THE GJM MODULE (REV 2B) FOR THE HP41

By Gregory J. McClure, August 2016

Table of Contents

1. Introduction

The GJM Module is a collection of routines that I like that makes my programming faster, easier, and with fewer bytes of code needed. They were written with the requirement that Ángel Martin's Library #4 be installed. Some of the FOCAL routines will require other modules be installed; the documentation will describe what modules are needed. I have written some routines myself, but many routines are from the rich public domain of routines from other fine MCODE and FOCAL programmers. Hopefully you will find this collection of routines as useful as I have.

Much of the organization of this module has been a result of my work with Ángel Martin on our shared module, the HP16C Emulator. The GJM module uses two banks and has a 2nd FAT similar to Ángel's SandMath, SandMatrix, and HP16C Emulator modules. GJM also has a LastF function, and creates a buffer (size 6) that allows the saving of the last 2nd FAT function that was executed. Much thanks goes to Ángel for his help and suggestions for making this module what I really have wanted… he took on the role of consultant and code reviewer, helping me better use the system routines as well as Lib4 routines, and his help has been greatly appreciated!

2. Module Requirements

This module requires (as mentioned above) the latest Library #4 module by Ángel Martin. It must be version Q or higher. This module sets itself up in page 4 (the takeover ROM position). It is not a takeover ROM, but supplies tons of common routines useful for multiple modules (Ángel makes full use of these routines for his AMC OS/X, SandMath, SandMatrix, HP16C emulator and other modules). The GJM module will check for that module on power up. If it doesn't exist it will warn the user. It is NOT recommended to continue using this module if you get this error message.

This module also requires an HP41CX (or similar O/S installed in an emulator). It just doesn't make sense to write code for the HP41 without the CX extensions, since this is required by Library #4 as well. This module will check for this, and if not a CX, it will warn the user. It is NOT recommended to continue using this module if you get this error message.

The LastF function will require room for the LastF buffer in memory. Since this buffer is used by Ángel Martin in some of his other modules, it may already be present. If used before the latest version of Library #4 was installed (version Q), the buffer *may have been created with the wrong size*. The problem with using the wrong size buffer is that *some functions that require this buffer may hang up* (infinite loop) because the buffer is not the right size. If you have the AMC OS/X or RAMPAGE module plugged in, you can check the size of buffer 9, the size should now be 6. If the buffer size is 5, it must be removed. *To remove the bad size buffer*, either use the DELBUF function from the RAMPAGE module, or turn off the calculator then remove all modules with a LastF function (SandMath, SandMatrix, HP16C, and GJM) and then turn the calculator back on. Then turn it off and reinsert the modules with the proper Library #4 version installed and turn it back on. If the buffer does not exist and no room is available, you will be warned when trying to execute the LastF function. It is not fatal, but just an inconvenience if you want to use LastF.

3. Main FAT functions

3.1 Main FAT functions at a glance

Here is a list of the functions in the Main FAT table (the one you get with CAT 2).

XROM #	Function	Description	Author
31,00	**-GJM REV 2A**	Rom Header	Greg McClure
31,01	**GJM#**	Prompts for / executes 2nd FAT function number	Ángel Martin
31,02	**GJM$**	Prompts for / executes 2nd FAT function name	Ángel Martin
31,03	**GJM**	GJM functions launcher	Ángel Martin
31,04	**CD**	Move “curtain” down	Greg McClure
31,05	**CU**	Move “curtain” up	Greg McClure
31,06	**X+1**	Increments X by 1	Ángel Martin
31,07	**X-1**	Decrements X by 1	Ángel Martin
31,08	**X/2**	Divides X by 2	Ángel Martin
31,09	**X*E3**	Multiplies X by 1000	Jean-Marc Baillard
31,10	**X/E3**	Divides X by 1000	Jean-Marc Baillard
31,11	**X<>Y?**	Tests if X not approx. equal to Y	Jean-Marc Baillard
31,12	**X=YZ?**	Tests if X=Y or X=Z	Ken Emery
31,13	**X=YZT?**	Tests if X=Y, X=Z, or X=T	Paul Kaarup
31,14	**M+**	Modular addition	Greg McClure
31,15	**M-**	Modular subtraction	Greg McClure
31,16	**M***	Modular multiplication	Greg McClure
31,17	**SQM**	Modular square	Greg McClure
31,18	**M^**	Modular power	Jean-Marc Baillard
31,19	**1/M**	Modular inverse	Jean-Marc Baillard
31,20	**SQRTM**	Modular square root	Jean-Marc Baillard
31,21	**LASTF**	Last 2nd FAT function re-execute	Ángel Martin
31,22	**E3/E+**	Divides X by 1000 then increments X	Ángel Martin
31,23	**E-E3***	Decrements X then multiplies by 1000	Greg McClure
31,24	**1+X/1-X**	Returns (1+X)/(1-X)	Greg McClure
31,25	**X-1/X+1**	Returns (X-1)/(X+1)	Greg McClure
31,26	**X+XE3/**	Returns X + X/1000	Greg McClure
31,27	**Y-X/Y+X**	Returns (Y-X)/(Y+X)	Ángel Martin
31,28	**-SLIDE RULE**	Slide Rule program header	Greg McClure
31,29	**SR**	Slide Rule program	Greg McClure
31,30	**-SUBSCALES**	Slide Rules Subscales header	Greg McClure
31,31	**P**	Pythagorean scale	Greg McClure
31,32	**Q**	Cube Root scale	Greg McClure
31,33	**W**	Square Root scale	Greg McClure
31,34	**L**	Log10 scale	Greg McClure
31,35	**LE**	LogE scale	Greg McClure
31,36	**CIR**	Circumference scale	Greg McClure

31,37	**U**	User Defined scale	Greg McClure
31,38	**S**	Sine scale	Greg McClure
31,39	**CC**	Cosine scale	Greg McClure
31,40	**T**	Tangent scale	Greg McClure
31,41	**LLM**	Log Ln Minus scale	Greg McClure
31,42	**K**	Cube scale	Greg McClure
31,43	**R**	Square scale	Greg McClure
31,44	**LLD**	Log Log scale	Greg McClure
31,45	**LL**	Log Ln scale	Greg McClure
31,46	**-FOCAL PROGS**	FOCAL programs header	Greg McClure
31,47	**AOS**	Algebraic Operating System emulator	Greg McClure
31,48	**CSPLINE**	Cubic Spline Interpolation	Greg McClure
31,49	**LDVW**	Load/View Registers	Greg McClure/ Ángel Martin
31,50	**LDVWD**	Load/View Data File	Greg McClure/ Ángel Martin
31,51	**NLSEQ**	Non-Linear Simultaneous Equations solver	Jean-Marc Baillard/ Greg McClure
31,52	**CONG**	Congruency solver	Greg McClure/ Ángel Martin
31,53	**-CONT FRC**	Continuous Fractions program header	Greg McClure
31,54	**XEQA**	Execute Program named in Alpha	Greg McClure
31,55	**CF2V**	Continuous Fractions Value program	Greg McClure
31,56	**-DERIVATIVES**	Derivatives program header	Greg McClure
31,57	**DERV**	1st and 2nd Derivative of User Defined Function	Greg McClure
31,58	**HEX2ROM**	HEX Bytes to XROM Value	Greg McClure
31,59	**ROM2HEX**	XROM Value to HEX Bytes	Greg McClure
31,60	**AMEAN**	Arithmetic Mean of Register Range	Ángel Martin
31,61	**GMEAN**	Geometric Mean of Register Range	Ángel Martin
31,62	**HMEAN**	Harmonic Mean of Register Range	Ángel Martin
31,63	**PMEAN**	Power Mean of Register Range (Generalized)	Ángel Martin

3.1.1 Differences between GJM 2A and GJM 2B modules

A problem with the **ROM2HEX** and **HEX2ROM** functions was reported. Seems it worked just fine with the V41 emulator, but didn't work on actual HP41CX and HP41CL machines. It turns out that V41 is the only emulator that gave accurate results, the HP41E and GO41CX (Android) showed a similar error to the HP41CX/CLs. This is because the V41 emulator does not emulate the display read functions the same as the actual HP41CX/CL. Thus the routines **ROM2HEX** and **HEX2ROM** needed to be modified to work on V41 as well as actual HP41CX/L machines. This is the only change made to the GJM module.

