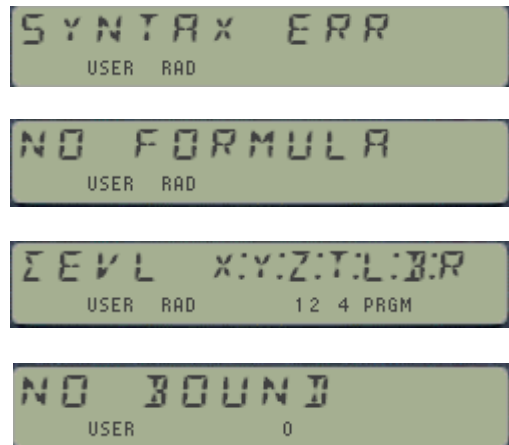
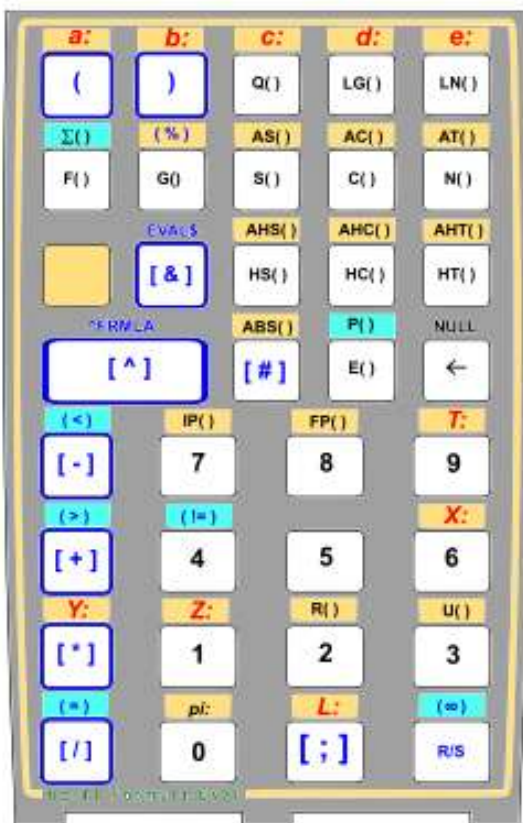
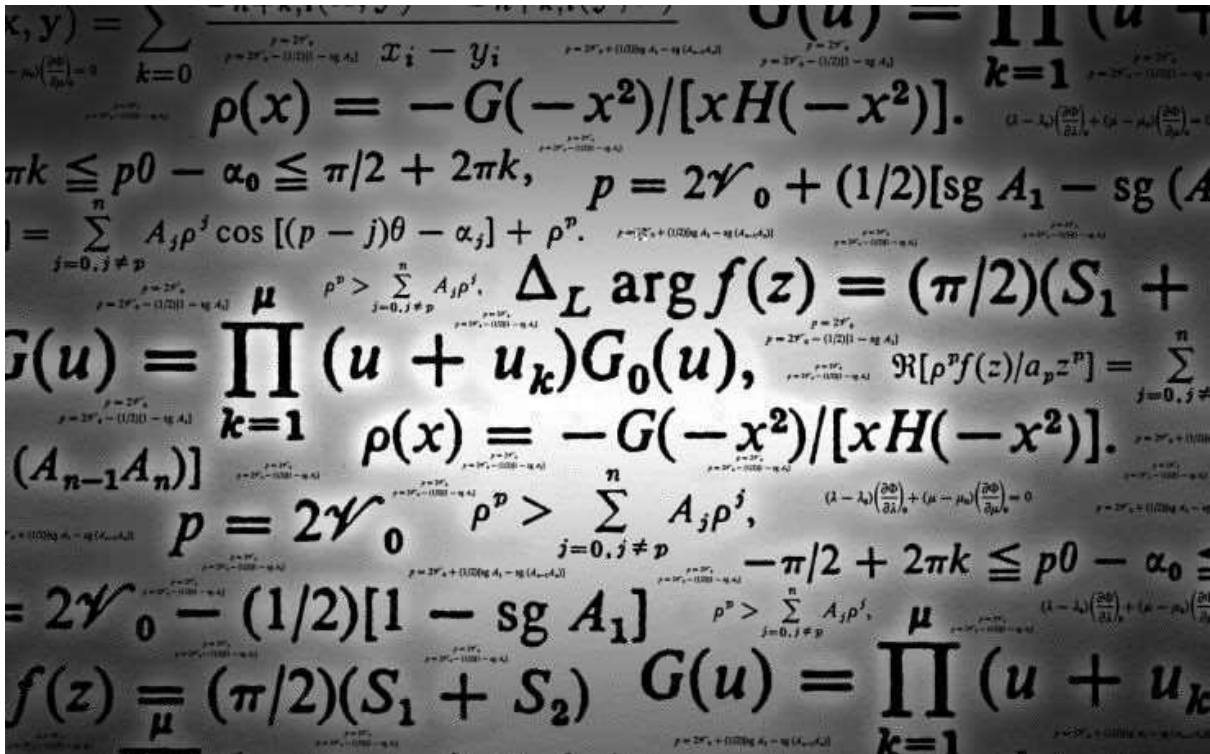


# FORMULA EVALUATION ROM

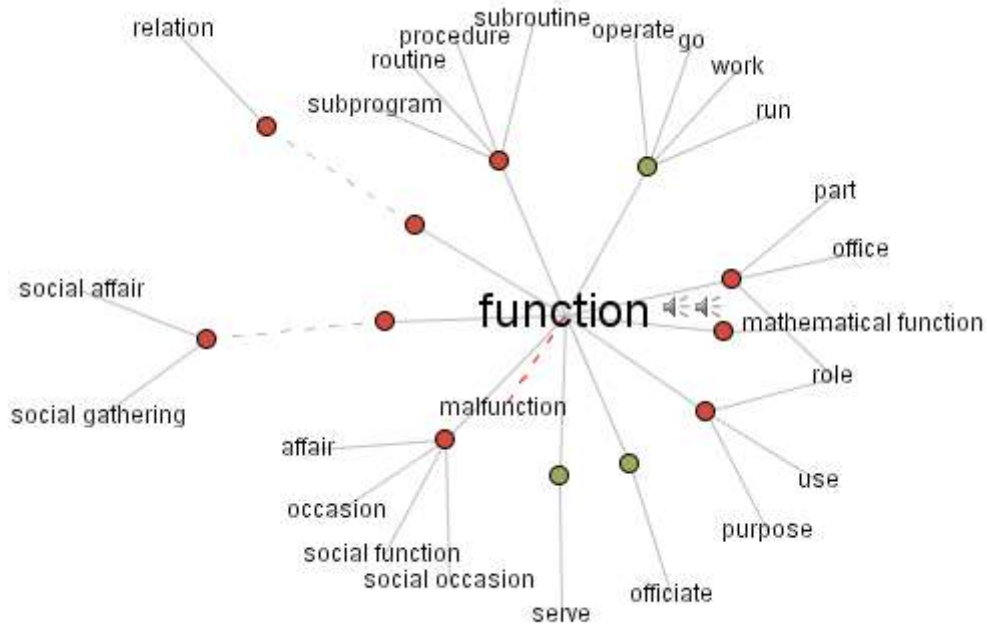
## HP-41 Module



Written & Programmed by  
Greg McClure and Ángel Martín  
Revision 3K+, June 2022

This compilation revision 1.6.6

Copyright © 2017-2022 Ángel Martín & Greg McClure



Published under the GNU software license agreement.

Original authors retain all copyrights and should be mentioned in writing by any part utilizing this material. No commercial usage of any kind is allowed.

Thanks to Mark Fleming for his through revisions to the manuals and suggesting numerous enhancements to the ROM.

Screen captures taken from V41, Windows-based emulator developed by Warren Furlow.  
See [www.hp41.org](http://www.hp41.org)

# FORMULA\_EVALUATION - 3K++

## HP-41 Module

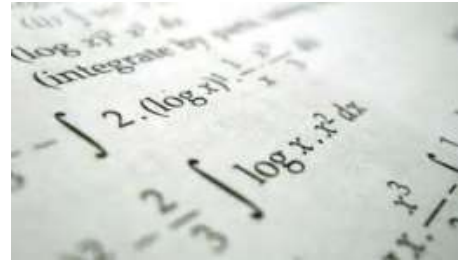
### Table of Contents

1. What's New in the 2018/19 revisions	
a. <a href="#">Teaching new tricks to an ol' dog</a> . . . . .	4
b. <a href="#">Scope, Intent and Dependencies</a> . . . . .	4
c. <a href="#">Module function Summary</a> . . . . .	5
2. Syntax and Rules of Engagement	
a. <a href="#">Variables, constants and parameters</a> . . . . .	8
b. <a href="#">Formula Entry remarks</a> . . . . .	8
c. <a href="#">Formula Evaluation rules</a> . . . . .	9
d. <a href="#">Syntax Table and keyboard Overlay</a> . . . . .	10
e. <a href="#">Chained evaluations &amp; Error handling</a> . . . . .	13
f. <a href="#">EVAL Launcher and Alpha to Memory</a> . . . . .	14
g. <a href="#">Other Utility functions</a> . . . . .	16
3. Example Programs	
a. <a href="#">Vector distances and Dot product</a> . . . . .	18
b. <a href="#">Polynomial Evaluation using Honer's method</a> . . . . .	19
c. <a href="#">Orthogonal Polynomials: Legendre, Hermite, and Chebyshev's</a> . . . . .	20
d. <a href="#">Real Roots of Quadratic Equation</a> . . . . .	21
e. <a href="#">Solve and Integrate Reloaded</a> . . . . .	22
f. <a href="#">Use of EVAL\$ with FINTG and FROOT</a> . . . . .	23
g. <a href="#">Lambert Function</a> . . . . .	24
4. EVAL\$ Advanced Applications	
a. <a href="#">Advanced test comparisons with EVAL?</a> . . . . .	25
b. <a href="#">WHILE we're at it: Putting EVAL? to work</a> . . . . .	26
c. <a href="#">What IF ?; Getting EVAL? money's worth</a> . . . . .	30
d. <a href="#">Even more difficult: FOR...NEXT loops</a> . . . . .	34
e. <a href="#">Evaluating Sums &amp; Series with EVALΣ</a> . . . . .	36
f. <a href="#">Evaluating Products with EVALΠ</a> . . . . .	37
g. <a href="#">Examples: Gamma and Digamma functions</a> . . . . .	38
h. <a href="#">Scripting Language facility using X-Mem</a> . . . . .	41
<a href="#">Appendix 1. Sub-functions in the auxiliary FAT</a> . . . . .	47
<a href="#">Appendix 2. Eval\$ Buffer Structure</a> . . . . .	48
<a href="#">Appendix 3. EVAL Applications ROM</a> . . . . .	50
<a href="#">Appendix 4. Underpinnings of DO-WHILE</a> . . . . .	56

