Part 1

Understanding Floating-point Arithmetic

Ian Logan

The aim of this article is to give the reader some insight into the complex world of floating-point arithmetic. Since the 4K ROM provided only integer arithmetic, readers who possess only this ROM will be unable to try the programs. Nevertheless they will be able to follow the text.

In the Sinclair Manual, *ZX81 Basic Programming*, chapter 27, Steven Vickers shows that a floating-point number consists of a single exponent byte and 4 mantissa bytes, but he gives no further information. In order to understand this subject it is probably best to return to first principles—so with pencil and paper to hand proceed.

Decimal format

In the beginning there were only simple integers. But soon they begat decimal numbers, which have an integer part, a decimal-point and a decimal part. And in their turn decimal numbers begat E-format, which has a mantissa part, an 'E' and an exponent part.

For example, the number 'four' can be expressed as:

4	- its integer value
4.000	- its decimal value
40000E-4	- just one of many E-format choices

It can readily be seen that in the E-format we have the essential parts of floating-point notation for decimal numbers all given, but it is useful at this point to introduce two conventions that will help us in conversion from decimal-floating-point to binary-floating-point.

1) Always express the mantissa starting with the decimal-point.

2) Do not attribute a sign to the mantissa. Simply state whether the value is positive or negative. So instead of:

	Write:	
40000E-4	.4E1	& positive
0.00678	.678E-2	& positive
-223.9	.2239E3	& negative
-0.7	.7E0	& negative

These conventions can be considered to be 'normalizing' the floating-point decimal number.

With a decimal number in its 'normalized' form we can now state that the mantissa is the decimal part of the form and the exponent is the integer part after the 'E'. The exponent is a signed integer and the overall form is either positive or negative. Consider the examples in Figure 1. The reader is urged to try further examples. (Perhaps with a friend marking the results.)

Figure 1.

Decimal	Normalized	Exponent	Mantissa	+/-
4	.4E1	+1	4	+
40	.4E2	+2	4	+
.4	.4E0	+0	4	+
-40.0	.4E2	+2	4	-
-123.456	.123456E3	+3	123456	-

Binary Format

As the 8K ROM program deals with binary-floating-point numbers and not decimal-floating-point numbers, the reader will now have to convert the above conclusions so that they apply to binary-format numbers.

First, consider the state when all binary numbers represented integer values, that is:

Decimal	Binary
45	0010 1101
255	1111 1111

In this state all values are integers and positive only. Next consider fixed-point binary numbers in which there is a fixed binary-point separating the integer byte(s) from the fraction bytes(s). That is:

Decimal Form	Binary Form		
	integer	point	fraction
45	0010 1101	•	00000000
45.5	0010 1101	•	10000000
45.75	0010 1101	•	11000000
45.875	0010 1101	•	11100000

Note that in a fixed-point number the first bit after the binary-point represents the value .5 and the second bit .25 etc. (The values diminish by a factor of 2.)

However, it is also possible to consider the fraction part byte by byte, which in decimal can be illustrated as follows:

From above, .11100000 gives 224/256 as the fraction part and this does give 0.875.

Now at last the binary numbers can be

Ian Logan, 24 Nurses Lane, Skellingthorpe, Lincoln LN6 0TT.

'normalized.' All that needs to be done is for the whole number to be moved to the left, or the right, as needed so that the most significant bit comes to be the first bit of the fraction part. The exponent is then given as the number of moves made (+ right, - left) and the mantissa is the number of bits wanted from the fraction part.

Hence from above:

Decimal Form	Exponent	Mantissa
45	+6 (dec.)	10110100
45.875	+6 (dec.)	10110111 (a bit is lost)

Note that in the example with a mantissa being limited to just 8 bits that the values 45.75 and 45.875 cannot be distinguished. This shows why the 8K ROM uses not one but 4 bytes for the mantissa and even then it 'rounds' off values—sometimes inconveniently.