3.2 Main FAT functions in detail

3.2.1 Execution of 2nd FAT functions

GJM# and **GJM$** are functions to look up and execute functions in the 2nd FAT. This allows going beyond the limitation of 64 functions/headers in the Main FAT table. These functions are described in sections 2.3 and 2.4.

To execute a 2nd FAT function by number (these functions start from 0, similar to XROM numbers) use **GJM#** and enter the function number (preceded by leading 0's) at the prompt. When entered into a user program, this will automatically enter both the GJM# function and the function number as separate steps. Function **GJM#** will use this value as the function number, it will not be interpreted as a separate value to be entered into X. It is important NOT to follow this value with another numeric value, as this confuses **GJM#**. Be sure at least one non-numeric entry instruction separates the function number following **GJM#** and the next numeric value.

To execute a 2nd FAT function by name, use **GJM$** and enter the function name at the alpha prompt. When done in a program, this will be converted to separate GJM# function and function number steps. The same requirements are needed as in **GJM#** with regard to not following the **GJM#** and function number steps with another value (at least one non-numeric entry must be put immediately after the **GJM#** function number). Even when executed in a program the function executed will be saved in the special **LastF** function buffer created by Ángel Martin, unless that buffer does not exist.

The **LastF** function in the GJM module will perform the last GJM# / GJM$ function performed.

3.2.2 The GJM Launcher

The **GJM** function is a quick launcher for most of the functions in this ROM. It allows one or two (Gold) keystrokes to execute either main FAT or 2nd FAT functions. Thus one assignment (**GJM** to some key of the user's choice) give quick access to these functions. The SST key acts as a shift to an alternate set of functions for the keys (prompt will show GJM2 instead of GJM), pressing it again brings back the original GJM function definitions. Holding the final key of the sequence down acts as a preview and keeping it down will cancel the function (except for the Gold key, which is cancelled by hitting Gold again, and similarly the SST key).

The following table shows all one-time key assignments (by group) for the **GJM** function (Some of the functions are from the 2nd FAT, and explained in section 4):

Group	Key(s)	Original key(s) meaning	Function performed
Modulus Functions	U	+	**M+**
"	Q	-	**M-**
"	B	1/X	**1/M**
"	Gold B	Y^X	**M^**
"	C	SQRT	**SQRTM**
"	Gold C	X^2	**SQM**
"	Y	*	**M***
"	:	/	**CONG**
Means Functions	A	Σ+	**AMEAN**
"	G	RDN	**GMEAN**
"	H	SIN	**HMEAN**
"	P	EEX	**PMEAN**
ROM Address	R	7	**ROM2HEX**
"	S	8	**HEX2ROM**
"	Gold R	SF	**R2HXAPP**
"	Gold S	CF	**HX2RAPP**
Duplex Means	Gold A	Σ-	**AGM**
"	Gold G	%	**GHM**
"	E	LN	**ELIPE**
"	K	XEQ	**ELIPK**
Quick Math Functions	SST Q	SST -	**X-1**
"	SST U	SST +	**X+1**
"	SST Y	SST *	**X*E3**
"	SST :	SST /	**X/E3**
"	SST ?	SST 3	**E3/E+**
"	SST Gold ?	SST ENG	**E-E3***
"	SST Z	SST 1	**1+X/1-X**
"	SST V	SST 4	**X-1/X+1**
"	SST Gold V	SST BEEP	**X+XE3/**
"	SST Gold Y	SST X>Y?	**Y-X/Y+X**
Test Functions	Gold Q	X=Y?	**X<>Y?**
"	Gold U	X<=Y?	**X=YZ?**
"	Gold Y	X>Y?	**X=YZT?**
"	?	3	**PRIME?**
Prime Functions	Z	1	**PFACT**
"	Gold =	SCI	**PTWIN**
"	=	2	**NXTPRM**
Math Functions	F	X<>Y	**FLOOR**
"	Gold F	CLΣ	**FRC2**
"	Gold Space	PI	**MAXR**
"	Gold Z	FIX	**NORM**
"	Space	0	**RND0**
"	X	6	**XROOT**
"	Gold X	R>P	**XY^**

FOCAL Programs	SST S	SST CF	**SR**
“	SST A	SST Σ+	**AOS**
“	SST C	SST SQRT	**CSPLINE**
“	SST N	SST ENTER^	**NLSEQ**
“	SST L	SST STO	**LDVW**
“	SST Gold L	SST LABEL	**LDVWD**
Curtain Functions	D	LOG	**CU**
“	Gold D	10^X	**CD**
Remaining Functions	SST D	SST LOG	**DERV**
“	SST F	SST X<>Y	**CF2V**
“	SST R/S	SST R/S	**CURFL**
“	Gold R/S	VIEW	**VA**
“	R/S	R/S	**STVIEW**
“	SST Gold R/S	SST VIEW	**XEQA**
“	PRGM	PRGM	**GJM#**
“	ALPHA	ALPHA	**GJM$**
“	SST PRGM	SST PRGM	**GJM#**
“	SST ALPHA	SST ALPHA	**GJM$**

3.2.3 Curtain functions

The absolute register number that marks the location of R00 is often called the “curtain”. Its value is kept in one of the system stack registers (c to be specific), as most synthetic programmers know. The system moves this value up or down depending on how many registers are specified with the SIZE instruction. A trick used by some synthetic programmers is to raise or lower the value of the “curtain” in a program. If the value is raised by n, then R0 thru Rn-1 are hidden, and Rn becomes R0, Rn+1 becomes R1, etc. If the value is subsequently lowered by n (which must be done before exiting the program for reasons explained next) then these hidden registers are recovered and the original register numbers are restored.

Actually, when the “curtain” is raised in this way, the n registers affected temporarily become program steps as far as the O/S is concerned. Since this could confuse the system when doing a CAT 1, “curtain” raising and lowering should be used carefully. In addition, if a PACK occurs while the “curtain” is raised like this, the hidden registers could easily (and probably will) change values. If the “curtain” is raised to temporarily save registers, it should be lowered back before doing these system functions (CAT 1 or PACK, or similar functions). Conversely, if the “curtain” is lowered and not raised back to its original value, certain labels and ENDs could be modified by simple RCL, STO and X<> instructions. This messes up the chain of CAT 1 and can lead to MEMORY LOST. However, used properly, “curtain” manipulation can be of great use to a programmer that needs to call a subroutine that uses the same registers as another program.

Function **CU** raises the curtain up. The function will prompt for the number of registers to hide. If this function is entered in a program, the number of registers to hide should be entered as a value after the **CU** function. This will be interpreted as the argument of **CU**, not as a value to enter into X. It must be followed by a non-numeric entry function or the **CU** will get confused (same caution as for **GJM#** function).

Function **CD** lowers the curtain down. The function will prompt for the number of registers to restore. If this function is entered in a program, the number of registers to restore should be entered as a value after the **CD** function. This will be interpreted as the argument of **CD**, not as a value to enter into X. It must be followed by a non-numeric entry function or **CD** will get confused (same caution as for **GJM#** function).

3.2.4 Simple X Manipulation functions

The following functions simply do a quick action on X, without disturbing any other stack registers, including LastX (speed was considered the important factor here). If the register needs to be restored (for whatever reason) a similar function is usually available to execute.