## Formula Evaluation ROM

Revision 3K++

HP-41 Module



### *Introduction. Teaching new tricks to an old dog.*

Welcome to the Formula Evaluation ROM, a plug-in module for the HP-41 platform that allows you to evaluate formulas typed in the ALPHA registers directly – without the need for RPN programs.

It is generally accepted that Symbolic Algebra and CAS are well beyond the scope of a venerable machine like the HP-41, quickly approaching 40-year old architecture and design. Some pioneering attempts were made in the old days, but their practical applicability (and very slow performance) would render them into little more than exploratory incursions into the field.

Fast-forward to the present with PC emulators and SY's 41-CL boards capable of TURBO speed – add to that the stubborn dedication of MCODE programmers refusing to accept defeat, and the results are interesting projects that push the limits of the original designs, like this one.

### Scope, Intent and Dependencies

The core of the routines is based on Greg McClure's idea for the design of the Symbolic Buffer – a dedicated structure in the I/O memory area capable to store unformatted data, and therefore suitable for abstract constructs like operations, function codes, and of course variable values. Wrapped around that core is a set of functions that allow the user to input formulas in a convenient way, save them in and recall them from data registers, and evaluate the results.

The initial design had very modest goals but was soon enough extended to include a comprehensive set of functions and operations, only restricted by the inherent limitations of the LCD display, the keyboard and other design aspects. Also remember that supporting all math are the 13-digit OS routines doing the number crunching.

**New:** Revision 1G added to the mix an intriguing set of functions for a *higher-level programming experience*: both **DO/WHILE loops** and **IF/ELSE/ENDIF groups** are available as direct applications of the underlying **EVAL\$** and **EVAL?** functions of the module.

This is not an AOS Module – even if you're already making that connection in your mind. If anything, it'll be more akin to the CALC mode on the HP-71, albeit with the obvious huge differences in power and flexibility. The Formula Evaluation concept is also somewhat similar to the AECROM's Self-Programming facility, which also uses the ALPHA register to enter the definition formula. However, with the Evaluation functions there are no FOCAL programs involved to calculate the results.

Note that the **EVAL\$** functions are programmable and can be used directly, replacing calls to FOCAL subroutines (typically made using "XEQ IND Rnn" with the ALPHA name stored in Rnn). In fact, this module includes versions of SOLVE and INTEG programs using **EVAL\$** directly.

As for dependencies, this module is a **Library#4-aware ROM** that requires the library#4 to be plugged in. Also, the ROM is only compatible with the CX OS, as internal routines from it are used.

## Formula Evaluation ROM – Function Summary

The table below lists all functions available in the module. The Main FAT section comprises 36 functions, while the Auxiliary FAT section adds another set of 29 functions. All of them are programmable and directly accessible by the user.

#	Name	Description	Input	Author
00	<b>-FORM EVAL+</b>	Section header	n/a	n/a
01	<b>^FRMLA _</b>	Enters Formula in ALPHA	Uses Custom Keyboard	Ángel Martin
02	<b>EVAL\$</b>	Evaluates Formula -> X	Expression in ALPHA	Greg McClure
03	<b>EVALY</b>	Evaluates Formula ->Y	Expression in ALPHA	Martin-McClure
04	<b>EVALZ</b>	Evaluates Formula ->Z	Expression in ALPHA	Martin-McClure
05	<b>EVALT</b>	Evaluates Formula ->T	Expression in ALPHA	Martin-McClure
06	<b>EVALR _ _ _</b>	Evaluates -> Data Register	Expression in ALPHA	Martin-McClure
07	<b>GET= _</b>	Recalls parameter value	a,b,c,d,e in prompt	Ángel Martin
08	<b>LET= _</b>	Sets Parameter Value	a,b,c,d,e in prompt	Ángel Martin
09	<b>SHOW= _</b>	Shows Parameter Value	a,b,c,d,e in prompt	Ángel Martin
10	<b>SWAP= _</b>	Swaps Parameter and X	a,b,c,d,e in prompt	Ángel Martin
11	<b>SF# _ _ _</b>	Sub-function by index	sub-fnc. Index#	Ángel Martin
12	<b>SF\$ _</b>	Sub-function by Name	sub-fnc. Name	Ángel Martin
13	<b>\$KY?N</b>	Bulk Key Assignments	Prompts Y/N, Cancel	HP Co.
14	<b>ΣEVL _</b>	Eval Launcher	Prompts for destination	Ángel Martin
15	<b>-EVAL\$ FNS</b>	Section header	n/a	n/a
16	<b>RCL\$ _ _</b>	Recalls Formula to ALPHA	Prompts for Rg#	Ángel Martin
17	<b>RG&gt;ST _ _</b>	Registers to Stack	Prompts for Re#	Ángel Martin
18	<b>SHFL _ _ _ _</b>	Shuffles Stack Registers	Prompts for Stk. Order	Ángel Martin
19	<b>ST&gt;RG _ _</b>	Stack to Regs	Prompts for Rg#	Ken Emery
20	<b>STO\$ _ _</b>	Stores Formula in Memory	Prompts for Rg#	Ángel Martin
21	<b>SWAP\$ _ _</b>	Swaps Alpha and Regs	prompts for Rg#	Ángel Martin
22	<b>"EVAL?"</b>	Evaluates Boolean Tests	Expressions in ALPHA	Ángel Martin
23	<b>EVALΣ</b>	Sums and Series	Expression in ALPHA	Ángel Martin
24	<b>EVALP</b>	Products	Expression in ALPHA	Ángel Martin
25	<b>LEFT\$</b>	Extracts Left text	#Chars in X	Ross Colling
26	<b>RIGHT\$</b>	Extracts right text	#Chars in X	Ross Colling
27	<b>A-PM</b>	ALPHA to Program Memory	String in ALPHA	Ángel Martin
28	<b>DO</b>	Begins While Loop	WHILE statement below	Ángel Martin
29	<b>WHILE</b>	Ends While Loop	Expression in ALPHA	Ángel Martin
30	<b>IF</b>	Begins IF group	Expression in ALPHA	Ángel Martin
31	<b>ELSE</b>	Branches IF	ENDIF statement below	Ángel Martin
32	<b>ENDIF</b>	Ends IF group	none	Ángel Martin
33	<b>FOR</b>	Begins For/Next loop	Bbb.eee in X	Ángel Martin
34	<b>NEXT</b>	Ends For/Next loop	none	Ángel Martin
35	<b>SWAP\$ _ _</b>	Swap ALPHA and Regs	Reg# in prompt	Ángel Martin
36	<b>ST&lt;&gt;RG _ _</b>	Swaps Stack and Regs	Reg# in Prompt	Ken Emery
0	<b>-TST FNS</b>	Section header	n/a	n/a
1	<b>B6?</b>	Buffer #6 Check	Data in I/O	Greg McClure
2	<b>B7?</b>	Buffer #7 Check	Data in I/O	Ángel Martin
3	<b>EVALb _</b>	Evaluates -> Buffer Register	Expression in ALPHA	Ángel Martin
4	<b>EVALL</b>	Evaluates Formula -> L	Expression in ALPHA	Martin-McClure
5	<b>EVAL#</b>	EVAL by index	Index in R00	Greg McClure
6	<b>LADEL</b>	Left ALPHA delete	Text in ALPHA	Ross Colling
7	<b>RADEL</b>	Right ALPHA delete	Text in ALPHA	Ross Colling
8	<b>TRIAGE</b>	Variable assignment	ASCCI file record	Martin-McClure
9	<b>WORKFL</b>	Current File Name	Appended to ALPHA	Sebastian Toleg
10	<b>CLRB6</b>	Clear Buffer#6	Buffer#6 in Memory	Greg McClure