But how are negative numbers dealt with? Well, it is easy; there is just a statement made to say whether the value is positive or negative. For example:

Decimal Form	Exponent	Mantissa	+/-
255	+8 (dec.)	11111111	+
-255	+8 (dec.)	11111111	-

Now it is time to run Program 1. This *Floating-point Demonstration Program* asks the user to enter any decimal number that he may wish, including fraction parts and 'E's '. The program then returns the true exponent, e', and the four bytes of the mantissa. (e' is the exponent as developed above.) For example, entering the number 255 gives:

Decimal number 255
Its exponent 8
And mantissa 255 0 0 0 0
And it is POSITIVE

and entering -9.9E37 will give:

Decimal number -9.9E+37
Its exponent 127
And mantissa 148 245 105 108
And it is NEGATIVE

Note: The last value can be checked by trying the line:

```
PRINT (148/256+245/256**2+105/25
6**3+180/256**4)*2**126*2
```

which gives 9.9E+37 as expected. (Note that 2**126*2 is used to prevent overflow.) Program 1 works by reading the floating-point number that has been attributed to the variable A as that number occurs in the variable area of the RAM. Certain changes have to be made to these bytes in order to give the true exponent and the appropriate mantissa. Note for interest the differences between values of A that ought to be the same. See Figure 2. The later result is a 'rounding' error.

Whereas Program 1 borrows the result of the ROM program to get to its answer, Program 2, *A Floating-point Builder*, develops the result by successive multiplications, divisions, and subtractions. So try Program 2 in order to become more familiar with binary floating-point numbers.

Note: The lines 170, 180, and 210 are all attempts to get around the problem of 'rounding' errors. However, the serious reader might be interested in the fact that with an initial value of A such as 8 then the value of A at line 170 is:

.999999999 < A < 1

'PRINT A' gives 1, but 'IF A=1' is false.

The explanation lies in the fact that A has the binary value of:

EXP. 0 , Mantissa 127 255 255 253

instead of the expected

EXP. 1 , Mantissa 128 0 0 0

and therefore shows that the COMPARISON operation is of greater sensitivity than the PRINT operation.

Does this 'bug' account for some programming problems?

Figure 2.

	1/2	dec.	gives	Exp. 0	Mantissa 128 0 0 0
but	.5	dec.	gives	Exp. -1	Mantissa 255 255 255 255

Sinclair floating-point conventions

So far in this article I have described the use of the true exponent and the true mantissa, but in Sinclair machines the floating-point numbers follow two conventions which are:

1) The exponent byte always has 128 decimal, Hex.80, added to it, unless it is the exponent for the value zero when the exponent is always zero. Hence the 'augmented exponent,' e, is the 'true exponent,' e', +128. (See how in line 120 of Program 1 this is taken into account.)

2) The true numeric bit 7 of the first byte of the mantissa which is always set in a floating-point that has been 'normalized' is understood to be present and the bit replaced by a sign-bit. This bit is set for negative numbers and reset for positive numbers (and zero). (See how in line 140 of Program 1 this is taken into account.)

To make this clear consider the examples in Figure 3.

Figure 3.

Decimal Format	True Format		Sinclair Format	
	Exp.	Mant.	Exp.	Mant.
1.0	1	128 0 0 0	129	0 0 0 0
2.0	2	128 0 0 0	130	0 0 0 0
-2.0	2	128 0 0 0	130	128 0 0 0
3.0	2	192 0 0 0	130	64 0 0 0
-3.0	2	192 0 0 0	130	192 0 0 0
0.0	0	0 0 0 0	0	0 0 0 0

Conclusions

Floating-point notation is logical, tedious perhaps, but very useful.

By way of lighter relief this month's game is an example of Basic programming that shows how bytes can be saved in 8K ROM programs—who said the 8K ROM wastes bytes?

The idea of the game is simply to find a number that results in the pattern filling the whole board. My best score so far is about 100.

Remember that RND generates a given series of numbers, depending on the SEED for its starting point, but additional dummy calls to RND will create new series. E.g.,

```
145 POKE 0,RND
```

would be economic for a simple arithmetic series—alternate calls to RND are used by the 'pattern.'

Part 2 of "Understanding Floating-point Arithmetic" will discuss the third language of the 8K ROM—the Calculator Language.

Bibliography

Sinclair ZX81 ROM Disassembly, Part A: 0000 H-00F54 H, by Dr. Ian Logan. Melbourne House outlets—£7. (Deals with the 'operating system' part of the 8K ROM program).

Sinclair ZX81 ROM Disassembly, Part B: 0F55 H-1DFE H, by Dr. Ian Logan and Dr. Frank O'Hara. Melbourne House outlets—£8. (Deals with 'expression evaluation' and the 'calculator routines' in full detail). ■

8K ROM
1K RAM

Linear Regression

Jon T. Passler

The "Linear Regression" Program computes the linear relationship between two sets of variables, expressed as the linear regression equation, and calculates the coefficient of determination, an indicator of the strength of the relationship. Given a set of two variables labelled X and Y, the program will yield an equation describing Y as a function of X.