X+1 increments the X register by 1. To restore X, use **X-1**. (Ángel Martin)

X-1 decrements the X register by 1. To restore X, use **X+1**. (Ángel Martin)

X/2 halves the X register. (No need for X*2 since **ST+ X** will do the same thing, and is the restore function as well). (Ángel Martin)

X*E3 multiplies the X register by 1000. To restore X, use **X/E3**. (Jean-Marc Baillard) See special note below.

X/E3 divides the X register by 1000. To restore X, use **X*E3**. (Jean-Marc Baillard) See special note below.

E3/E+ first divides X by 1000, then increments X. To restore X use **E-E3***. (Ángel Martin)

E-E3* first decrements X, then multiplies X by 1000. To restore X use **E*E3/**. (Greg McClure)

X+XE3/ adds X/1000 to X (basically multiplies X by 1.001, to restore X (and destroy T), do **1.001, /**). (Greg McClure)

1+X/1-X adds and subtracts X from 1, then divides the sum by the difference. The restore function is **X-1/X+1**. (Greg McClure)

X-1/X+1 subtracts and adds 1 to X, then divides the difference by the sum. The restore function is **1+X/1-X**. (Greg McClure)

Y-X/Y+X divides (Y-X) by (Y+X). To restore do **CHS, 1+X/1-X, X<>Y, ST* Y, X<>Y** (Ángel Martin)

Special note concerning **X*E3** and **X/E3**: The microcode for these functions simply raises or lowers the exponent by 3. It is possible for **X*E3** to raise the exponent >99, or **X/E3** to lower the exponent <-99. This is NOT an error, but will not display properly. If you do a LOG on the number, you will see that indeed the exponent was >99 or <-99. This may or may not be what you want, and only a few functions will work with such numbers. For example **E 99, X*E3, E 3, /** will work (since final result is in range), but **E 99, X*E3, 5, *** will not (it is OUT OF RANGE). You have been forewarned.

3.2.5 Stack Test functions

There is always a reason for more test functions, right? Here are three additional tests that might be useful.

X<>Y? is actually a test to see if X is not *approximately* Y. It checks if X and Y differ by < 2E-9. If so, they are considered approximately equal the next instruction is skipped, else the next instruction is executed. Jean-Marc Baillard created this function.

X=YZ? is a dual test. If X=Y then the next instruction is executed. If X<>Y but X=Z then the next instruction is skipped, but the following one is executed. If X<>Y and X<>Z then TWO instructions are skipped, and the one following those is executed. Ken Emery is the author of this function.

X=YZT? takes that one step further. It is a triple test. If X=Y then the next instruction is executed. If X<>Y but X=Z then the next instruction is skipped, but the following one is executed. If X<>Y and X<>Z but X=T then the next TWO instructions are skipped, and the one following those is executed. If X<>Y and X<>Z and X<>T then the next THREE instructions are skipped, and the one following those is executed. Paul Kaarup is the author of this gem.

3.2.6 Modulus Math functions

For those acquainted with modular math, the following modulus functions are provided:

M+ performs Z+Y MOD X. It works for values up to 10 digits and takes into consideration the sign of the values. M+ handles differently signed parameters. It is in MCODE and uses 13-digit math, making it much faster.

M- performs Z-Y MOD X. The same comments apply as in **M+**.

M* performs Z*Y MOD X. The same comments apply as in **M+** and **M-**.

SQM performs Y^2 MOD X. It really uses much of the same code as **M***, it is actually doing a Y*Y MOD Z.

With the above MCODE routines, the following FOCAL functions taken from Jean-Marc Baillard run much quicker…

1/M performs 1/Y MOD X. This function may or may not have an answer. Remember, the definition is "Return the value that, when multiplied by Y MOD X yields 1". This value may not exist. The function will stop with "DATA ERROR" if this is the case.

SQRTM performs SQRT(Y) MOD X. This function will return either 0 (no solution) or the control number of the registers containing the answers. Remember, the definition is "Return the value that, when multiplied by itself, returns Y MOD X". So if 1.002 is returned to X then the answers are in R1 and R2. They should be considered as dual answers, that is, +R1, -R1, +R2, and –R2 (that would be 4 answers).

Where is ***M/*** ? Well actually this is the congruence function (if AX = B MOD C then X = B/A MOD C). The **CONG** function solves AX=B MOD C, expecting A in Z, B in Y, and modulus C in X. This may or may not yield an answer (for example 2X = 3 MOD 10 has no solutions), so it is possible that the function will stop with a DATA ERROR. If not, then X will contain the primary answer, and Y will contain the value that can be added or subtracted any integer number of times for the other answers. For example, to solve 2X = 4 MOD 10, do **2, ENTER^, 4, ENTER^, 10, CONG**; the result is X = 2, Y = 5. This means the solution set is {…, -8, -3, 2, 7, 12, 17 …}. The Alpha register is used, so it will be cleared if a solution is found. If not, then synthetic registers M, N, and O will contain the reduced A, B, and C (a GCD is performed on A, B, and C before 1/M is performed and this is saved in M, N, and O). This may be useful in determining why the DATA ERROR occurred. BTW I have listed Ángel Martin as a co-author, since he did much of the grunt work to help determine the method of solution needed. Once I read all the info, applying **1/M, GCD,** and **M*** was a simple matter.

3.2. Continued Fractions Evaluation functions

Continued Fractions are expressions of the form:

$$B(0)+\frac{A(1)}{B(1)+}\frac{A(2)}{B(2)+}\cdots\frac{A(N)}{B(N)+}\cdots$$

The use of + in the denominator indicates that the remainder of the terms actually are part of that denominator. So the above expression means B(0) + A(1) / [B(1) + A(2) / [B(2) + A(3) / […]]]. This can be mathematically abbreviated as B(0) + [A(1), A(2), A(3), … ; B(1), B(2), B(3), …] which will be used here. The number of expressions may or may not be infinite.

Many values are easily expressed as continuous fractions. Some examples are:

Tanh(x) = [X, X^2, X^2, X^2, … ; 1, 3, 5, 7, …]
Pi = [4, 1^2, 3^2, 5^2, 7^2, … 1, 2, 2, 2, 2, …] (one of MANY representations of Pi)
(2 / (e-1)) - 1 = [1, 1, 1, 1, … ; 6, 10, 14, 18, …] (again one of MANY representations of e)
Sqrt(2) - 1 = [1, 1, 1, 1, … ; 2, 2, 2, 2, …]

The simpler form of continuous fractions often used are expressions of the form:

$$B(0) + \frac{1}{B(1)+}\frac{1}{B(2)+}\cdots\frac{1}{B(N)+}\cdots$$

mathematically abbreviated as [B(0); B(1), B(2), B(3), …]. For example:

e = [2; 1, 2, 1, 1, 4, 1, 1, 6, 1, 1, 8, …]

Some expressions are not so easily represented in this form. For example:

Pi = [3; 7, 15, 1, 292, 1, 1, 1, 2, 1, 3, 1, …] (there is no pattern to express the next B(N)].

The **CF2V** program is designed to calculate a continued fraction value. It requires a user created subroutine that calculates A(N) and B(N) for N >= 1. The program makes X available in R01 and N available in R02 for this program, and expects A(N) in stack register X and B(N) in stack register Y on completion of the subroutine. The subroutine must be callable by a global label (of up to 7 characters). The program uses R00 thru R04.

To execute **CF2V**, put the value of B(0) in stack register Y, and the value of X to be saved in stack register X, then put the name of the routine that calculates A(N) and B(N) into the alpha register. Execute CF2V to evaluate the continued fraction.