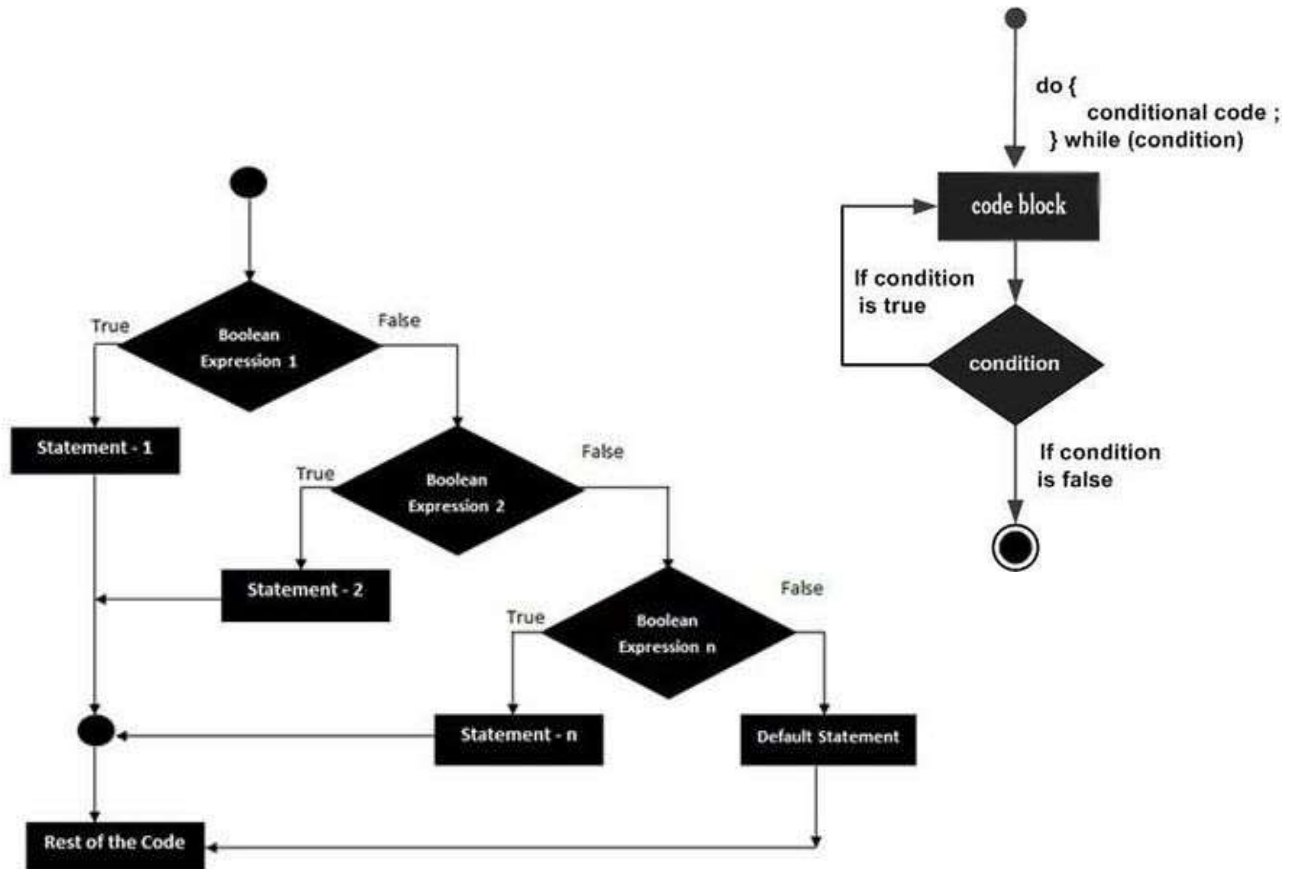
#	Name	Description	Input	Author
11	<b>CHK\$</b>	Checks Syntax	Expression in ALPHA	Ángel Martin
12	<b>TST\$</b>	Test ALPHA operators	{ "!=", "=", "<", ">" }	Ángel Martin
13	<b>PSHB6</b>	Push X to Buffer#6	Data in X	Greg McClure
14	<b>POPB6</b>	Pop data from Buffer#6	Data in buffer reg	Greg McClure
15	<b>NXTCHR</b>	Get Next Char	Text in ALPHA	Greg McClure
16	<b>PRVCHR</b>	Get Previous Char	Text in ALPHA	Greg McClure
17	<b>B7&gt;ST</b>	Copies buffer to Stack	None	Ángel Martin
18	<b>ST&gt;B7</b>	Copies Stack to Buffer	None	Ángel Martin
19	<b>BLIP</b>	Make a Sound	None	Ángel Martin
20	<b>DTOA</b>	Display to Alpha	LCD text	Ángel Martin
21	<b>DTST</b>	Display Test	none	Chris Dennis
22	<b>ΣDGT</b>	Sum of Mantissa Digits	Number in X	Ángel Martin
23	<b>ZOUT</b>	Shows Complex value	Re in X, Im in Y	Ángel Martin
24	<b>CAT+ _</b>	Sub-function CATalog	R/S, SST, BST. XEQ	Ángel Martin
25	<b>XQ&gt;GO</b>	Pops the first RTN addr	Skips the 1 <sup>st</sup> . return	Håkan Thörngren
26	<b>KRTN2</b>	Kills 2 <sup>nd</sup> . RTN addr	Skips the 2 <sup>nd</sup> . Return	Ángel Martin
27	<b>?RTN</b>	Tests for pending RTN	Skips next line if False	Doug Wilder
28	<b>RTNS</b>	Number of pending RTN	Data in RTN stack	Ángel Martin
29	<b>FILL</b>	Fills Stack w/ X-value	value in X	J.D. Dodin

Additionally, the module comes with a library of pre-programmed applications, as follows:

00	<b>-EVAL APPS</b>			
01	<b>AIN\$</b>	ALPHA integer part	Value in X	Fritz Ferwerda
02	<b>"ARPLY"</b>	Alpha Replace Y by X	Old in Y, new in X	Greg McClure
03	<b>"IT\$"</b>	Integrates		
04	<b>"SV\$"</b>	Solves f(x)=0	Guess in X	PPC Members
05	<b>"AGM"</b>	Arithm-Geom. Mean	x, y in X, Y	Ángel Martin
06	<b>"d2\$"</b>	2D-Distance	P1, P2 in Stack	Martin-McClure
07	<b>"d3\$"</b>	3D-Distance	Prompts for Vectors	Martin-McClure
08	<b>"DOT\$"</b>	Dot Product 3x3	Prompts for Vectors	Martin-McClure
09	<b>"CL\$"</b>	Ceiling Function	Argument in X	Ángel Martin
10	<b>"FL\$"</b>	Floor Function	Argument in X	Ángel Martin
11	<b>"HRON\$"</b>	Triangle Area (Heron)	A, b, c in Y,Z,T	Angel Martin
12	<b>"LINE\$"</b>	Line equation thru points	Y2,X2,Y1,X1 in Stack	Angel Martin
13	<b>"NDF\$"</b>	Normal Density Function	μ in Z, σ in Y, x in X	Ángel Martin
14	<b>"P4\$"</b>	Polynomial Evaluation	Prompts for Coefficients	Ángel Martin
15	<b>"QRT\$"</b>	Quadratic Equation Roots	Coefficients in Z, Y, X	Martin-McClure
16	<b>"RS\$"</b>	Rectangular to Spherical	{x, y, z} in X, Y, Z	Ángel Martin
17	<b>"SSR\$"</b>	Spherical to Rectangular	{R, phi, theta} in X, Y, Z	Ángel Martin
18	<b>-\$AND MTH</b>	<b>Section header</b>	<b>n/a</b>	<b>n/a</b>
19	<b>"KK\$"</b>	Elliptic Integral 1 <sup>st</sup> . Kind	argument in X	Ángel Martin
20	<b>"NCK\$"</b>	Combinations	n in Y, k in X	Ángel Martin
21	<b>"NPK\$"</b>	Permutations	n in Y, k in X	Ángel Martin
22	<b>"LEG\$"</b>	Legendre Polynomials	order in Y, argument in X	Ángel Martin
23	<b>"HMT\$"</b>	Hermite's Polynomials	order in Y, argument in X	Ángel Martin
24	<b>"TNX\$"</b>	Chebyshev's Pol. 1 <sup>st</sup> . Kind	order in Y, argument in X	Ángel Martin
25	<b>"UNX\$"</b>	Chebyshev's Pol. 2 <sup>nd</sup> . Kind	order in Y, argument in X	Ángel Martin
26	<b>"e^X"</b>	Exponential function	Argument in X	Ángel Martin
27	<b>"ERDO\$"</b>	Erdos-Borwein constant	None	Ángel Martin
28	<b>"FHB\$"</b>	Generalized Faulhaber's	N in Y, x in X	Ángel Martin
29	<b>"HRM\$"</b>	Harmonic Number	N in X	Ángel Martin
30	<b>"GAM\$"</b>	Gamma function (Lanczos)	Argument in X	Ángel Martin
31	<b>"JNX\$"</b>	Bessel J integer order	n in Y, x in X	Ángel Martin

#	Name	Description	Input	Author
32	"LNG\$"	LogGamma	Argument in X	Ángel Martin
33	"PSI\$"	Digamma function	Argument in X	Ángel Martin
34	"WL\$"	Lambert W Function	Argument in X	Ángel Martin
35	"ERF\$"	Error Function	Argument in X	Ángel Martin
36	"CI\$"	Cosine integral	Argument in X	Ángel Martin
37	"SI\$"	Sine Integral	Argument in X	Ángel Martin
38	"JDN\$"	Julian Day Number	MDY Date in {Z,Y,X}	Ángel Martin
39	"CAL\$"	Calendar Date	JND in X	Ángel Martin
40	-SCRIPT EVL	Section Header	n/a	n/a
41	"EVALXM"	Evaluates an XM File	ASCII File Script	Greg McClure
42	"EVLXM+"	Executes Script File	File Name in ALPHA	Greg McClure
43	1ST	1 <sup>st</sup> . Position	Program usage	Greg McClure
44	2ND	2 <sup>nd</sup> Position	Program usage	Greg McClure
45	3RD	3 <sup>rd</sup> Position	Program usage	Greg McClure
46	4TH	4 <sup>th</sup> Position	Program usage	Greg McClure
47	"EVLΣ+"	Enhanced EVALΣ	Formula in ALPHA	Martin-McClure
48	"EVLΠ+"	Enhanced EVALΠ	Formula in ALPHA	Martin-McClure
49	"GMXM"	Makes GAMMA Script	none	Martin-McClure
50	^01	Puts chars in R00-R01	String in ALPHA	Martin-McClure
51	+REC	Advance File Record	Selected XM File	Martin-McClure
52	"FCT#"	Factorial using Do/While	Argument in X	Ángel Martin
53	"FIB#"	Fibonacci using Do/While	Argument in X	Ángel Martin

From low-level routines to the keyboard overlay, a lot of work went into making the Formula Evaluation ROM. Much of it is transparent to the user, but it all plays an important role when it comes to the moment to put it to a good use. We hope you find the module useful and enjoy using it as much as we have enjoyed writing it !