These variables can be taken from any situation in which a logical relationship is expected, such as rainfall and crop yield, the prime interest rate and auto sales, or time and any variable which changes (generally in one direction) over a period of time. For a time series, X can be expressed in periods, starting with period 1.

The coefficient of determination, R2 or R squared, is a measure of how much of the variability in Y is "explained" by, or related to, the variability in X. R2 varies between 0 and 1, and R2 multiplied by 100 gives a percent indication of the validity of, or accuracy in, expressing Y as a function of X.

For a quick example, let Y = 1 + 2 * X. If X = 1, 2, and 3, then Y would be 3, 5, and 7. Run the program, enter a 3 in response to the number of entries, then enter X's and Y's pairwise, or, to mix things up a bit, enter 2, 5, 3, 7 and 1, 3. You should get the equation Y = 1 + 2 * X back, and an R2 of 1, or 100%, since the equation perfectly describes the relationship between each pair of entries.

Linear regression can be used to approximate the value of one variable (given the value of another), identify the trend in time series and forecast future values, or evaluate the influence of one variable on another (R2). ■

Jon T. Passler, 344 Cabot St., Beverly, MA 01915.

```
 10 REM LINEAR REG
 20 PRINT "N OF ITEMS?"
 30 INPUT N
 40 LET SX=0
 50 LET SY=0
 60 LET XX=0
 70 LET YY=0
 80 LET XY=0
 90 PRINT "INPUT X/S AND Y/S"
100 FOR A=1 TO N
110 IF A>20 THEN SCROLL
120 INPUT X
130 PRINT X;"  ";
140 INPUT Y
150 PRINT Y
160 LET SX=SX+X
170 LET SY=SY+Y
180 LET XX=XX+ABS X**2
190 LET YY=YY+ABS Y**2
200 LET XY=XY+X*Y
210 NEXT A
220 CLS
230 PRINT
240 PRINT "Y = ";(XX*SY-SX*XY)/
(N*XX-ABS SX**2);" + ";(N*XY-SX*
SY)/(N*XX-ABS SX**2);" * X"
250 PRINT
260 PRINT "R2 = ";ABS ((N*XY-SX
*SY)/((N*XX-ABS SX**2)**.5*(N*YY
-ABS SY**2)**.5))**2
```

Program 1: Floating-point Demonstration Program

```
 10 PRINT AT 17,0;"ENTER ANY NU
MBER"                                  Any decimal number.
 20 INPUT A
 30 CLS
 40 LET V=PEEK 16400+256*PEEK 1
6401                                   Get the present value of
                                       VARS.
 50 DIM B(5)                           For the 5 bytes.
 60 FOR C=1 TO 5                       Get each byte from the
 70 LET B(C)=PEEK (V+C)                variable area.
 80 NEXT C
 90 PRINT "DECIMAL NUMBER";TAB
17;A
100 PRINT
110 PRINT
120 PRINT "ITS EXPONUNT";TAB 17
;B(1)-126*(B(1)<>0)                     Form the true exponent.
130 PRINT
140 PRINT "AND MANTISSA";TAB 17
;(A<>0)*(B(2)+128*(B(2)<128));TA
B 21;B(3);TAB 25;B(4);TAB 29;B(5
)                                      Form the true mantissa.
150 PRINT
160 PRINT "AND IT IS";TAB 17;"P
OSITIVE" AND (A>=0);"NEGATIVE" A
ND (A<0)                               Give the sign.
170 RUN
```

Program 2: Floating-point Builder

```
 10 INPUT A                            Any decimal value.
 20 CLS
 30 LET B=SGN A                        Keep the sign.
 40 PRINT "DECIMAL NUMBER";TAB
17;A
 50 LET A=ABS A                        Ignore negative sign.
 60 PRINT
 70 PRINT
 80 LET E=0                            Set exponent to zero.
 90 PRINT "ITS EXPONENT";TAB 17
;E
100 IF A>=.5 AND A<=1 OR A=0 TH
EN GOTO 150                            Exit when "normalized."
110 LET E=E-(A<1)+(A>1)                Exponent changes by one.
120 LET A=A*(.5+1.5*(A<1))             A chanes by .5 or 2 fold.
130 PRINT AT 3,17;E                    Watch it changing in SLOW.
140 GOTO 100
150 PRINT
160 PRINT "AND MANTISSA";TAB 17
;
170 IF A>.999999999 THEN LET A=
1                                      See text.
180 LET F=.003906249997                A little under 1/256.
190 FOR G=1 TO 4                       Each mantissa byte.
200 LET H=INT (A/F)                    The decimal value.
210 IF H>255 THEN LET H=128            For a rounding error.
220 PRINT H;" "                        The byte and a "space."
230 LET A=A-INT (A/F)*F                Decrease A.
240 LET F=F/256                        Change for each byte.
250 NEXT G
260 PRINT
270 PRINT
280 PRINT "AND IT IS";TAB 17;"P
OSI" AND (B>=0);"NEGA" AND (B<0)
;"TIVE"                                Fetch the sign.
290 RUN
```

