIS LINE WITH CR AND RETURN FOR NEW X
```

8. SUPERVISORY FUNCTIONS

8.1 The WRITE Command

For editing purposes, FOCAL provides the ability to print all or parts of the program text with the WRITE command. It can be used to print single lines or subroutines. WRITE 2.2 will print just the line which is numbered 2.2 WRITE 2.0 will print only the lines between 2.01 and 2.99, and the command WRITE ALL will print the entire program ordered by increasing line number.

8.2 The ERASE Command

The ERASE command is used to delete lines or groups of lines from a program. To erase a single line from the text, the user only has to type ERASE followed by the line number as in ERASE 22.34. To erase an entire group of lines such as a subroutine, the user can type ERASE followed by the group number. To delete a subroutine with the group number 95, the user should type ERASE 95.0.

ERASE can also he used to clear an entire program, its variables and their values. This is only done when the user wants to write a new program. The ERASE ALL releases all the memory assigned to the last program so that it can be used by the new one. It is naturally a good habit to save any lengthy programs on cassette tape before erasing them.

8.3 The MODIFY Command

The MODIFY command is used in immediate mode to edit portions of lines in a FOCAL program. It accepts a line number designating the statement to be edited; this line number must be followed by a carriage return. The actual editing is performed on the line following the MODIFY command. The user must guide the MODIFY editor with certain non-printing commands. After the carriage return, MODIFY waits for the user to type a single character from the keyboard; this character will be used as the "search character." MODIFY will print the line in question up to and including this search character, or the user can direct modify to perform one of several other tasks described below. Each of these tasks is selected with a special character which is typed after the search character. If MODIFY does not recognize this character as being a member of its special list, it assumes that a text insertion is being made.

1. "CNTRL/G". This does nothing to the text already defined, but prepares the MODIFY function for a new search character. This new search character must follow immediately; neither will be printed. Any searches through the rest of the current line will use this new character.

2. "CNTRL/L" or "FORM FEED". This command restarts the search procedure in MODIFY which will begin typing the rest of the line until the current search character is found. As in the first search sequence undertaken by MODIFY, the search chyracter will be typed, and MODIFY will then wait for more commands.

3. "DELETE". This deletes the last character in the line. Successive DELETE's delete characters in order from right to left towards the line number.

4. "CNTRL-X". This deletes everything in the line up to and including the last character printed. The rest of the line, as yet imprinted, remains intact. It will be shifted over, however, so that it follows the line number.

5. "RETURN". This deletes anything to the right of the last character last printed.

6. "LINE FEED". This instruction tells MODIFY to save the line as presently defined.

8.4 FOCAL Library Commands

Processor Technology Cassette Focal has an interface with SOLOS/CUTER to save and recall programs and data from cassette files. This interface is through a series of LIBRARY statements:

```
LIB SAVE <file name>          Save current program
LIB LOAD <file name>          Get program from tape
LIB OPEN <#>,<file name>      Open a data file
LIB TYPE <#>,<var list>       "Type" data to file
LIB ASK <#>,<var list>        "Ask" for data frost file
LIB REWIND <#>                Rewind a data file
LIB CLOSE <#>                 Close data file
LIB QUIT                      Return to SOLOS/CUTER
```

```
<file name>     is any valid SOLOS/CUTER file name of 1 to 5
                characters, with an optional unit specifies.
<#>             is a digit between 0-9.
<var list>      Valid FOCAL variable list which may contain
                constants, variables and expressions for TYPE,
                just variables for ASK.  Must not contain
                quoted text, "$","#","?" or "!".
```

All LIBRARY commands may appear in a program or may be used in the immediate mode. Except for L TYPE and L ASK, none of the library commands should precede other commands on the same line, since the rest of the line will be ignored.

The L LOAD command may be used within a program to chain to another program. This will be discussed more fully under LIB LOAD below.

Naturally, all library commands can be abbreviated. For example LIB CLOSE 5 can be written as L C 5.

Due to the fact that only two cassette drives can be used at one time, only two files can be open at one time, one on each unit. Whenever <filename> is specified, it may be followed by an optional "/"" and a unit number, indicating which tape drive to use. If no unit is specified, then tape unit 1 is assumed. For example:

```
*LIB OPEN 5, TUNA/2    Opens the file "TUNA" on
                       cassette drive #2;
*LIB LOAD FISH         Loads the file "FISH" from
                       cassette drive #1.
```

LIB SAVE < file name > will save an exact copy of the internal form of the current program in memory onto file < file name >.

LIB LOAD < file name > will load in the data in the file over any program which is currently in memory. If the LIB LOAD command was issued from a program, the newly read in program will he executed starting at its lowest line number. This allows chaining of FOCAL programs. To pass data from the first program to the second, it should be written onto a data file by the first program (using LIB TYPE-see below) and read back from the same file number by the second program. (using LIB ASK). It is not necessary to re-OPEN the data files during a chain as long as the same file numbers are to he used in both programs. When chaining, variable's values are lost, unless they are saved on a data file.

LIB OPEN < # >,< file name > is used to set up a data file for subsequent LIB TYPE or LIB ASK statements. The opened file is assigned the number <#>, and this number must be used to refer to the file in any subsequent data file operations such as TYPE. REWIND, etc.

LIB TYPE < # >, <var list > is similar to the regular TYPE command except that its output goes to file < # > where the number < # > has been assigned to a file using the LIB OPEN command. In addition, only numerical data can be written to a FOCAL data file-no ,quoted text, #, !, $ or ? . The data is written to the file in binary form-4 bytes per number. This data is written starting at the current cursor position for the file. For example, the sequence

```
:LIB OPEN 5,POTTS
:LIB TYPE, 5,A,B^10,9*1024
:LIB CLOSE 5
```

results in the values of the three expressions A, B ^ 10 and 9*1024 being written to the beginning of the file POTTS.

LIB ASK < # > , < file name > is used to read values from a file which LIB TYPE has written on. The number < # > must have previously been assigned to < file name > in an OPEN statement.

LIB REWIND < # > sets the cursor for file < # > (which has been assigned in an OPEN) to the beginning of that file. This is where the cursor is when the file is first opened. LIB REWIND can be useful when chaining between programs, since the second program must start at the beginning to read the data from a file.

LIB CLOSE < # > removes the association between the given number and the file to which it was assigned in a previous LIB OPEN, writes out the file's buffer if necessary, and frees the buffer space for later use. After executing a CLOSE, the specified number may be re-used in a LIB OPEN statement. Re-using a number without first closing it will result in an error.

LIB QUIT leaves FOCAL and returns to SOLOS/CUTER. Before leaving, QUIT closes all files.

8.5 The TRACE Feature

The TRACE feature is provided to help debug stored programs. It can be activated from almost anywhere in a program and can be deactivated as easily. While operating, trace types out each line it sees being executed by FOCAL and reports on any variable values that are changed during each line of the program. A question mark is the symbol used to both activate the trace and deactivate it. Any question mark encountered outside a comment statement and outside the text parts of the "ASK" and "TYPE" commands will change the trace mode. If the question mark is encountered while trace is active, the trace will be deactivated. If seen while trace is "OFF" the trace mode will be turned "ON".

9. RUNNING FOCAL

9.1 Hardware Requirements

In order to run FOCAL, an 8080-based computer must be equipped with at least 8K of resident random access memory, and the SOLOS or CUTER monitor program. The memory should be addressed starting at zero continuously to 8K. FOCAL is set up to use 16K of memory, allowing more than 9K for programs and data. More than 16K can be used by changing memory locations 0006 and 0007 to contain the highest address of continuous available memory.

For example, if 20K (5000 Hex) is available, enter the following into memory:

```
ADDRESS   0006      OOH
          0007      50H
```

These values may be changed any time after loading FOCAL. The number of memory locations used by any stored program can be calculated from the rule: SIZE = 8S+ C+ 4L where S is the number of variables, C is the number of characters in the stored program text, and L is the number of lines in the program. FOCAL can be SAVEd with the new value if desired.

FOCAL is saved on cassette in CUTS format, which may be read by any Sol computer, or any other computer with a CUTS module and the CUTER monitor program. To load FOCAL, use the command: XEQ FOCAL <CR>, in SOLOS/CUTER command mode.

When FOCAL is first loaded into memory, it checks itself in memory by the use of checksums. If the tape has been damaged, a bad memory location is encountered, or other hardware problems have caused incorrect code to appear in memory, the message "CHECKSUM TEST FAILED" will appear. The error may not be serious, and FOCAL may be used, but it is best to try reloading the tape, correcting hardware problems, if present.

9.2 About Cassette Recorders

Successful and reliable results with cassette recorders require a good deal of care. Use the following procedures:

1) Use only a recorder recommended for digital usage. For use with the Processor Technology Sol or CUTS, the Panasonic RQ-413 AS or Realistic CTR-21 is recommended.

2) Keep the recorder at least a foot away from equipment containing power transformers or other equipment which might generate magnetic field picked up by the recorder as hum.

3) Keep the tape heads cleaned and demagnetized in accordance with the manufacturer's instructions.

4) Use high quality brand-name tape preferable a low noise, high output type. Poor tape can give poor results, and rapidly wear down a recorder's tape heads. Bulk erase tapes before using.

5) Keep the cassettes in their protective plastic covers, in a cool place, when not in use. Cassettes are vulnerable to dirt, high temperature, liquids, and physical abuse.

6) Experimentally determine the most reliable setting for volume and tone controls, and use these settings only.

7) On some cassette recorders, the microphone can be live while recording through the AUX input. Deactivate the mike in accordance with the manufacturer's instructions. In some cases this can be done by inserting a dummy plug into the microphone jack.

8) If you record more than one file on a side, SAVE an empty file named "END" for example, after the last file of interest. If you read this file header, you will know not to search beyond it for files you are seeking.

9) Do not record on the first or last 30 seconds of tape on a side. The tape at the ends gets the most physical abuse.

10) Most cassette recorders have a feature that allows you to protect a cassette from accidental erasure. On the edge of the cassette opposite the exposed tape are two small cavities covered by plastic tabs, one at each end of

the cassette. If one of the tabs is broken out, then one side of the cassette is protected. An interlock in the recorder will not allow you to depress the record button. A piece of tape over the cavity will remove this protection.

11) Use the tape counter to keep track of the position of files on the cassette. Always rewind the cassette and set the counter to zero when first putting a cassette into the recorder. Time the first 30 seconds and note the reading of the counter. Always begin recording after this count on all cassettes. Record the beginning and ending count of each file for later reference. before recording a new file after other files, advance a few counts beyond the end of the last file to insure that it will not be written over.

12) The SOLOS/CUTER command CATalog can be used to generate a list of all files on a cassette. Exit FOCAL using L Q, type CAT <CR>, rewind to the beginning of tape, and press PLAY on the recorder. As the header of each file is read, information will be displayed on the screen. If you have recorded the empty file called END, as suggested, you will know when to search no further. If you write down the the catalog information along with the tape counter readings and a brief description of the file, you will be able to locate any file quickly. After completing the catalog, you may re-enter FOCAL by typing EX 0 <CR>.

13) Before beginning work after any modification to the system, test by SAVEing and GETting a short test program. This could prevent the loss of much work.

9.3 Miscellaneous Notes

Should you accidentally leave FOCAL before saving your program, you can restart it by typing EXEC 0 to SOLOS or CUTER, without losing the program.

FOCAL does not understand lower case letters. All variables, commands, functions, etc. must be in UIPPER CASE. Lower case is okay within quotes or file names.

Remember that binary arithmetic is not exact-it is only very close. Therefore, you may expect very small (but nevertheless disconcerting) errors such as

```
:T 23+31
 54.00001
```

which are caused by FOCAL's attempts at rounding off numbers which it cannot store exactly. You can make these errors much less frequent by not printing as many digits to the right of the decimal point. (See TYPE)

9.4 Errors

The following is a list of error codes issued by FOCAL. The error number represents the address in the FOCAL program where the error was detected.

ERROR CODE	MEANING
00.00	Mode-select restart.
00.0D	BUFFER overflow, line.
02.96	Bad line number.
02.BB	Bad line number.
03.22	Bad character.
03.27	Bad character.
04.43	Stack overflow - expression too complex.
05.48	Bad line number.
05.BE	No such group.
05.FD	DO reference;, missing line.
06.1D	GOTO references missing line.
06.60	Illegal command.
06.08	Missing left paren in JUMP.
06.F0	Missing left paren in IF.
07.39	Left of = bad in FOR or SET
07.59	Excess right paren in FOR or SET.
07.70	Bad expression in FOR.
09.85	No such line number.
0A.C5	Missing terminator.
0A.E5	Missing operator before left paren.
0B.39	Arithmetic overflow.
0B.D7	Missing left paren in function reference
0B.EF	Missing left paren.
0C.18	Mismatched right paren.
0C.2D	ERASE F ? ? ?
0C.35	Bad argument in Erase.
10.79	Can't raise to a negative power.
10.D3	Bad library command.
10.EB	Missing comma in OPEN.
11.5E	Missing comma after file number.
12.4C	Bad file number.
12.51	Bad file number.
12.91	Tape read error or user hit mode select while reading.
12.94	File already open.
12.97	File not open.
12.9A	Unit already open.
12.9D	Unit not open.
12.A0	File name missing.
12.DB	End of File encountered.
12.DE	Error in read.
12.F6	Write operation not valid on this file.
13.44	Name too long.
13.65	Bad unit number.