## *Syntax and Rules of Engagement.* { **^FRMLA**, **EVAL\$** }

---

Syntax rules always come together with this kind of functionality by definition: the formulas must abide with the expected forms and formats for the Evaluation engine to decode them properly.

Obviously, power users can use a free-form manual typing in ALPHA (which requires access to curved parenthesis and other special characters, as provided by the AMC\_OS/X Module) – but a much more convenient approach is to use the **^FRMLA** facility that chooses the right mnemonics for the functions and assists with the editing.

Here the 41-LCD limited length and modest character set force some compromises for practical and effective rules, still meaningful enough to be unambiguous and easily recognized by the user. A good balance between those two is the ultimate goal of every design.

Conceptually speaking, formulas are expressions that contain references to three components: Data, Operators, and Functions. The data is further sub-divided in *variables*, *parameters*, and *constants*. These are expected to be in the following arrangement:

- Variables are the stack registers contents, and are referenced by the corresponding register letter {XYZTL}. ALPHA DATA contents are not allowed.
- Constants are explicit **integer** values (up to 9 digits) typed directly in the LCD, and
- Six additional parameters referenced by the lower-case letters {a, b, c, d, e} and the upper case "F", with values stored previously from X into the parameter buffer using function **LET=**. You can also Swap, Recall or View their values using **SWAP=**, **GET=** and **SHOW=**, followed by the corresponding parameter letter.

### *Formula entry general remarks:*

- The special characters are entered automatically by **^FRMLA**; some examples are the left and right parenthesis, the hash sign (#) for unary negative, the "alien" sign for the Greek letter  $\pi$ , and the ampersand (&) for the MOD function.
- Two- and Three-character mnemonics are completely deleted when using the back-arrow key. Underscores replace the deleted characters, and are removed appropriately with the next character entry
- The LCD will only show the last 12-characters typed in, without any scrolling to the left if you delete back passed that point – at which point you'll be flying blind...
- During the entry process some characters show punctuation signs (like dot, colon). This is for editing purposes only (to inform the back arrow of the length to delete), and they won't be transferred to ALPHA in the final form.
- The formula entry is terminated pressing [ALPHA] or [R/S] indistinctly. This will show the formula and return control to the Operating system. Note that *if close-parenthesis are missing they will be automatically added to the formula.*



- ALPHA contains the complete expression, which can be up to 24 characters long (an audible tone will sound if you reach the limit). If your expression is longer, you'll need to break it in two and evaluate each part sequentially.
- As a bit of intelligence logic, the function will automatically add a left parenthesis right after any function mnemonic has been entered.
- There is a partial built-in syntax checking performed on exit, which verifies matching counts of left and right parenthesis. Too many rights will trigger a "SYNTAX ERR" message, whilst if there are more lefts than rights the code will complete the expression appending as many right parentheses as needed to make the counts match.
- Any other improper expressions won't be noticed until their evaluation time by EVAL\$

#### *Formula evaluation Rules:*

- All operations must be declared explicitly, i.e. not implicit multiplication using "XY" – it needs to be "X\*Y". Ditto for constants, like "2\*π"
- For equal-precedence operations, the interpretation is always done *from left to right*. Thus for instance, "X^Y^Z" calculates  $(x^y)^z$ , and "X&Y&Z" calculates  $\text{MOD}(\text{MOD}(x, y), z)$
- Following the standard conventions, *powers have precedence over all other operators* (addition, subtraction, multiplication, division, modulus). Thus "Y\*5^Z" calculates  $y(5^z)$ , and \*not\*  $5y^z$ , which would be typed "(Y\*5)^Z"
- Also *multiplication, division, and modulus exponents have precedence over the addition and subtraction*. Thus "X+3\*Y" calculates  $x+(3.y)$ , and \*not\*  $(x+3).y$  - and "2^X+5" calculates  $2^x+5$ , and \*not\*  $2^{(x+5)}$  – which would be typed "2^(X+5)"
- Multiplication, division, and modulus have the same precedence level with one another, thus their interpretation follows the "from left to right" rule as stated before.
- And finally, addition and subtraction also have the same precedence level. i.e. the expression "2-5+1" calculates  $(2-5)+1 = -2$ , and \*not\*  $2-(5+1) = -4$  ; which would be typed as: "2-(5+1)" instead.
- As hinted at above, you need to use parenthesis to force an interpretation different from the standard convention. Always remember that "with power comes responsibility" ... so refrain from typing nonsensical strings if you can avoid it ;-)

#### In summary:

^ is the highest precedence

\*, /, and & (mod) are the next highest precedence and are considered equal (left to right)

+, - are the lowest operator precedence and are considered equal (left to right)

*All together now:* "X + Y - Z \* T / L ^ 3" would be:  $(X + Y) - ((Z * T) / (L ^ 3))$

## Formula Evaluation Syntax Table

Finally, the tables below show the symbols and abbreviations used by the functions. All in all, quite a sizable set covering the basic functions plus the Hyperbolic added to the mix as a bonus. Note the mnemonic selection avoids conflicts with variables, like "N" in TAN and the "T" register.

Key	LCD Symbol	Function
[ + ]	<b>+</b>	Sum
[ - ]	<b>-</b>	Subtraction
[ * ]	<b>*</b>	Product
[ / ]	<b>/</b>	Division
[ENTER^]	<b>^</b>	Power
[Σ+]	<b>(</b>	Open Parenthesis
[1/X]	<b>)</b>	Close Parenthesis
[CHS]	<b>#</b>	Negative value
[ ] [ISG]	<b>ABS</b>	Absolute value
[ ] [SF]	<b>IP</b>	Integer part
[ ] [CF]	<b>FP</b>	Fractional part
[SQRT]	<b>Q</b>	Square Root
[XEQ]	<b>&amp;</b>	Modulus
[ % ]	<b>%</b>	Percentage
[ ] [SCI]	<b>R</b>	Square Power
[ ] [ENG]	<b>U</b>	Cube power
[EEX]	<b>E</b>	Exponential
[X<>Y]	<b>FT</b>	Factorial
[RDN]	<b>G</b>	Sign
[SIN]	<b>S</b>	Sine
[COS]	<b>C</b>	Cosine
[TAN]	<b>N</b>	Tangent
[ ] [ASIN]	<b>AS</b>	Arc Sine
[ ] [ACOS]	<b>AC</b>	Arc Cosine
[ ] [ATAN]	<b>AT</b>	Arc Tangent
[STO]	<b>HS</b>	Hyperbolic SIN
[RCL]	<b>HC</b>	Hyperbolic COS
[SST]	<b>HT</b>	Hyperbolic TAH
[ ] [LBL]	<b>AHS</b>	Hyperbolic ASIN
[ ] [GTO]	<b>AHC</b>	Hyperbolic ACOS
[ ] [BST]	<b>AHT</b>	Hyperbolic ATAN
[LN]	<b>LN</b>	Natural Log
[LOG]	<b>LG</b>	Decimal Log

Key	LCD Symbol	Name
[ ] [a ]	<b>a</b>	parameter
[ ] [b ]	<b>b</b>	parameter
[ ] [c ]	<b>c</b>	parameter
[ ] [d ]	<b>d</b>	parameter
[ ] [e ]	<b>e</b>	parameter
[ ] [CLΣ]	<b>F</b>	parameter
[ ] [ π ]	<b>π</b>	pi
[ 0 ]	<b>0</b>	integer
[ 1 ]	<b>1</b>	integer
[ 2 ]	<b>2</b>	integer
[ 3 ]	<b>3</b>	integer
[ 4 ]	<b>4</b>	integer
[ 5 ]	<b>5</b>	integer
[ 6 ]	<b>6</b>	integer
[ 7 ]	<b>7</b>	integer
[ 8 ]	<b>8</b>	integer
[ 9 ]	<b>9</b>	integer
[ ] [X ]	<b>X</b>	Variable
[ ] [Y ]	<b>Y</b>	Variable
[ ] [Z ]	<b>Z</b>	Variable
[ ] [T ]	<b>T</b>	Variable
[ ] [LastX ]	<b>L</b>	Variable

Key	LCD Symbol	Name
[ ] P-R	<b>Σ</b>	SUM Eval
[ ] RTN	<b>P</b>	Product Eval
[ , ]	<b>;</b>	Semi-colon
[ ] VIEW	<b>I</b>	Infinite index
[ ] [x=y?]	<b>&lt;</b>	Comparison
[ ] [x<=y?]	<b>&gt;</b>	Comparison
[ ] [x=0?]	<b>=</b>	Comparison
[ ] [BEEP]	<b>!=</b>	Comparison