Program 3: Floating-point Number Game

```
 10 PRINT AT VAL "20",NOT PI;"N
EW NUMBER?"
 20 INPUT N
 30 RAND N
 40 CLS
 50 FOR A=NOT PI TO VAL "15"
 60 PRINT "■";"                    ■"
AND (NOT A OR A=VAL "15");TAB VA
L "15";"■"
 70 NEXT A
 80 LET A=VAL "7"
 90 LET B=A
100 LET C=NOT PI
110 LET D=VAL "30"
120 LET D=D-SGN PI
130 IF D=NOT PI THEN RUN
140 LET E=INT (RND*INT PI)-SGN
PI
150 LET F=INT (RND*INT PI)-SGN
PI
160 PRINT AT A+E,B+F;
170 IF PEEK (PEEK VAL "16398"+V
AL "256"*PEEK VAL "16399")<>NOT
PI THEN GOTO VAL "120"
180 PRINT "*"
190 LET C=C+SGN PI
200 PRINT AT VAL "18",NOT PI;"S
TARS = ";C
210 LET A=A+E
220 LET B=B+F
230 GOTO VAL "110"
```

8K ROM
1K RAM

Part 2

Understanding Floating-point Arithmetic

Dr. Ian Logan

In the last article I described how decimal numbers are converted to their floating-point form for the 8K ROM monitor program, but I did not discuss how floating-point numbers are then manipulated. In this article I shall describe the workings on the 'third language' of the 8K ROM—the CALCULATOR LANGUAGE—that is used to add, subtract, multiply, divide and generally manipulate floating-point numbers.

The CALCULATOR LANGUAGE

This language is a STACK OPERATING language and is used to put numbers on to the calculator stack of the ZX80/81, remove numbers from the calculator stack, perform operations on the numbers that are already on the stack and make safety copies of numbers by copying them to the calculator's memory area.

Figure 1.

Address	HEX. code	Mnemonic	Comment
OAEF	ED	RST 0028,FP-CALC.	i.e. CALL 0028
OAFO	01	DEFB +01	Operation '1'
OAF1	34	DEFB +34	Operation '52'

As an example of how the language is used, the first occurrence of the language is shown in full from the listing of the 8K program in Figure 1. This can be shown by entering the Basic line:

PRINT PEEK 2799,,PEEK 2800,,PEEK 2801

which gives:

239
1
52

This example shows that the first step required is to call the calculator subroutine. This is done using an indirect jump through the RESTART at 0028. The actual calculator subroutine occurs at 199C in the 'unimproved' ROM and at 199D in the 'improved' ROM.

The bytes of data that follow an RST 0028 instruction are the operation codes of the CALCULATOR LANGUAGE. In this example the first operation to be performed is operation '1' that asks the calculator to EXCHANGE the top two numbers on the calculator stack. The second operation is operation '52.' This is the END-CALC. operation which shows the calculator that there are no further manipulations to be performed.

In the 8K ROM program there are about 40 calls to the calculator subroutine and practically all of them are a good deal more complicated than this first example from the PRINT command routine.

Dr. Ian Logan, 24 Nurses Lane, Skellingthorpe, Lincoln LN6 OTT.

A Demonstration Program

The only way to understand well just how a STACK OPERATING language is used is to produce a simple demonstration program and then as experience is gained to make the program more ambitious.

Program 1—*A Basic Stack Calculator* is such a program written for a 1K ZX81. The first part of the program—lines 10-70—creates a small 'calculator stack' that appears on the screen as a stack of 5 locations each capable of holding a single decimal number. The very important 'last location' pointer is then shown. This pointer tells the user which location is currently the holder of the 'last value,' i.e., the location that holds the topmost number of the 'current stack.'