This program uses a special function called **XEQA**. This function simply takes the name of a user function in the alpha register, and executes that function. It is like an XEQ IND nn except the name of the function is in the alpha register, and it MUST be a global routine in RAM. The name is restricted to 7 characters or less. For those interested, **XEQA** uses the name in the alpha register (specifically the M register), calls a special search function in the operating system (ASRCH), and if found to be a FOCAL program in RAM, executes the program starting that location.

The program also uses an internal hidden function (described below). It is NOT designed to be used in the user function created. Simply end that function with RTN or END. The operating system normally does not allow returning to MCODE from FOCAL programs. So one of the secrets of CF2V is that it calls **XEQA**, then this hidden function. That function then returns back to the **CF2V** MCODE after doing a real RTN. So for those interested, **CF2V** consists of the following:

Initialization code (check that registers 00 thru 04 exists, and initializes them)

Begin main loop which

> Bumps the counter (register 02)
>
> Performs a mini FOCAL program (**XEQA**, **-CONT FRC**),
>
> (**-CONT FRC** is the hidden function that gets us to the next step)
>
> **-CONT FRC** then remove the mini FOCAL program address from user stack (by doing a RTN)
>
> **-CONT FRC** checks end of loop conditions, and jumps back to main loop in **CF2V** if not done

If no more loops needed, then recall register 00 as the answer.

Here is an example of use of **CF2V**. Let's say we want to evaluate the Tanh function mentioned above. We would create the following program in memory (assume we use the label TT):

```
LBL "TT"                                 LBL 01
RCL 02        ; Get N from R02           -              ; N – 1 in X
1             ; Is it 1?                 RCL 02         ; Get N again
X#Y?                                     +              ; 2N – 1 in X
GTO 01        ; No, skip to LBL 01       RCL 01
RCL 01        ; B(1) = 1 in Y, A(1) = x in X   X^2      ; B(N) = 2N – 1 in Y, A(N) = x^2 in X
RTN                                      END
```

Enter 0 for B(0), ENTER^, 1 for x (to evaluate Tanh(1)), Alpha, "TT", Alpha and execute **CF2V**. The answer of 0.761594156 (assuming FIX 9) is displayed in a few seconds. The value returned should be accurate to at least 9 significant digits. Try 0, ENTER^ 2 (to evaluate Tanh(2)), execute CF2V, answer is 0.964027580. If you want to see the estimates as they are calculated, just put a VIEW 00 statement at the beginning of routine "TT".

A more interesting example might be the Incomplete Gamma functions. As you know the following relationship applies for Gamma functions:

$$\gamma(a,x) + \Gamma(a,x) = \Gamma(a)$$

Continued fractions exist for both lower and upper functions:

$$\Gamma(a,x) = [x^a e^{-z}, a-1, 2(a-2), 3(a-3), \ldots; 1+x-a, 3+x-a, 5+x-a, 7+x-a, \ldots]$$
$$\gamma(a,x) = [x^a e^{-x}, -ax, x, -(a+1)x, 2x, -(a+2)x, 3x, \ldots;\ a, a+1, a+2, a+3, a+4, a+5, \ldots]$$

Since this requires two variables, we need to store variable a in R05 before running each continued fraction value. Here are sample programs for calculating A(N) and B(N) (note use of other functions in GJM module):

```
01 LBL "UIG"
02 RCL 02        ; Get counter (N)
03 1
04 X#Y?         ; If not 1, skip
05 GTO 01
06 RCL 01        ; Calculate (1+x-a)
07 +
08 RCL 05
09 -             ; We now have B(1)
10 LBL 00        ; Get A(1)
11 RCL 01        ; Calculate x^ae^-x
12 RCL 05
13 Y^X
14 RCL 01
15 CHS
16 E^X
17 *             ; We now have A(1)
18 RTN
19 LBL 01
20 X<>Y          ; Calculate (n+x-1) where n=2N-1
21 ST+ X
22 X-1           ; GJM module function!
23 RCL 01
24 +
25 RCL 05
26 -             ; We now have B(N)
27 STO Y         ; Save due to next calculation
28 RCL 05        ; Calculate n(a-n) where n = N-1
29 RCL 02
30 X-1
31 STO Z
32 -
33 *             ; We now have A(N)
34 RTN
```

```
35 LBL "LIG"
36 RCL 02        ; Get counter (N)
37 1
38 X#Y?
39 GTO 02
40 RCL 05        ; We now have B(1)
41 GTO 00        ; Get A(1) and exit
42 LBL 02
43 RCL 02        ; Get N
44 2
45 MOD
46 X=0?
47 GTO 03
48 RCL 02        ; Odd N handling
49 X-1           ; Calculate nx where n=(N-1)/2
50 X/2
51 RCL 01
52 *             ; We now have A(N)
53 GTO 04
54 LBL 03        ; Even N handling
55 RCL 02        ; Calculate –(a+n)x where n = N/2-1
56 X/2
57 X-1
58 RCL 05
59 +
60 RCL 01
61 *             ; We now have A(N)
62 CHS
63 LBL 04        ; Calculate a+n where n = N-1
64 RCL 05
65 RCL 02
66 X-1
67 +             ; We now have B(N)
68 X<>Y          ; B(N) in Y, A(N) in X
69 END
```

Let's calculate $\Gamma(3,4)$ and $\gamma(3,4)$. Since B(0) is always 0 then we enter Alpha, "UIG", Alpha, 3, STO 05, 0, ENTER^, 4 (for a=3 and x=4) and XEQ CF2V. You should get 0.476206611. Now do Alpha, "LIG", Alpha, 0, ENTER^, 4 and XEQ CF2V. You should get 1.523793388 (which agrees with the value from ICGM in SandMath module except for the very last place). The sum of these values should be $\Gamma(3)$ or 2.0, it is 1.999999999, which is close enough!

If for some reason the calculation takes too long to wait for (all depends on what A(N) and B(N) are) the program can be stopped by pressing R/S. It is stopped somewhere in the user created routine. Register 00 will contain the last value calculated for the continued fraction, register 02 is the number of loops performed to that point.

The idea for this program comes from Jean-Marc Baillard (in fact the TANH example is from his documentation), but usage is slightly different (he puts the name of the routine to execute in R00, here that register is used for the solution), and my routine is MCODE instead of FOCAL. I use R03 for C(N) and R04 for D(N) in using the Modified Lentz formula for continued fraction evaluation.

3.2.8 Derivatives program

The **DERV** program is designed to take the 1st and 2nd derivatives of a global function defined by the user (visible via Catalog 1). The function needs to be continuous thru the range around the value at which the derivatives of the function are desired. The program uses R00 and R01 to sum the term evaluations obtained by the formula used, R02 is used for the value sent to the user program, R03 is the saved step size, and R04 to contain the counter (which goes from 0 to 10).

The program takes two values, the value of the function to evaluate the derivatives, and the step size to use for the derivative evaluation formula (this is the distance between points sampled). When developing this program, many formulas were available to use… this program uses the 10 point formulas developed by Jean-Marc Baillard. It takes a bit longer than the formulas used by the PPC module, but the accuracy is far better. The PPC module FD program does have one advantage, it finds a one-sided formula (in case of discontinuity at a point), but it does not contain a 2nd derivative program.

The program also uses an internal hidden function (described below). It is NOT designed to be used in the user function created. Simply end that function with RTN or END. The operating system normally does not allow returning to MCODE from FOCAL programs. So one of the secrets of DERV is that it calls **XEQA**, then this hidden function. That function then returns back to the **DERV** MCODE after doing a real RTN. So for those interested **DERV** consists of the following:

Initialization code (check that registers 00 thru 04 exists, and initializes them)
Begin main loop which

> Bumps the counter (register 04) [skipped the first time]
> Performs a mini FOCAL program (**XEQA**, **-DERIVATIVES**)
> > (**-DERIVATIVES** is the hidden function that gets us to the next step)
>
> **-DERIVATIVES** then remove mini FOCAL program address from user stack (basically does a RTN)
> **-DERIVATIVES** checks if the counter is 10, finishes up if so
> If not finished then increments the counter and jumps back to main loop in **CF2V**

If no more loops needed, then the program recalls registers 00 and 01 as the first and second derivatives.