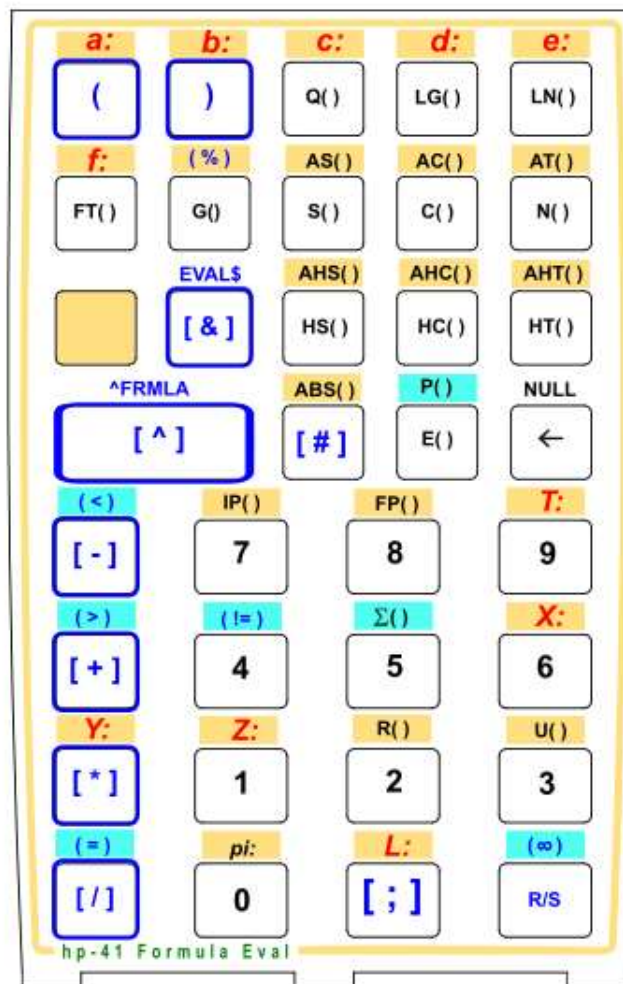
Remember the precedence rules as covered in the previous paragraphs; some will take you a little to get used to but very soon you'll feel comfortable and be putting them to its paces.

Note that the **EVAL\$** functions are programmable and can be used directly, replacing calls to FOCAL subroutines (typically made using "XEQ IND Rnn" with the ALPHA name stored in Rnn). In fact, this module includes versions of SOLVE and INTEGRATE programs using **EVAL\$** directly.

## The Custom Keyboard Overlay.

Using **^FRMLA** simplifies the text entry and speeds the editing process. The picture below shows the custom keyboard overlay used by **^FRMLA**. Most functions have the same location as the original HP-41 functions, so it should be easy to get familiar with the complete layout.

1. There's no need to turn ALPHA on to enter the formula.
2. *Operators* use the standard arithmetic keys plus [XEQ] for MOD and [%] for (%).
3. They are shown in blue font and their keys have a blue frame around them in the overlay.
4. *Variables* and parameters are always accessed as SHIFTed keys. They're shown in red font.
5. Use the numeric digit keys to enter constants directly.
6. *Functions* are shown in black font. They're in both SHIFTed and Un-SHIFTed positions.
7. Use the Back-arrow at will to correct or modify the expression.
8. Press [R/S] or [ALPHA] to terminate the entry.
9. If missing, *the right-parentheses will be added automatically by the function.*



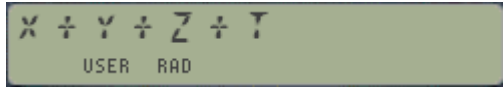
Note: Characters in blue background are only used by EVAL?, EVALΣ and EVALP

Warning: Even though **FRMLA^** is programmable, you should be aware that the expression entered in ALPHA will not be added as text lines steps. For an automated transfer to program memory you'll need to use the function **A-PM** ; see description in a later section.

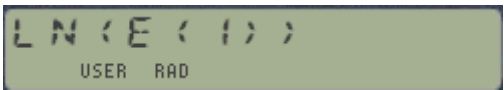
## Show Me. (Missourians rejoice!)

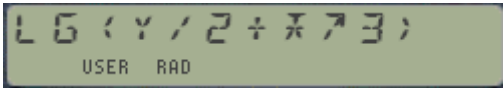
The following examples should be helpful to familiarize yourself with the capabilities and operation of the functions. Set the calculator in RAD mode and populate the stack with the following values:


X=1, Y=2, Z=3, T=4. Then **EVAL\$** returns the following results:

 = 1.000000000  
easy does it...

 = 1.000000000  
a more rugged test!

 = 1.000000000  
a trivial example showing function of a function

 = 1.505235155  
Calculated as: LOG(y/2 + π^3)

 Larger real root of a quadratic equation with coefficients a,b,c stored in the buffer registers

with a=1, b=4, c=1 it returns: -0.267949193

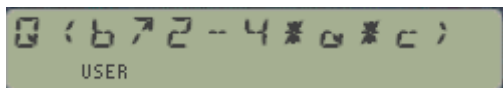
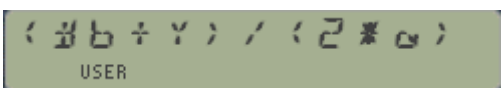
## The Quintuple Twins. Chained Evaluations

You can store the final result in any of the five stack registers – simply using one of the five functions available in the module. The most common destination will be the X register, and that's the one used by **EVAL\$**. The remaining functions have the destination register as last character of the name, thus we have **EVALY**, **EVALZ**, **EVALT** and **EVAL** to choose from, depending on the cases. Note that all except EVAL (for obvious reasons) *will save the previous contents of the destination register in LastX* – which then becomes "LastY", "LastZ", or "LastT".

The result of one evaluation can be used as input parameter in a subsequent one, enabling a chained calculation mode. Being able to choose the location where the result is placed is therefore very convenient for this operation.

Let's see an example to calculate the real roots of the quadratic equation:  $x^2 + 4x + 1 = 0$ , with the coefficients stored in the buffer parameters as follows: a=1, b=4, c=1.

Using a more descriptive formula than the one above makes it a tad too large to fit in a single ALPHA expression, thus we prepare the following two equations and store them in memory:

 , and:   
Stored in R01-R04                      Stored in R05-R08

**RCL\$ 01, EVALY, RCL\$ 05, EVAL\$**                      => -0.267949193

As the Y value is still available, we can obtain the other root typing its corresponding formula:



, then EVAL\$ again => -3.732050808

Note that if the equation has complex roots the discriminant will be negative, and that'll trigger a **DATA ERROR** condition. Should that happen, the expression in ALPHA can become scrambled – which brings us to the next paragraph on error conditions.

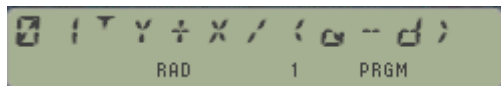
For your convenience the EVAL\_APPS module includes **QRT\$**, an application program to calculate the two real roots of a quadratic equation. Note that **QRT\$** will handle the negative discriminant so ALPHA won't be rendered unusable.

### Entering Formulas in program memory with **A-PM**

No doubt using **^FRMLA** is a powerful and convenient way to enter formula expressions in the ALPHA register, but by itself it's not capable of entering those expressions in program memory. To do that you can use the function **A-PM** – which will transfer the current content in ALPHA as program lines in memory, breaking the text into two when the total length exceeds 15 characters.

To use it just position the program pointer at the location where you want them to be inserted, leave the program mode and execute **A-PM**. The transfer will occur automatically. Note that the function will check for available memory before inserting the new steps – showing **NO ROOM** if such isn't the case, and that if the program pointer is over a ROM location the appropriate "ROM" error will be shown.

For example, the ALPHA string: " $Y \div X / (a - d)$ " is transferred to the program step:



It comes without saying that **A-PM** is an interesting function to say the least – but more than that, the technique used to create program steps from the ALPHA information also lays the foundation for self-programming routines, which will be fully exploited in the "Equation Solver" ROM, a follow-up companion module to this one.

Note: Later on, we'll see another way to manage formula expressions not in ALPHA but in ASCII files in extended memory. This will be done either:

1. Using ALPHA and a combination of the X-Functions INSREC/APPREC, or
2. Using the enhanced ASCII File editor (**ED+**) in the WARP Core module – capable of direct editing of special characters like the parentheses and all other control symbols.

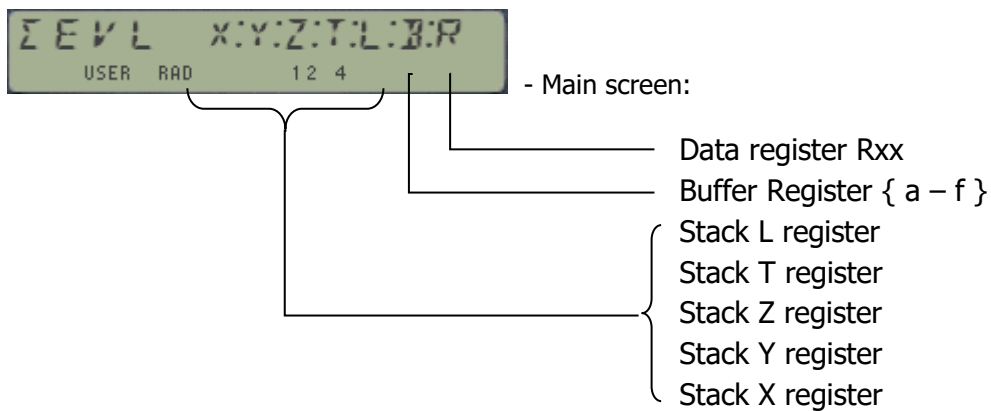
**Revision Note:** Revision 3K includes two more EVAL functions similar to these but that leave the result in the buffer registers {a – f} or in data registers instead of the stack. You can use them to directly store the result there without altering other registers. Both are prompting functions:

for **EVALR** \_ \_ \_ the prompt specifies the data register number, and  
for **EVALb** \_ the buffer register letter (a – f) – or number (1 – 6) in program use.