There are many points that can be made about stack systems but perhaps the most important point is that numbers are not removed from a stack but always copied from it and with each 'removal' the actions involved are a copying of the 'last value' and the decreasing of the STACK POINTER, in this case, the 'last location' pointer.

In lines 100-130 the program awaits the user's entry and then jumps accordingly. In this first program there are 5 operations that can be performed:

1) Input a number: The user is asked for a number and this number goes to the top of the stack.

2) Print a number: The 'last value' is displayed and the program pauses for a key to be pressed before proceeding. Printing the 'last value' does 'remove' it from the stack and hence loses the value. Any attempt to print a 'last value' from an empty stack leads to failure.

3) Adding two numbers: When there are at least two numbers on the stack, the last two numbers can be added together. This operation has the effect of losing the second operand.

4) Subtracting two numbers: The difference between the last two numbers becomes the 'last value.'

5) Multiplying two numbers: The product of the last two numbers becomes the 'last value.'

Program 1 shows that a stack of 5 locations and the handling of 5 operations can be performed on a 1K machine but the reader is urged to add extra operations if extra RAM is available, or to replace the existing 5 operations if not. Suitable operations to be added are EXCHANGE, DELETE, DUPLICATE, SQR, ** and SIN.

Some examples for a beginner to try are:

a) Print the sum of 2.55 and 3.66. This will involve selecting operation '1' and entering the value 2.55, selecting operation '1' and entering the value 3.66, selecting operation '3' to add the two numbers and finally to select operation '2' to print the result.

b) Print the product of 2.55 and 3.66. This will involve the sames step as before but operation '5' is used instead of operation '3'.

It is customary to describe the steps involved in what is termed 'Reverse Polish Notation' in which commas are used to separate the operations. In the present cases the results would be:

For
PRINT the result of 2.55 + 3.66'
write
STACK 2.55, STACK 3.66, ADD, PRINT'
or more simply
'2.55,3.66,+'.

For
'PRINT the result of 2.55*3.66'
write
'2.55,3.66,*'.

The reader is urged to try his own, more complicated, examples before proceeding to consider the operations of the 8K ROM's CALCULATOR LANGUAGE.

The OPERATIONS of the CALCULATOR LANGUAGE

So far I have described the CALCULATOR LANGUAGE as dealing only with floating-point numbers but this is not the whole answer as the language also manipulates character strings. It does this by considering, when required, the 'last value' as being a set of parameters that describes the string—the second and third bytes hold the pointer to the start of the string and the fourth and fifth bytes the counter for its length. (The first byte is redundant.)

The actual operations are: (operation codes in decimal)

0- Jump if 'last value' is true.
1- Exchange the top two values.
2- Delete the 'last value.'
3- Subtract the 'last value' from the value beneath it.
4- Multiply the top two values.
5- Divide the first value by the 'last value.'
6- Raise the first value to the power of the 'last value.'
7- Logically 'OR' the top two numbers.
8- Logically 'AND' the top two numbers.
9- Test the top two numbers for '<='
10- Test the top two numbers for '>='
11- Test the top two numbers for '<>'
12- Test the top two numbers for '>'
13- Test the top two numbers for '<'
14- Test the top two numbers for '='
15- Add the last two numbers together.
16- Logically 'AND' a string and a number.
17- Test the top two strings for '<='
18- Test the top two strings for '>='
19- Test the top two strings for '<>'
20- Test the top two strings for '>'
21- Test the top two strings for '<'
22- Test the top two strings for '='
23- Concatenate the top two strings.

Note: All the above operations (except operation '0') are binary operations and require two operands. The result 'overwrites' the first operand and this creates a 'new last value,' the second operand is thereby 'deleted.'

Note: A logical operation will, if the result is FALSE, return a 'last value' of zero and, if TRUE, a 'last value' of not-zero, usually one.

24- Perform the 'unary minus' operation.
25- Perform the 'CODE' function.
26- Perform the 'VAL ' function.
27- Perform the 'LEN ' function.
28- Perform the 'SIN ' function.
29- Perform the 'COS ' function.
30- Perform the 'TAN ' function.
31- Perform the 'ASN ' function.
32- Perform the 'ACS ' function.
33- Perform the 'ATN ' function.
34- Perform the 'LN ' function. (Note: Decimal 34)
35- Perform the 'EXP ' function.
36- Perform the 'INT ' function.
37- Perform the 'SQR ' function.
38- Perform the 'SGN ' function.
39- Perform the 'ABS ' function.
40- Perform the 'PEEK' function and thereby return the 'last value' equalling the value held in a location.
41- Perform the 'USR ' function and return with the 'last value' equalling the value of the BC register pair.
42- Perform the 'STR$' function.
43- Perform the 'CHR$' function.
44- Perform the 'NOT ' function.