An example of program use is in order:

Let's say we want to find the derivative of f(x) = sin(x) at x=1. First we need to create a Global label program…

```
LBL "AA"
RAD
SIN
END
```

BTW, RAD can be removed if manually set *before* executing **DERV**. Let's try a step value of .03 (so the points sampled will be (.85, .88, .91, …, 1.12, 1.15). So .03, ENTER^, 1, ALPHA, "AA", ALPHA, XEQ **DERV**. On return, X contains 0.540302302 (the actual 1st derivative is 0.54032306) and Y contains -0.841470900 (the actual 2nd derivative is -0.841470985). BTW for testing the sine function for other values and step sizes, f'(sin(x)) = cos(x)dx, and f''(cos(x)) = -sin(x)dx^2, that is to say, you can test the values obtained by this program for this example by taking the cos(x) and –sin(x) for the actual 1st and 2nd derivative values.

3.2.9 Slide Rule program

I have always wanted an electronic version of a real slide rule (with A, B, C, and D type scales and all kinds of subscales), with 10-digit accuracy on setting and reading the scales, and automatic positioning of the decimal… Well, here it is! Program **SR** emulates a slide rule with 20 main scales and 15 possible subscales for each main scale. **SR** initializes the program, and sets the virtual hairline to scales C and D at position 1.

Usage of this program and examples of use can be found in Appendix A – Slide Rule program usage.

3.2.10 AOS program

The **AOS** (Algebraic Operating System) program is designed to allow entry of data and operations using operations and parenthesis as written. The partial answers are saved in Extended Memory in a small file created by the user when **AOS** initializes. It follows operation hierarchy (* is performed before +, etc).

Usage of this program and examples of use can be found in Appendix B – AOS program usage.

3.2.11 CSPLINE program (and use of LDVWD loader)

Cubic Spline interpolation is a method of creating a smooth curve between Cartesian points plotted. **CSPLINE** takes a user created data file of X,Y data pairs in Extended Memory and creates the array solution required to display any interpolated X,Y data pair desired. It also shows the slope of the line at that point for easier sketching of the solution. The **LDVWD** program can assist in creating the data file in Extended Memory. ***This program requires the Advantage or SandMatrix module.***

Usage of this program and examples of use can be found in Appendix C – Cubic Spline program usage. It includes instructions on how to use **LDVWD**.

3.2.12 NLSEQ program (and use of LDVW loader)

Linear simultaneous equation solutions are easily solvable using either the Advantage or SandMatrix module simultaneous equation functions. Non-linear simultaneous equation solutions are NOT so easy to solve. However, Jean-Marc Baillard has come up with a method for solving these equations. He may not have had the Advantage or SandMatrix modules available, so his FOCAL routines do all the work (including simultaneous equation solving subsection) in main memory.

With **NLSEQ**, I have adapted Jean-Marc's program to allow it to perform the solution section using Extended Memory Matrix files. Thus ***the Advantage or SandMatrix module is required to run this program***. The **LDVW** program can assist in creating the estimated solution arrays in Main Memory (two such guess arrays are required) and to read the solution vector. Main memory is also required to enter the non-linear function definitions for use.

Usage of **NLSEQ** is identical to Jean-Marc Baillard's NLS program (see his instructions and examples at http://hp41programs.yolasite.com/system-eq.php). The non-linear functions need to be created in RAM, the initial guesses and function names need to be placed in registers starting R1, and the size in R0. Execute **NLSEQ** and when the program stops the solution will replace the guess1 vector registers. For example, if a 4x4 solution is needed, 4 goes into R0, guess1 goes into R1 thru R4, guess2 goes into R5 thru R8, and function names go into R9 thru R12. R1 thru R4 will contain the final solution found by **NLSEQ**. Instead of using memory to create the solution matrix and vector, it uses Extended Memory Matrices (*M size 4x4, and *S size 4x1, and then solves them using MSYS.

To use **LDVW**, put the control word for the data registers to modify (in the above example, we want to put in two 4 element guesses, so that would be 1.008) into X, clear flag 08 if we want to enter the data, set flag 08 if we only want to view the data.

Here is an example from Baillard's documentation for a 4 element non-linear simultaneous equation set:

Equations to solve are:

$W + X + Y + Z - 16 = 0$

$WXY - 3Z = 0$

$4W^2 - XYZ - 40 = 0$

$WXYZ - 140 = 0$

The following program should be entered by the user (assuming global labels chosen are F1, F2, F3, F4):

```
LBL “F1”, RCL 01, RCL 02, RCL 03, RCL 04, +, +, +, 16, -, RTN,
LBL “F2”, RCL 01, RCL 02, RCL 03, *, *, RCL 04, 3, *, -, RTN,
LBL “F3”, RCL 01, ST+ X, X^2, RCL 02, RCL 03, RCL 04, *, *, -, 40, -, RTN,
LBL “F4”, RCL 01, RCL 02, RCL 03, RCL 04, *, *, *, 140, -, END
```

We are looking for a solution around W=4, X=1, Y=3, Z=6, so using **LDVW** (with CF 08 and 1.008 as a control word in the X register) or manually, put the following guesses into R01 thru R08:

R01 = 4, R02 = 1, R03 = 3, R04 = 6, R05 = 4.1, R06 = 1.1, R07 = 3.1, R08 = 6.1

Put the following alpha data into R09-R12:

R09 = F1, R10 = F2, R11 = F3, R12 = F4

Now perform 4, STO 0, **NLSEQ**. The following results should show up in R01 thru R04…

W = R01 = 4.266540475, X = R02 = 1.353632234
Y = R03 = 3.548526784, Z = R04 = 6.831300511

XEQ “F1” yields 0.000000010, XEQ “F2” yields 0.000000000, XEQ “F3” yields 0.000000000, and
XEQ “F4” yields 0.000000000.

3.2.13 Means and more Means

The Arithmetic, Geometric, and Harmonic means can be applied to multiple values, not just two as is the restriction of **AGM**, **AGM2**, and **GHM** (these functions just aren’t defined for more than two values and are discussed in the section covering the 2nd FAT functions, specifically section 4.2.3). The **AMEAN**, **GMEAN**, and **HMEAN** programs can take the means of multiple values stored in registers. Entering the control word describing the register set in X and executing **AMEAN**, **GMEAN**, or **HMEAN** will result in that mean being put into X (and the control word saved in LastX). So, for example, to get one of these means for values in registers 10 thru 15, put 10.015 in X and execute the appropriate mean function.

But that’s not all folks! The **PMEAN** program is really a generalized mean function. The power is put into Y and the control word in X, and the Generalized Power Mean is calculated for the values pointed to by the control word. The PMEAN formula is $((x_1\text{^}y + x_2\text{^}y + x_3\text{^}y + \dots + x_n\text{^}y)/n)\text{^}(1/y)$. For y=0 this would normally lead to a problem. However the limit as y -> 0 for this formula yields the Geometric Mean, so when y=0, the **GMEAN** function is substituted.

From the above formula you can see that y=1 yields the Arithmetic mean, and y=-1 yields the Harmonic mean. However fractional and other negative values can be used, and you will notice that as Y becomes infinite (positive), the mean tends to be the MAX value of the numbers. As Y becomes infinite (negative), the mean tends to be the MIN value of the numbers. No, 0.5 will NOT yield the AGM (and -0.5 will NOT yield the GHM) unless, of course, the register values are identical! It is not that simple to get those values, and the power value required changes depending on the two values chosen for AGM or GHM.

With the exception of the **AMEAN** program, all values used in the registers must be ***non-zero positive*** values. Otherwise a DATA ERROR will occur.