You only need to enter the value in manual mode, or as subsequent program step in a program. Note that **EVALb** and **EVALl** are implemented as sub-functions to save FAT entries.

### Eval Launcher: the new face of the Eval Module { **ΣEVL** }

Revision 3K adds an **EVAL Function launcher**, grouping all the possible storage destinations of the evaluated result into a convenient command prompt, as shown below:



Besides these functions, note that **EVALP** and **EVALΣ** are also covered by any of the EVAL# cases when the formula expression starts with "P" or "Σ" respectively.

**ΣEVL** is purposely not programmable, so you can use it in a program to select the function of choice. As an example of utilization in a program, the three program steps below will store the result of the evaluation in buffer register "b". The first two steps are to invoke the **EVALb** sub-function (using the sub-function launcher **SF#** and its index), and the third line is the buffer register number (2 for the second one):

```
01 SF#
02 17 ; 17th subfunction
03 2 ; second buffer reg, i.e. "b"
```

The main launcher also provides shortcut access for other five functions not listed at the prompt, as follows:

- **USER** key invokes the **^FRMLA** function
- **ENTER^** invokes **CAT+** for sub-function enumeration catalog
- **RADIX** invokes the **LASTF** facility
- **ALPHA** invokes the **SF\$** sub-function launcher by name
- **PRGM** invokes the **SF#** sub-function launcher by index



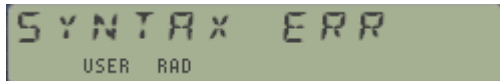
## Caveat Emptor: Error Handling

The functions do a reasonable job at error handling, but the implementation is not bullet-proof, nor does it cover all possible contingencies. There are two main causes for errors: bad syntax (including multiple cases of incomplete or inconsistent expressions), and wrong values (in the function definition domains, ranges of the result etc.).

Of these two the most difficult to handle are the error conditions incurred by the individual functions, say trying to calculate the logarithm of a negative number. You should be aware that in some instances this type will show the **DATA ERROR** message and abort the execution of **EVAL\$** at the point where the error is encountered.

The code includes pre-checking of argument values for FACT, LOG, LN, SQRT, ASIN and ACOS functions, properly skipping the execution for non-valid ones and showing a **"DATA ERROR"** message. Division by zero is also accounted for. The **"ALPHA DATA"** and **"OUT OF RANGE"** conditions should always be properly handled.

And finally the bad syntax condition is also properly handled, and reported using a dedicated **"SYNTAX ERR"** message as well (which I can already tell you'll be soon tired of seeing) :



Note however that the bad syntax conditions can be caused by many different reasons, and not all of them may be captured by the **EVAL\$** logic. For instance: writing two variable names without an operation between them, or a parameter name followed by an open parenthesis without a matching closing one in mid-string. Adding error trapping for every possible contingency will not be productive due to the additional code and the impact in performance. So treat it gingerly, as it corresponds to a very-venerable machine tip-toeing into new realms ;-)

Programmer's Note: As of revision 1H the technique used to scan the formula characters in the ALPHA register was changed to use the CX-OS routine [FAHED] ("Find Alpha HEaDer"). This allowed for a substantial code size reduction (which was quickly re-used for other functionality added to the module), and also made for a speedier execution of the code. As an additional benefit it was possible to remove sub-functions for last-character marking and unmarking, as well as the text-rotation undoing steps – since now the text is not being rotated to begin with. More robust, shorter, and faster code: it doesn't get any better!

## Other Utility Functions.

Functions **^FRMLA** and **EVAL\$** are the two main pillars of the module – but there's much more to it. In addition to the parameter buffer management functions (**LET=**, **GET=**, **SWAP=**, and **SHOW=** described before), the module includes a few other functions very useful to prepare your variables and to manage the expressions entered in by **^FRMLA**. They are described below.

- **STO\$** and **RCL\$** perform both ways of the data copying between Alpha and four contiguous standard data registers. Note that these functions are programmable, and in a program the initial reg# is taken from the program step following the function. Note that while IND arguments are valid, this function does not support Stack or IND Stack arguments.

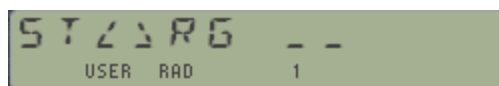
WARNING: Do not try to read directly (with RCL) registers used by **STO\$**, **RCL\$** since this will change the contents of the registers and **RCL\$** will not be able to restore the alpha string correctly. For advanced users, this is because doing a RCL on these registers forces normalization, whereas the values created by these functions are NOT normalized.

- **ST>RG** and **RG>ST** move the 5 Stack registers to/from 5 adjacent data registers, starting at the number entered in the prompt. Like RCL\$ above, in a program the initial reg# is taken from the program step following **ST>RG**. Note as well that while IND arguments are valid, this function does not support Stack or IND Stack addressing.

This method to copy the stack had the advantage to leave the buffer "shadow" registers unaltered, so they can be used to hold parameters in formula evaluations.



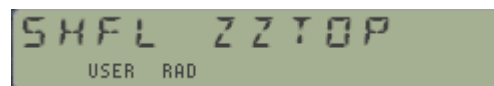
- Similarly, **ST<>RG** and **SWAP\$** exchange the 5 Stack registers and the 4 ALPHA registers respectively with 5 or 4 adjacent data registers, starting at the number entered in the prompt.



- **RIGHT\$** and **LEFT\$** are string manipulation functions. They use the number in X as number of characters to extract *from the right or from the left of the string respectively*. Part of these functions is the deletion of the rightmost or leftmost character, used in a loop to complete the total number of characters. These partial subroutines are also included in the auxiliary FAT as **RADEL** and **LADEL**.
- **B7>ST** and **ST>B7** are small utilities in the auxiliary FAT to move the contents of the stack and the "shadow" registers, back and forth respectively. Obviously, **ST>B7** is equivalent to using "XYZTL" in ALPHA with **SHFL** as describe above, but it uses less bytes in a FOCAL program (four instead of ten). The reverse direction **B7>ST** is written as a sequence of **GET=** calls to populate the stack. (\*)

- **SHFL** is a powerful **Stack Shuffle & Digit Entry** function that makes modifications to multiple stack registers simultaneously in a selective manner, including deletion, digit value entry (0-9) and register exchanges. The function prompts five fields, representing the new arrangement of the stack variables - referenced by the current one.

Thus "XYZTL" would leave things unchanged, and "00000" will be equivalent to CLST plus STO L. For example, to clear registers X, Z and L you'll use "0Y0T0". To swap registers Y and Z, clearing LastX on the fly: "XZYT0". To enter 1,2,3,4 in the stack you'll type "1234L".



In addition to the five stack registers and "zero" for deletions, the four components of the ALPHA register (M, N, O, P) are also allowed in the prompts. This adds flexibility and certain complexity to the scope. It should be noted that the M register is used internally by the function so for all practical purposes it's not really useful here.

**SHFL** is also programmable. In a program the parameter information is taken from the ALPHA register (really the M component as mentioned) as a string containing the five letters for the destinations. *Non-valid letters will leave the corresponding register unaltered.*

Note: You should be aware that **SHFL** uses the parameter buffer (id#=7) to hold a copy of the current stack registers after the shuffling. This could be useful to recall the previous values (basically an UNDO facility) but will conflict with your parameter assignments using **LET=** if you have made them.

The function has a shortcut for the "no changes" case XYZTL. Pressing the radix key at the prompt will make that as the input sequence automatically; creating a "shadow" copy of the stack in the buffer registers as follows:

Buffer id#	Buffer Reg	Type	Used for:
07	b5	BCD value	L-Reg / "a"
	b4	BCD value	T-Reg / "b"
	b3	BCD value	Z-Reg / "c"
	b2	BCD value	Y-Reg / "d"
	b1	BCD value	X-Reg / "e"
	b0	admin	Header

- **FILL** is the little brother of the above, i.e. a sweet and short routine which basically fills the stack with the value in X. So it is equivalent to **SHFL** "XXXXL" in the previous case. This short routine was first published by J.D. Dodin, one of the advanced capabilities pioneers.

(\*) Note that sub-functions need to be accessed using a sub-function launcher, either **SF\$** - typing their name - or **SF#**- entering its corresponding index number. See section in page# 39 for details.

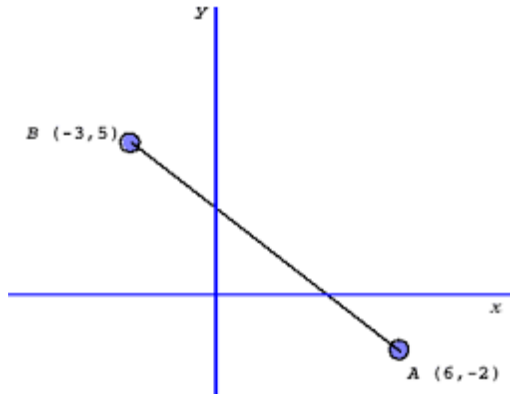
## Example Programs.

### 1. Vector Distances, and Dot Product.

Three more easy examples follow (also included in the ROM) to calculate the distance between two points, (2D and 3D) and the dot product of two 3D-vectors.