Note: All the above functions overwrite the 'last value' with the result, i.e., they modify the 'last value' as required.

45- The 'last value' is simply duplicated.
46- A MODULUS operation. The first operand, N, is overwritten with the 'remainder,' and the 'last value,' M, is overwritten with the 'quotient,' INT (N/M).
47- A simple jump.
48- A STACK DATA operation. The following 2-5 bytes are treated as data bytes, expanded and passed to the stack.
49- A DECREASE THE COUNTER operation. The system variable BERG can be used as a counter and this operation in effect performs a DJNZ instruction.
50- Jump if 'last value' is negative.
51- Jump if 'last value' is positive.
52- The END-CALC. operation that forces an EXIT from the calculator subroutine.
53- A REDUCE ARGUMENT operation to modify the argument of a SIN and a COS.
54- A TRUNCATE operation that modifies the 'last value' to give its 'integer towards zero' result.
55- The special SINGLE OPERATION used by the EXPRESSION EVALUATOR.
56- An E-TO-FP operation in which the first operand is multiplied by 10 to the power of the 'last value.'
134- Call the SERIES GENERATOR passing to it 6 constants.
136- Call the SERIES GENERATOR passing to it 8 constants.
140- Call the SERIES GENERATOR passing to it 12 constants.
160- STACK ZERO. Put the value zero on the stack.
161- STACK ONE. Put the value one on the stack.
162- STACK a HALF. Put the value 1/2 on the stack.
163- STACK PI/2. Put the value PI/2 on the stack.
164- STACK TEN. Put the value ten on the stack.
192- 'ST-MEM-0'. Store the 'last value' in MEM-0.
193- 'ST-MEM-1'. Store the 'last value' in MEM-1.
194- 'ST-MEM-2'. Store the 'last value' in MEM-2.
195- 'ST-MEM-3'. Store the 'last value' in MEM-3.
196- 'ST-MEM-4'. Store the 'last value' in MEM-4.
197- 'ST-MEM-5'. Store the 'last value' in MEM-5.
224- 'GET-MEM-0'. Put the value from MEM-0 on the stack.
225- 'GET-MEM-1'. Put the value from MEM-1 on the stack.
226- 'GET-MEM-2'. Put the value from MEM-2 on the stack.
227- 'GET-MEM-3'. Put the value from MEM-3 on the stack.
228- 'GET-MEM-4'. Put the value from MEM-4 on the stack.
229- 'GET-MEM-5'. Put the value from MEM-5 on the stack.

Note: Operations 48, 160-164, and 224-229 all provide a means of putting additional numbers on the calculator stack.

Note: The printing of a floating-point number is not done by a CALCULATOR operation but by the PRINT A FLOATING-POINT NUMBER subroutine at 15D7—unimproved ROM (15DB—improved ROM)

Note: Storing a 'last value' does not 'delete' it.

'YOUR FIRST WORDS'

In order to write machine code subroutines that use the CALCULATOR LANGUAGE it is simply a matter of following the rules:

a) Call the calculator.
b) Stack your numbers.
c) Manipulate them.
d) Exit from the calculator.

Program 2—*Producing A Constant* shows a very simple operation being performed. As it is written, the operation 160 is used to make the 'last value' on the stack a zero and this zero is subsequently printed.

The numbers that are available as constants are limited and therefore an alternative method of passing values from Basic to the calculator stack is needed. Program 3—*Stack Any Number* is therefore the next stage to be reached. This program shows how numbers can be transferred to the stack via the calculator's memory area. In the program the 5 bytes of the floating-point form are transferred to MEM-O and then in the CALCULATOR PROGRAM the number is copied from MEM-O by using operation 224.

Overall View

In this article I have tried to show the essential features of the CALCULATOR LANGUAGE, but I have kept away from discussing how the calculator subroutine might be used. However, it would be useful to include a slightly more complicated example from the 8K program.

The function TAN modifies the value of the 'last value' so that it goes from say, X to TAN X. In the 8K ROM program the actual subroutine is situated at 1D6D—unimproved ROM (16DE—improved ROM). See Figure 2.

Figure 2.