AMEAN, **GMEAN**, **HMEAN** and **PMEAN** come from Ángel Martin.

3.2.14 XROM to and from HEX bytes

Sometimes it is nice to be able to translate between XROM idents (##,##) and the FOCAL bytes that represent the XROM function (Ax, xx). Function **HEX2ROM** prompts H"A_"_ _ and expects three hex digits (the first can't be > 7). On successful entry of the 3rd hex digit the corresponding XROM value will be placed into the Alpha register and displayed (in the form _ _ , _ _).

Function **ROM2HEX** does the reverse. It prompts ROM: _ _ , _ _ and expects 4 decimal values (max for the first pair is 31, max for the second pair is 63). On successful entry of the 4th decimal digit the corresponding hex bytes will be placed in the Alpha register and displayed (in the form xx:xx).

If at any time during entry for any of these function the opposite function is desired, pressing the H key will switch to the opposite routine (**ROM2HEX** <> **HEX2ROM**).

Note section 4.2.5 for similar functions **HX2RAPP** and **R2HXAPP** that append to Alpha instead of replace it.

4. 2nd FAT functions

4.1 2nd FAT functions at a glance

This is a list of the functions in the 2nd FAT table (the one you get with GJM# 000 or GJM# 001).

GJM#	Function	Description	Author
000	**-GJM FAT2**	FAT2 Header	Ángel Martin
001	**FCAT**	FAT2 Function Catalog	Ángel Martin
002	**FLOOR**	Floor function	Jean-Marc Baillard
003	**FRC2**	Modified FRC function (X-FLOOR(X))	Jean-Marc Baillard
004	**GHM**	Geometric Harmonic Mean	Greg McClure
005	**MAXR**	Enter Max Real Value (9.999…. E99)	Jean-Marc Baillard
006	**NORM**	Normal of X, Y, and Z	Jean-Marc Baillard
007	**RND0**	Round to nearest integer	Jean-Marc Baillard
008	**XROOT**	Xth Root of Y (picks up sign of X)	Jean-Marc Baillard
009	**YX^**	Xth Power of Y (allows 0^0 = 1)	Jean-Marc Baillard
010	**HX2RAPP**	HEX2ROM, but appends to Alpha instead of repl	Greg McClure
011	**R2HXAPP**	ROM2HEX, but appends to Alpha instead of repl	Greg McClure
012	**PFACT**	Prime Factor	Peter Platzer
013	**PTWIN**	Prime Twin Locator	Peter Platzer
014	**UV**	Internal routine used by 1/M function	Jean-Marc Baillard
015	**AGM**	Arithmetic Geometric Mean	Ángel Martin
016	**AGM2**	Alternate AGM method	Ángel Martin
017	**ELIPE**	2nd Order Elliptic Integral	Ángel Martin
018	**ELIPK**	1st Order Elliptic Integral	Ángel Martin
019	**CURFL**	Return Current Working XMem Filename to Alpha	Sebastian Toelg
020	**STVIEW**	Quick View of Stack	Ángel Martin
021	**PRIME?**	Programmable prime check	Paul Kaarup
022	**NXTPRM**	Find Next Prime	Paul Kaarup
023	**VA**	View Alpha without pausing	Ken Emery

4.2 2nd FAT functions in detail

4.2.1 The 2nd FAT Catalog

To get a complete list of functions in the 2nd FAT, use **FCAT** **(GJM# 001)** or you can also use **-GJM FAT2** **(GJM# 000)**. This is the same routine used by Ángel Martin for the 2nd FAT catalog listing functions in his modules.

4.2.2 Pieces Parts functions

The **FLOOR** **(GJM# 002)** function returns the greatest integer less than or equal to X. This will be INT(X) if X is positive, and INT(X+1) if X is negative. The **FRC2** **(GJM# 003)** is an alternate FRC function, it actually does X-

FLOOR(X). This will not be the same value as FRC(X) if X is a negative non-integer value. These functions come from Jean-Marc Baillard.

4.2.3 Duplex Means

An interesting definition of the mean of two values occurs when combining Arithmetic, Geometric, and Harmonic means. The Arithmetic-Geometric mean is a special value used by elliptical functions, and is defined as limit of A=ArithmeticMean(A,B) and B=GeometricMean(A,B) repeated until A-B = 0. The Geometric-Harmonic mean is defined as the limit of A=GeometricMean(A,B) and B=HarmonicMean(A,B) repeated until A-B = 0.

AGM **(GJM# 015)** calculates the Arithmetic Geometric Mean, while **AGM2** **(GJM# 016)** is an auxiliary routine used for the elliptic routines described below. **GHM** **(GJM# 004)** calculates the Geometric-Harmonic mean. Ángel's **AGM** and **AGM2** routines inspired my **GHM** routine.

As an interesting note, AM(A,B) >= AGM(A,B) >= GM(A,B) >= GHM(A,B) >= HM(A,B).

What happened to the Arithmetic Harmonic mean? That is simply the Geometric mean in disguise. No need for the function.

ELIPE **(GJM# 017)** and **ELIPK** **(GJM# 018)** are the Complete Elliptical integral function examples from Ángel Martin that directly use **AGM** and **AGM2** and are faster than other methods of calculating these integrals, ike the one based on the hypergeometric function as implemented in SandMath (see documentation of this method at http://www2.mae.ufl.edu/~uhk/AGM-2012.pdf). Thus they are included here for comparison.

These functions appear in formulas from numerous fields in math and science, such as calculation of the perimeter of an ellipse, the period of a simple pendulum oscillation, and the mutual inductance between two coaxial coils.

4.2.4 Powers and Roots

The HP41 power function (Y^X) has two difficulties for use: 1) it does not properly handle 0^0, and 2) it does not always yield the negative result for roots. The routines **XROOT** **(GJM# 008)** and **YX^** **(GJM# 009)** were created by Jean-Marc Baillard to overcome these difficulties. **XROOT** will yield the primary root and match the sign to that of the Y argument (which may or may not be what you want or expect, so use accordingly). **YX^** will properly handle 0^0, otherwise it is identical to Y^X.

4.2.5 XROM to and from HEX bytes continued

As mentioned before, sometimes it is nice to be able to translate between XROM idents (##,##) and the FOCAL bytes that represent the XROM function (Ax, xx). Function **HEX2APP** **(GJM# 010)** prompts H"A_"_ _ and expects three hex digits (the first can't be > 7). On successful entry of the 3rd hex digit the corresponding XROM value will be appended to the Alpha register and displayed (in the form _ _ , _ _).

Function **R2HXAPP** **(GJM# 011)** does the reverse. It prompts ROM: _ _ , _ _ and expects 4 decimal values (max for the first pair is 31, max for the second pair is 63). On successful entry of the 4th decimal digit the corresponding hex bytes will be appended to the Alpha register and displayed (in the form xx:xx).

If at any time during entry for any of these function the opposite function is desired, pressing the H key will switch to the opposite routine (**R2HXAPP** <> **HX2RAPP**).

4.2.6 Primes and Prime Twins

A programmable prime factor finder is included, named **PFACT** **(GJM# 012)**. Input is in X, and output is 0 if prime, or the smallest prime factor if composite. Prime twins are defined as two consecutive primes of format (N-1 and N+1). The **PTWIN** **(GJM# 013)** takes one argument (where to start searching) in X, and returns the next N+1 prime twin value. Thanks to Peter Platzer for these gems.

The **PRIME?** **(GJM# 021)** function allows programmable decision making based on the value in X. If it is prime, then one program step will be skipped, otherwise the next instruction is executed. The **NXTPRM** **(GJM# 022)** function returns the next prime found after the value in X. These routines come from Paul Kaarup.

4.2.7 View Alpha function

The **VA** function is really nothing more than an AVIEW that will allow a program to continue regardless of the setting of any flags. Ken Emery is the author of this program.