For the 2D-distance the two points are expected to be in the stack: P1(T, Z) and P2(Y, X).

Example. Calculate the distance between the points P1(-3,5) and P2(6,-2) from the figure below:



Type:

5, ENTER^, -3, ENTER^, -2, ENTER^, 6,

XEQ "d2\$" => d2=11.40175425

And the formula used is:

$$d2 = \sqrt{(T - Y)^2 + (Z - X)^2}$$

For the 3D-distance and the Dot product the routines will prompt for the point/vector coordinates. Here the first vector is stored in the parameter buffer registers using **SHFL** "XYZTL", which leaves the stack unchanged but also makes the following transformation:

X -> "e"  
Y -> "d"  
Z -> "c"  
T -> "b"  
L -> "a"

Consequently, the formulas used are:

$$3d = \sqrt{(R(X-e) + R(Y-d) + R(Z-c))^2}$$

$$DOT = X * e + Y * d + Z * c$$

Examples. Let V1(1, 2, 3) and V2(4, 5, 6). Calculate the dot product and vector distance.

XEQ "d3\$"  
3, ENTER^, 2, ENTER^, 1, R/S  
6, ENTER^, 5, ENTER^, 4, R/S

"V1=?"  
"V2=?"  
"d3=5.196152423

XEQ "DOT\$" ..... => "DOT=32.00000000

The routine listings are below, really a minimalistic coding just driving EVAL\$ and data input/output:

01	LBL "d2\$"	12	EVAL\$	23	ARCL X
02	"Q((T-Y)^2+(Z-X)^2)"	13	"DOT="	24	PROMPT
03	-"^2)"	14	ARCL X	25	RTN
04	EVAL\$	15	PROMPT	26	LBL 00
05	d2=	16	RTN	27	"V1=?"
06	ARCL X	17	LBL "d3\$"	28	PROMPT
07	PROMPT	18	XEQ 00	29	"XYZTL"
08	RTN	19	"Q(R(Z-c)+R(Y-d) "	30	SHFL
09	LBL "DOT\$"	20	>"R(X-e))"	31	"V2=?"
10	XEQ 00	21	EVAL\$	32	PROMPT
11	e*X+d*Y+c*Z	22	"d3="	33	END

## 2. Polynomial Evaluation using Honer's method.

All it takes is re-writing the expression using in the Honer/Ruffini form, as follows:

Let  $P(x) = [a.x^4 + b.x^3 + c.x^2 + d.x + e]$ ; from here:

$$P(x) = e + x * (d + x * (c + x * (b + x * a)))$$

Examples: with  $a = 1$ ,  $b = 2$ ,  $c = 3$ ,  $d = 4$ , and  $e = 5$ , evaluate the polynomial at  $x = 2$  and  $x = -2$ .

Assuming the coefficients are stored in the homonymous buffer parameter registers (which is done using **LET=** statements repeatedly), we type the formula and proceed to evaluate it:

**^FRMLA** => " $e + x * (d + x * (c + x * (b + x * a)))$ " - 23chars exactly.  
**2, EVAL\$** => 57.00000000  
**-2, EVAL\$** => 9.000000000

The module includes the program "**P4\$**" that prompts for the coefficients and calculates the value. Since the coefficients are stored in the parameter buffer, no standard data registers are used.

The same formula can be used for polynomials of smaller orders, just use zero for the coefficients of the terms not required (obviously at least one term should exist to be a meaningful case).

1	LBL "P4\$"			21	3
2	4			22	RCL 03
3	LBL 00			23	LET=
4	"a"			24	2
5	ARCLI			25	RCL 04
6	" / - = ? "			26	LET=
7	PROMPT			27	1
8	STO IND Y			28	"e+x*(d+x*(c+x*( "
9	RDN			29	" -b+x*a)))"
10	DSE X			30	STOS
11	GTO 00			31	LBL 01
12	"a(0)=?"			32	"X=?"
13	PROMPT			33	PROMPT
14	LET=			34	RCL\$ (00)
15	5			35	EVAL\$
16	RCL 01			36	"P="
17	LET=			37	ARCL X
18	4			38	PROMPT
19	RCL 02			39	GTO 01
20	LET=			40	END

### 3. Orthogonal Polynomials: Legendre, Hermite, and Chebyshev's

These examples use the **EVAL\$** function within a DSE loop, taking advantage of the recurrent definition of these polynomials and the LastX functionality of **EVAL\$**. The results are left in X, and the value of the previous order polynomial is available in LastX. From the definitions:

Type	Expression	n=0 value	n=1 value
Legendre	$n.P(n,x) = (2n-1).x.Pn-1(x) - (n-1).Pn-2(x)$	$P0(x) = 1$	$P1(x) = x$
Hermite	$Hn(x) = 2x.Hn-1(x) - 2(n-1).Hn-2(x)$	$H0(x) = 1$	$H1(x) = 2x$
Chebyshev 1 <sup>st</sup> . Kind	$Tn(x) = 2x.Tn-1(x) - Tn-2(x)$	$T0(x) = 1$	$T1(x) = x$
Chebyshev 2 <sup>nd</sup> . Kind	$Un(x) = 2x.Un-1(x) - Un-2(x)$	$U0(x) = 1$	$U1(x) = 2x$

Examples. Calculate the values for: P(7, 4.9); H(7, 3.14); T(7, 0.314); and U(7, 0.314)

7, ENTER^, 4.9, XEQ "LEG\$"	=> 1,698,444.019,	P7
LastX	=> 188,641.3852	P6
7, ENTER, 3.14, XEQ "HMT\$"	=> 73,726.24330	H7
LastX	=> 21,659.28040	H6
7, ENTER^, .314, XEQ "TNX\$"	=> -0.786900700	T7
LastX	=> 0.338782777	T6
7, ENTER^, .314, XEQ "UNX\$"	=> -0.582815681	U7
LastX	=> 0.649952293	U6

The programs don't make use of any data registers, all operations are performed in the stack.

1	LBL "LEG\$"	26	-
2	4	27	X<>Y
3	"((2*T-1)*Z*X-L*"	28	STO Z
4	-"(T-1))/T"	29	FS? 00
5	GTO 00	30	ST+ X
6	LBL "HMT\$"	31	FS? 02
7	3	32	GTO 02
8	GTO 00	33	"2*Z*X-L"
9	LBL "TNX\$"	34	FS? 01
10	0	35	-"*2*T"
11	GTO 00	36	LBL 02
12	LBL "UNX\$"	37	EVAL\$
13	E	38	ISG T
14	LBL 00	39	NOP
15	X<>F	40	DSE Y
16	RDN	41	GTO 02
17	X<>Y	42	RTN
18	X=0?	43	LBL 00
19	GTO 00	44	E
20	E	45	RTN
21	X=Y?	46	LBL 01
22	GTO 01	47	X<>Y
23	STO T	48	FS? 00
24	FS? 02	49	ST+ X
25	ST+ T	50	END



#### 4. Real Roots of Quadratic Equation.

The short FOCAL program below calculates the real roots of a quadratic equation, checking for negative discriminant beforehand – so DATA ERROR will be shown for complex roots (see program in the appendix for an enhanced versions). Just enter the coefficients in the stack and execute **QRT\$**, and the two roots are shown and left in Y and X on exit.

Example: Calculates the roots of  $Q(x) = x^2 + 2.x - 3$

1, ENTER^, 2, ENTER^, -3, XEQ "QRT\$" => X1= 1; X2 = -3

1	LBL "QRT\$"	9	1	17	AVIEW
2	LET=	10	"(b^2-4*a*c)"	18	"(X+b)/2/a"
3	3	11	EVAL\$	19	EVAL\$
4	RDN	12	SQRT	20	"X2="
5	LET=	13	"(X-b)/2/a"	21	ARCL X
6	2	14	EVALY	22	AVIEW
7	RDN	15	"X1="	23	END
8	LET=	16	ARCL Y		

#### 5. Bessel functions of 1st. Kind for Integer orders.

This short FOCAL program calculates the Bessel functions J and I for positive integer orders, applying a direct sum evaluation of the general terms defined by the formulas below. Note that despite a relative fast convergence the execution takes its time to reach up to the ninth decimal digit, so rounding is done to the display settings. Because of its length exceeding the ALPHA capacity the general term expression is split in two, with an intermediate evaluation into the T register needed. *The final result is left in X and R00*

$$J_{\alpha}(x) = \sum_{m=0}^{\infty} \frac{(-1)^m}{m! \Gamma(m + \alpha + 1)} \left(\frac{x}{2}\right)^{2m+\alpha}$$

To use them just enter N and X, then call the corresponding routine. For example:

1, ENTER^, 1, XEQ "\$JNX" => J(1,1) = 0.4400505860

01\*LBL JNX\$"

02 CF 00  
03 GTO 00

04\*LBL "INX\$"

05 SF 00  
06\*LBL 00  
07 -1  
08 ENTER^  
09 CLX  
10 STO 00

11\*LBL 01

12 RDN  
13 1 ; next index  
14 + ; placed in X  
15 "(Y/2)^(2\*X+Z)"  
16 FC? 00 ; is it J?  
17 ">#1^X" ; alternate  
18 EVALT ; first part  
19 "T/FT(X)/FT(X+Z)"  
20 EVALT ; second part

21 R^

22 ST+ 00 'add to sum  
23 FS? 10  
24 VIEW 00 'show partial  
25 RND ;rounding  
26 X#0?  
27 GTO 01  
28 RCL 00  
29 .END.