RST 0028	- With X as a 'last value' call the calculator. Stack holds X.
DEFB +2D	- Operation 45 and the 'last value' is duplicated. Stack holds X, X.
DEFB +1C	- Operation 28 and SIN X is found. Stack holds X, SIN X.
DEFB +01	- Operation 1—exchange the values. Stack holds SIN X, X.
DEFB +1D	- Operation 29 and COS X is found. Stack holds SIN X, COS X.
DEFB +05	- Operation 5. The two values are divided. Stack holds SIN X/COS X and as TAN X = SIN X/COS X always the stack holds only TAN X.
DEFB +34	- Operation 52—leave the calculator.
RET	- Return to the calling subroutine.

Any reader who wishes to understand the CALCULATOR LANGUAGE is urged to read:

SINCLAIR ZX81 ROM Disassembly, Part A: 0000H-0F54H, by Dr. Ian Logan, and *Part B: 0F55H-1DFEH*, by Dr. Ian Logan and Dr. Frank O'Hara. Part A deals mainly with the operating system whereas Part B deals mainly with the CALCULATOR LANGUAGE as discussed above. The books are published by Melbourne House (Publishers) Ltd.

This month's program is not a game but a program that shows the 'randomness' of RND. (A ZX80 4K ROM program of this type was included in *The ZX80 Companion*)

In the *Random Number Graph* the user is asked to provide a starting seed value for the random number generator and the program then takes the next 200 random numbers for its analysis.

The program produces a 'curve' that shows just how 'non-random' the pseudo-number generator actually is in the 8K ROM.

I would be interested to know if anyone finds a seed value that gives a straight line—remember that the sequence of random numbers can be changed by addition of a dummy call to RND.

Listing 1: A Basic Stack Calculator.

```
  10 DIM A(VAL "5")                    THE STACK HAS 5 LOCATIONS.
  20 LET Z=VAL "40"                    Z IS A DUMMY VARIABLE.
  30 LET P=SGN PI                      P WILL POINT TO THE FIRST
  40 CLS                                  LOCATIONS.
  50 FOR B=VAL "5" TO SGN PI STE
P -SGN PI
  60 PRINT "LOC. ";B;" HOLDS",A(
B)
  70 NEXT B
  80 PRINT AT VAL "8",NOT PI;""""  SHOW WHICH LOCATION WAS THE
LAST"" LOC. IS ";P-SGN PI              "LAST LOCATION".
  90 PRINT AT VAL "13",NOT PI;"E
NTER CODE"
 100 PRINT "(1-IN.,2-PR.,3-ADD,4  ACCEPT THE OPERATION CODE.
-SUB,5-MULT)"
 110 INPUT B
 120 LET P=P-SGN PI                   DECREASE THE POINTER.
 130 GOTO B*VAL "200"                 JUMP ACCORDING TO THE CODE.
     INPUT A NUMBER
 200 PRINT AT VAL "19",NOT PI;"I
NPUT VALUE"
 210 INPUT B                          ACCEPT A VALUE.
 220 LET A(P+SGN PI)=B                ASSIGN THE VALUE.
 230 LET P=P+VAL "2"                  INCREASE THE POINTER.
 240 GOTO Z                           START AGAIN.
     PRINT A NUMBER
 400 PRINT AT VAL "19",NOT PI;"L
AST VALUE = ";A(P)                    PRINT THE "LAST VALUE".
 410 PAUSE Z*Z                        WAIT FOR A KEY TO BE PRESSED.
 420 GOTO Z                           START AGAIN.
     ADD TWO NUMBERS
 600 LET A(P-SGN PI)=A(P-SGN PI)      ADD THE NUMBERS.
+A(P)
 610 GOTO Z                           START AGAIN.
     SUBTRACT TWO NUMBERS
 600 LET A(P-SGN PI)=A(P-SGN PI)      SUBTRACT THE NUMBERS.
-A(P)
 610 GOTO Z                           START AGAIN.
     MULTIPLY TWO NUMBERS
1000 LET A(P-SGN PI)=A(P-SGN PI)      MULTIPLY THE NUMBERS.
*A(P)
1010 GOTO Z                           START AGAIN.
```

Note: In order to get this program working in a 1K machine, the values zero and one have been replaced by NOT PI and SGN PI respectively; all other integers are entered as VAL strings and the dummy variable Z is used. If more than 1K RAM is available, then none of these features need to be included.