4.2.8 Miscellaneous functions

These quicky functions don't fall into a common category with anything, so here are the remaining functions in the 2nd FAT:

MAXR **(GJM# 005)** quickly enters the maximum possible real value into register X (9.99… E99). (Jean-Marc Baillard)

NORM **(GJM# 006)** performs the function SQRT(X^2 + Y^2 + Z^2). (Jean-Marc Baillard)

RND0 **(GJM# 007)** rounds the value in X to the nearest integer. (Jean-Marc Baillard)

UV **(GJM# 014)** is actually included as an internal routine for the 1/M function. It solves Bezout's Identity (for A in Z , B in Y, and C in X, i.e. it solves Au+Bv=C where C is a multiple of GCD(a,b), results in v in X and u in Y, and GCD(a,b) in Z. (Jean-Marc Baillard)

CURFL **(GJM# 019)** is a neat function that returns the name of the Current Working File (last one pointed to in XMEM) into the Alpha register. This routine is used by the LDVWD program as an example! (Sebastian Toelg)

And finally **STVIEW** **(GJM# 020)** quickly reviews the stack contents (pausing briefly between each value of T, Z, Y, X, and L). (Ángel Martin)

Appendices

Appendix A: Slide Rule program usage

A.1 SR Overview

This program is designed to emulate most common slide rules (including log log, duplex, trig). User flags 0 thru 6 are used, along with registers 00 thru 03. The emulator automatically handles decimal point and sign for values and calculates using standard accuracy for all functions, in effect extending the normal slide rule capabilities. The scales extend to either side to the range and accuracy of the values allowed by the HP-41.

A.2 SR Scale Specifics

It is assumed that the user has a basic understanding of the use of a slide rule. This emulation has three modes:

- HL > SCALE mode: Emulates moving the hairline to the position on the scale specified (indicated by flag 0 clear when entering value/scale).
- SCALE > HL mode: Emulates moving the inner scale position under the hairline (indicated by flag 0 set when entering value/scale).
- READ mode: Emulates reading the value of the scale under the hairline (indicated by flag 1 set when entering the scale to use).

There are four major scales used: A, B, C, and D. The main scales are based on I and F scale modifications to these four, making a total of 20 main scales: A, AI, AF, AIF, AFI, B, BI, BF, BIF, BFI, C, CI, CF, CIF, CFI, D, DI, DF, DIF, DFI. I don't believe I have ever seen the FI-style scales on any slide rules, but they are included for sake of completeness (for examples, CIF is the C scale inverted, then divided by PI, CFI is multiplied by PI, then inverted).

There are 15 subscales, any one may be optionally used on any of the 20 main scales. On a log log duplex trig slide rule, for example, the LL scales are based on the D scale; on this emulator it is thus called the DLL scale. The 15 subscales are:

- P -- Pythagorean scale, which is SQRT(1-X^2)
- Q -- Cube Root scale
- K -- Cube scale
- W -- Square Root scale
- R -- Square scale
- U -- User function
- L -- Log scale (base 10)
- LN -- Log scale (base e)

- CIR -- Circumference scale (Pi * (X*X/4)
- S -- Sine scale
- CC -- Cosine scale (named CC so that it won't be confused with LBL C)
- T -- Tangent scale
- LL -- Log Log scales (base e)
- LLD -- Log Log scales (base 10)
- LLM -- Log Log (base e, exponent - 1)

With 20 main scales, and 16 combinations each (subscale is optional), this slide rule emulator emulates 320 possible scales.

A.3 SR Flag Usage

User flags 0 thru 6 are used by the SR program…

Flag	When clear means…	When set means…
0	Mode HL>SCALE (except if flag 1 set)	Mode SCALE>HL (except if flag 1 set)
1	Flag 00 invalid	Mode READ
2	I modifier off	I modifier on
3	F modifier off	F modifier on
4	IF sequence (flags 2 and 3 will be set)	FI sequence (flags 2 and 3 will be set)
5	C or D scale	A or B scale
6	A or D based scales (outer scales)	B or C based scale (inner scales)

A.4 SR Register Usage

Registers 0 thru 3 are used by the SR program…

Register	Usage
0	Position of hairline on the D scale
1	Offset of C scale to the D scale
2	Saved value on scale set
3	Alpha name of user defined scale

A.5 SR User Keyboard

A SCALE	B SCALE	C SCALE	D SCALE	RESET
F-type SCALE	HL>SCALE MODE	SCALE>HL MODE	I-type SCALE	READ MODE

A.6 SR Subscale Keyboard

Key Col 1	Key Col 2	Key Col 3	Key Col 4	Key Col 5	Use
P	Q	W	L	LN	**XEQ 01 thru 05**
CIR	U	S	C	T	**XEQ 06 thru 10**
LLM	K	R	LLD	LL	**Gold a to e**

A.7 SR User Instructions

After loading the program, XEQ "SR" (which will automatically turn on USER mode). Any time after the program has started, XEQ E will also reset and restart the program. If USER mode is on, execution of the program is much easier (Labels A thru J are available with one keystroke, a thru e are available with two).

To set a value (assumes USER mode is on):

- If the mode is not correct, press G to setup HL>SCALE mode, or H to setup SCALE>HL mode.
- Enter the number, then press either A, B, C, or D for the major scale.
- If desired press I and/or F in the order needed.
- If no subscale is required, press R/S. The scale name and value appear in the display, otherwise…
    - If a subscale is required, XEQ 01 - 10, or press Gold a - e to choose the subscale needed (01=P, 02=Q, 03=W, 04=L, 05=LN, 06=CIR, 07=U, 08=S, 09=C, 10=T, a=LLM, b=K, c=R, d=LLD, e=LL).
    - The scale:subscale name and value will appear in the display (no R/S is needed).
- After displaying the scale:subscale selected and showing the value, the emulator automatically switches modes (toggles between HL>SCALE mode and SCALE>HL mode)

To read any scale:subscale (assumes USER mode is on):

- Press J, then either A, B, C, or D for the major scale.
- If desired press I and/or F in the order needed.
- If no subscale is required, press R/S. The scale name and read value appear in the display, otherwise
    - If a subscale is required, XEQ 01 - 10, or Gold a - e to choose the subscale needed (01=P, 02=Q, 03=W, 04=L, 05=LN, 06=CIR, 07=U, 08=S, 09=C, 10=T, a=LLM, b=K, c=R, d=LLD, e=LL).
    - The scale:subscale name and read value will appear in the display (no R/S is needed).
- HL>SCALE mode is automatically set.

When first reset, the program is in HL>SCALE mode.

Here is the template for user defined functions for use with the U subscale:

```
LBL “SS” ; substitute name of subscale for SS (6 letters or less)
CF? 01   ; Not reading?
GTO 00   ; Go to set scale logic
…        ; Enter routine used to read the scale
         ; (for the S scale, this would be the SIN function)
RTN
LBL 00   ; Set logic
…        ; Enter routine used to set the scale
         ; (for the S scale, this would be the ASIN function)
END
```

The emulator automatically sets flag 25 before calling the subscale routine, then checks for out of range condition on return. The name of the subscale program should be put in Reg 03 (by using the ALPHA register and doing ASTO 03). Then choose the U scale (XEQ 07). The subscale program should use the X register for its value and return the answer in X register. It should not alter Regs 0 thru 3, but may fully use any other register.