Listing 2: Producing a Constant.

```
  5 REM USE THE PRINTER BUFFER
 10 LET A=16444
 15 REM CALL THE CALCULATOR
 20 POKE A,239                        RST 0028
 25 REM CHOOSE AN OPERATION IN
       THE RANGE 160-164.
 30 POKE A+1,160                      OPERATION "160"
 35 REM EXIT FROM CALCULATOR
 40 POKE A+2,52                       OPERATION "52"
 45 REM CALL PRINT-NUMBER
 50 POKE A+3,205                      CALL 15D7 (OR 15DB)
 60 POKE A+4,215                      (USE 219 FOR IMPROVED ROM)
 70 POKE A+5,21
 75 REM EXIT FROM USR
 80 POKE A+6,201                      RET
 85 REM  A SIMPLE HEADER
 90 PRINT "LAST VALUE WAS ";
 95 REM RUN MACHINE CODE
100 RAND USR A
```

Listing 3: Stack Any Number.

```
  5 REM ENTER YOUR NUMBER
 10 INPUT N
 15 REM TRANSFER IT TO MEM-0
 20 LET V=PEEK 16400+256*PEEK 1
6401
 30 FOR B=1 TO 5
 40 POKE 16476+B,PEEK (V+B)
 50 NEXT B
 60 LET A=16444
 70 POKE A,239
 75 REM CHOOSE YOUR OPERATIONS
NOW AND INCLUDE OPERATION "224"
 80 POKE A+1,224
 90 POKE A+2,52
100 POKE A+3,205
110 POKE A+4,215
120 POKE A+5,21
130 POKE A+6,201
140 PRINT "THE NUMBER WAS ";
150 RAND USR A
```

Note: The transfer of the 5 bytes from the variable area can be done in machine code as shown in Figure 3.

```
RST 0028

OPERATION "224"
OPERATION "52"
CALL 15D7 (15DB)
(USE 219 FOR IMPROVED ROM)

RET
```

Figure 3.

```
LD      HL,(VARS)
INC     HL
LD      BC,+0005
LD      DE,(STKEND)
LDIR
LD      (STKEND),DE
```

Use STKEND for direct transfer to the stack, or MEMBOT to use MEM-O. (stacking only)

Listing 4: The Random Number Graph Program

```
 10 PRINT AT VAL "18",NOT PI;"E
NTER NUMBER"
 20 INPUT N                          SUPPLY A SEED 0-65535
 30 CLS
 40 PRINT "RANDOM NUMBER GRAPH"
 50 PRINT "====================="
 60 RAND N
 70 DIM A(VAL "10")                  THE ARRAYS FOR THE 10 GROUPS
 80 DIM B(VAL "10")
 90 FOR N=SGN PI TO VAL "100"        TAKE 100 CALLS TO RND
100 LET R=SGN PI+INT (VAL "10"*      AND CREATE GROUPS FOR EACH TENTH
RND)                                 IE.  .1, .2,.... .9
110 LET A(R)=A(R)+SGN PI             COUNT THE GROUPINGS
120 NEXT N
130 FOR N=SGN PI TO VAL "5"          SHAPE THE GRAPH BY TAKING THE
140 LET K=N                          OUTER PAIR, THE NEXT PAIR,ETC
150 GOSUB VAL "300"
160 LET K=VAL "11"-N
170 GOSUB VAL "300"
180 NEXT N
190 FOR N=SGN PI TO VAL "10"         PRINT THE GRAPH
200 PRINT "+";TAB B(N);"▒"           GRAPHIC SHIFTED A
210 NEXT N
220 PRINT "01234567890123456789      PRINT THE SCALE
"
230 RUN
300 LET M=VAL "50"                   SET M AS A UPPER LIMIT
310 FOR J=SGN PI TO VAL "10"         LOOK AT THE 10 SCORES
320 IF M<=A(J) THEN GOTO VAL "3      ASSIGN THE LOWEST TOTAL TO B(K)
50"
330 LET M=A(J)
340 LET L=J
350 NEXT J
360 LET B(K)=A(L)                    SET EACH "LOWEST TOTAL" TO A
370 LET A(L)=VAL "40"                HIGH VALUE, TO SHOW IT HAS
380 RETURN                           BEEN USED
```

RUN—I suggest in FAST.

Note: If more numbers would be preferred, change, e.g., as follows:

For 10,000 random number:

90 FOR N=SGN PI TO VAL "10000"

and

110 LET A(R)=A(R)+VAL ".01"

This gives an almost straight line from a starting seed of 1.