The following subscale would allow a SINH scale for U…

```
LBL “SINH”
FC? 01 ; Not read scale?
GTO 00 ; Go to set logic
E^X    ; Read is Hyperbolic Sine
ENTER
1/X
-
2
/
RTN    ; Return
LBL 00 ; Set function
ENTER  ; Set is Hyperbolic ArcSine
X^2
1
+
SQRT
+
LN
END    ; Return and end
```

A.8 SR Examples

Calculate PI*3*5*7 (example assumes FIX 4, keypress assumes USER mode on):

Enter	Keypress	Comments	Output
	E	Resets SR emulator Ennun 0 is off (HL>SCALE mode)	“READY”
3	D	Move hairline to 3 on the D scale	“D”
	R/S	No subscale, complete action Ennun 0 is now on (SCALE>HL mode)	“D 3.0000”
5	C	Move 5 on the CI scale under hairline	“C”
	I	Ennun 2 is now on (I scale)	“CI”
	R/S	No Subscale, complete action Ennun 0 is off (HL>SCALE mode)	“CI 5.0000”
7	C	Move hairline to 7 on C scale Ennun 2 turned off	“C”
	R/S	No Subscale, complete action Ennun 0 is on (SCALE>HL mode) Usually you want 0 on to read	“C 7.0000”
	J	Prepare to read scale Ennun 1 is on (READ mode) Ennun 0 is off	“READ SCALE”
	D	Want to read DF scale	“D”
	F	Ennun 3 is on (F scale)	“DF”
	R/S	No subscale, complete read	“DF 329.8672”

Determine the previous result raised to the 3rd power:

Enter	Keypress	Comments	Output
	J	Prepare to read scale Ennun 1 is on (READ mode) Ennun 0 is off	“READ SCALE”
	D, F	Choose the DF scale	“DF”
	B	K subscale (no need for R/S)	“DFK 35893641.07”

Calculate the ASIN (0.5):

Enter	Keypress	Comments	Output
	E	Reset (Hairline to 1, scales aligned)	“READY”
.5	D, R/S	Move hairline to 0.5 on the D scale	“D 0.5000”
	J	Read the S subscale of D	“READ SCALE”
	D		“D”
	XEQ 08		“DS 30.0000”

Calculate ASINH(3*Pi) (assumes that the subscale program SINH above is in memory):

Enter	Keypress	Comments	Output
	E	Reset	“READY”
“SINH”	ASTO 03	Store name of routine in register 03	“SINH”
3	D, R/S	Move hairline to 3 on the D scale	“D 3.0000”
	J	Read the U subscale off DF	“READ SCALE”
	D, F		“DF”
	XEQ 07		“DFU 6195.8269”

Appendix B: Algebraic Operating System (AOS) program usage

B.1 AOS Overview

The Algebraic Operating System emulator is designed to act like nonRPN calculators that use parenthesis and pending operations to solve numeric math operations. This program requires an Extended memory file (name AOS) to store data for pending operations for parenthesis operation. The program does not require any other memory except for the stack (which is fully used).

B.2 AOS Flag Usage

Flag	Use when set
0	+ pending (flag 1 MUST be clear)
1	- pending (flag 0 MUST be clear)
2	* pending (flag 3 MUST be clear)
3	/ pending (flag 2 MUST be clear)
4	^ pending
5	Open ('s pending

B.3 AOS User Keyboard

AOS +	AOS -	AOS *	AOS /	AOS ^
AOS (	AOS)			AOS = (R/S)

B.4 AOS User Instructions

After XEQ "AOS" the AOS flags and AOS buffer will initialize. It will ask for the size of the Extended Memory file to use. If the AOS Data file already exists, it will ask for the new size. If no new size is given the data file is not resized. User mode will be enabled.

B.5 AOS Example

Usage of the AOS program is best served by a simple example.

Calculate (1+2) * (3/4) + (5^(1/2))

Enter	Keypress	Comments (and Ennun.s)	Ennunciators (red = on)	Output
	XEQ “AOS”	Reset AOS	01234	“SIZE?” (if no file) “NEW SIZE?” (if file)
20	R/S	Small array		0.0000
	F	(		0.0000
1	A	1 +	**0**1234	1.0000
2	G	2), + performed	01234	3.0000
	C	*	01**2**34	3.0000
	F	(, * with value saved	01234	3.0000
3	D	3 /	012**3**4	3.0000
4	G	4), / performed, * with value recalled	01**2**34	0.7500
	A	+, * performed	**0**1234	2.2500
	F	(	01234	2.2500
5	E	5 ^	0123**4**	5.0000
	F	(, ^ with value saved	01234	5.0000
1	D	1 /	012**3**4	1.0000
2	G	2), / performed, ^ with value recalled	0123**4**	0.5000
	G	), ^ performed, + with value recalled	**0**1234	2.2361
	J or R/S	= final + performed	01234	4.4861

In this example, after entering the final 2, instead of using G the final answer could have been calculated by entering J or R/S (J or R/S will perform all pending parenthesis and functions).

For those interested, the data file saves required values from the stack and the status of the flags every time the AOS (function is performed. It restores the flags and data values required back to the stack when AOS) is performed. The enunciators show which operations and how many stack registers will be stored (only one register is required for the operations saved).

Appendix C: Cubic Spline program usage

C.1 CSPLINE Overview

The cubic spline algorithm is a mathematical interpolation method used to create a smooth curve between points on a graph. At least four data points are required to create a cubic spline. These data points need to be placed into a data array in Extended Memory. The user can decide the name of this array. After creating this array and filling it with X,Y data pairs, executing CSPLINE will create the splines needed to graph the curve. The program requires Matrix functions to create the solution, so either the Advantage or SandMatrix module is required for this program. If using the SandMatrix module, the SandMath module will also be required.

To solve the cubic spline, first a tri-diagonal matrix needs to be created, and a solution vector. “*ABC” and “*R” matrices are temporarily created for this purpose. Once solved, the “*MN” data array is created with the cubic spline coefficients, and “*ABC” and “*R” are removed. “*MN” is used to display the interpolated Y and YPRIME for any X entered.

C.2 Usage of LDVWD

Program LDVWD has been created to enter/review the data points in the array. First the data array must be created. For a 4 point solution the size must be 8, so with the name of the array in Alpha, and 8 in X, use CRFLD (use PURFL if already created). If user flag 8 is set, LDWD will only view the data in the array. If user flag 8 is clear, it will show and optionally allow modification of the data in each element. If entering data, just enter each element until done.

C.3 CSPLINE Usage by Example

It is easiest to show how to use CSPLINE by giving an example.

Let’s say we want to create a smooth curve going thru points [0,0], [1,1], [2,9], and [3,10]. We wish to use the “natural” slope of the curve at both the beginning and the end (it can optionally be specified for either end).

Let’s create data array “XY” (needs to be size 8). So “XY”, 8, CRFLD. Now insure flag 8 is off and XEQ “LDVWD”. Enter in points 0, 0, 1, 1, 2, 9, 3, and 10 at the prompts. We are now ready to perform the spline interpolation.

XEQ “CSPLINE”. It asks for the array name, enter XY and press R/S, it asks if we want initial slope. No response at this prompt forces “natural” slope for the initial point. R/S then asks for the the final slope. No response forces “natural” slope for the final point. R/S then displays the steps as it calculates the arrays, then solves the simultaneous equations created.

We are ready to display results for points. LBL A is always available to quickly get to this point. After the solution is created, we are at this label. R/S prompts for “NEXT X?”, enter the X value we want Y value for. R/S calculates the Y coordinate interpolated, and R/S again calculates the slope at that point. In this example, let’s get the slope calculated for the first point [0,0]. Enter 0, R/S, and it shows Y is 0 (not a surprise), R/S gives a slope of -1.3333. R/S (or A) and get "NEXT X?", 1, R/S and it shows Y is 1 (not a surprise), R/S gives a slope of 5.6667. X = 1.5 gives Y=5, and slope of 9.1667, and so on for any other points you want to solve. The natural slope at the end point of 3 was also -1.3333.

We can rerun the program and specify initial and final slopes if we wish, try it and see what points and slopes are interpolated with init and final slopes of 0.