## 6. Solve and Integrate Reloaded.

The next two programs are a straightforward application of **EVAL\$** to the well-known Solve & Integrate cases. These routines are brand-new versions, based on the Secant method for Solve and the Simpson rule for Integrate. They assume that the function is entered in the ALPHA register as a formula before calling the program, which you can do using **^FRMLA** of course.

The main advantage is the direct replacement of the "XEQ IND Rnn" calls to the integrand or solved-for functions, replaced by **EVAL\$** instructions. Apart from that everything else is very similar to other well-known routines, like SV and IT from the PPC ROM. Just enter two root guesses (must be different) in Y and X for **SV\$**; or the integration data (number of slices in Z, interval in Y,X) for **IT\$** and call the corresponding routine. It really doesn't get any easier!

For example, to find a root of  $f(x) = \exp(x) - 3$  between  $x=1$  and  $x=2$ :

**^FRMLA** "E(X)-3", 1, ENTER^, 2, XEQ "SV\$" => 1.098612289

And to find the integral of the same function between 0 and 2 - with max# =10 (max number of subintervals needs to be entered in Z, right before the interval [a, b])

10, ENTER^, 0, ENTER^, 2, XEQ "IT\$" => 0.389058523

There you have it, no need to write auxiliary routines (which take RAM memory), or to deconstruct the formula into an RPN-compatible format. The FOCAL listings for these routines are included below. Note how they take full advantage of the formula evaluation functionality and are shorter than the original ones (notably so **SV\$**)

1	<b>LBL IT\$</b>	<i>N, a, b, in stack</i>	27	LBL 09	
2	<b>STO\$ 07</b>		28	RDN	
3	ENTER^	<i>b</i>	29	ST+ Y(2)	
4	<b>EVAL\$</b>		30	RCL Y(2)	
5	STO 11	<i>F(b)</i>	31	<b>EVAL\$</b>	
6	RDN		32	ST+ X(3)	2x
7	"(X-Y)/Z/2"	<i>(a-b)/2N</i>	33	RTN	
8	<b>EVAL\$</b>		34	LBL 01	
9	<b>RCL\$ 07</b>		35	RCL 11	
10	RCL Y(2)	<i>a</i>	36	"X*T/3"	
11	<b>EVAL\$</b>	<i>F(a)</i>	37	<b>EVAL\$</b>	
12	ST+ 11	<i>F(a) + F(b)</i>	38	<b>RCL\$ 07</b>	
13	LBL 00		39	END	
14	CLX		1	<b>LBL "SV\$"</b>	
15	<b>E</b>		2	<b>STO\$ 07</b>	
16	ST- T(0)	<i>decrement N</i>	3	LBL 00	
17	XEQ 09		4	<b>EVALZ</b>	
18	ST+ X(3)	4x	5	X<>Y	
19	ST+ 11	<i>add to sum</i>	6	<b>EVALT</b>	
20	R^		7	"Y-Z*(Y-X)/(Z-T)"	
21	X=0?		8	<b>EVAL\$</b>	
22	GTO 01		9	FS? 10	
23	RDN		10	VIEW X(3)	
24	XEQ 09		11	<b>RCL\$ 07</b>	
25	ST+ 11	<i>add to sum</i>	12	X#Y?	
26	GTO 00		13	GTO 00	
			14	END	

## 7. Use of EVAL\$ with FINTG and FROOT.

For those who use the SandMath module, machine code versions for solving and integrating exist as **FINTG** and **FROOT**, which run faster than the counterpart FOCAL programs **IT\$** and **SV\$**.

The disadvantage to this is that a user program must be created that puts the formula in the Alpha register and executes **EVAL\$**. But that program is minimalistic in nature as we're about to see.

Here is an example of using **FROOT** to solve "SIN(X) + COS(X)" between 120 and 150 degrees.

First, use **^FRMLA** to create " $\sin(X) + \cos(X)$ " in the Alpha register. This does not require the AMC\_OS/X or other module capable to use ALPHA special characters.

You can save this string to registers {R00-R03} using **STO\$ 00**.

Next create a small program (this example uses "SC" for the program name):

```
01 LBL "SC"
02 RCL$ (00) - no need to enter the register index after RCL$ if it is zero
03 EVAL$
04 END
```

Now put 120 in Y, and 150 in X (to find the root between 120 and 150) and XEQ "**FROOT**" which will prompt for the program name, enter "SC" and hit Alpha to execute.

Et voila, 135.0000000 returns as the answer.

To integrate SIN(X)+COS(X) between 0 and 1 radian, XEQ "RAD", put 0 in Y, 1 in X and XEQ "**FINTG**" which will prompt for the program name, enter "SC" and hit Alpha to execute. Result is 1.301168679 (to 9decimal places).

If you're using the AMC\_OS/X module *you can also enter the formula directly in the LBL "SC" subroutine*, instead of using the data registers and **RCL\$** instruction. This will eliminate the need for the data registers and the operation will not take longer to perform.

In that case the program will look like this:

```
01 LBL "SC"
02 " $\sin(X) + \cos(X)$ "
03 EVAL$
04 END
```

For sure this does not address the more complex cases involving special functions, but it pretty much covers 80% of the field.

Note: If you're re-constructing formulas from RPN programs, make sure that the right conventions are used when you transcribe the programs, for instance  $Y^X$ , and  $Y \& X$  for MOD, etc. But this should be much more intuitive this way around than putting the formula in RPN to begin with.

Warning: **SV\$** uses registers R07-R10, while **IT\$** uses registers R07-R11.

No data registers are used by **FROOT** and **FINTG** – which use another memory buffer instead.

## 8. Lambert Function (**WL\$**).

The Lambert W function is the inverse function of  $f(w) = w \cdot \exp(w)$  where  $\exp(w)$  is the natural exponential function and  $w$  is any complex number. The function is denoted here by  $W$ .

As it's well known, the most common way to calculate the Lambert function involves an iteration process using the Newton method. Starting with a good guess the number of iterations is small, leading to a relatively fast convergence.

The formula used for the successive iterative values is:  $z = W(z)e^{W(z)}$ .

$$w_{j+1} = w_j - \frac{w_j e^{w_j} - z}{e^{w_j} + w_j e^{w_j}}.$$

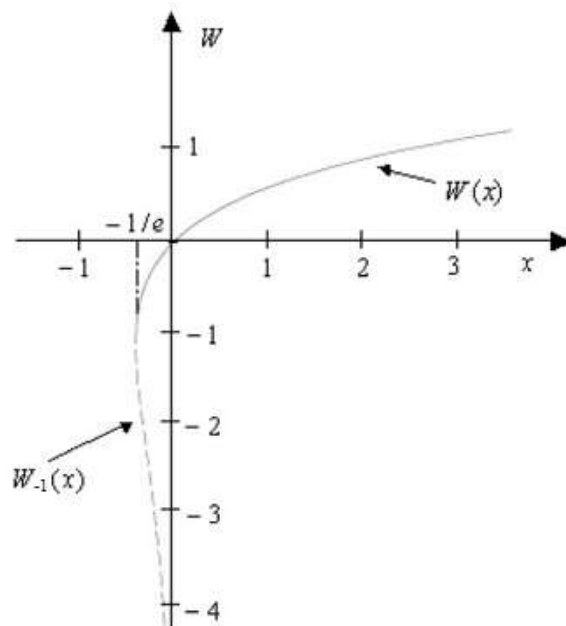
Which with a bit of ingenuity can be written in exactly 24 characters, and therefore only one call to **EVAL\$** is required per each iteration. Assuming the current value  $w$  is in  $X$ , and the argument  $z$  is in  $Z$  the expression for the next value ( $w''$ ) is as follows:

$$W'' = X - (X * E(X) - Z) / (1 + X) / E(X)$$

This is a very good example of how to put those pesky precedence rules to work to our advantage.

The small FOCAL routine below shows the complete code – just 18 steps in total, which include visualization of the iterations when UF 10 is set, as well as dealing with pesky oscillations in the last decimal digit caused by the Newton method in some instances.

1	LBL "WL\$"
2	FIX 9
3	STO Z
4	LN1+X
5	"X-(X*E(X)-Z)/(1"
6	" +X)/E(X)"
7	LBL 00
8	STO Y
9	EVAL\$
10	FS? 10
11	VIEW X
12	X<>Y
13	RND
14	X<>Y
15	RND
16	X#Y?
17	GTO 00
18	END



The initial guess is  $\ln(1+x)$  – which works rather well to obtain the “main” branch result of the function. For arguments between  $(-1/e)$  and zero you can modify the routine to use “-2” instead to calculate the second branch results. Here too this routine does not compete for speed with the all-MCODE Lambert function in the SandMath – nor was it intended to.

## Advanced EVAL\$ Applications.

The module includes a set of functions and FOCAL routines designed to make general-purpose test comparisons, and to evaluate finite and infinite Products, Sums and Series. The FOCAL routines are designed pretty much as if they were MCODE functions, in that they preserve the contents of the stack registers and fully support chained evaluations. Let's see them individually.

### 1. Advanced Test Comparisons with **EVAL?**

Extending beyond the standard set of test functions of the calculator like  $X > Y?$  - **EVAL?** allows you to *compare two general-purpose expressions with one another* – not altering the numeric value of the stack or buffer registers.

Each expression can include any combination of variables, operators and functions as described in the **EVAL\$** sections. The routine uses the test operators "=", "<", ">", and "≠" as delimiters to separate the ALPHA expression in two parts; it then evaluates both and makes the comparison on the resulting values for each of them. Also note that the combination "<=" and ">=" is supported as well.

Note that "≠" denotes the character #29 of the native set, as used in OS functions like  $X \neq Y?$  and  $X \neq 0?$ . It's not the "hash" character (#) used to denote unary minus as seen before.

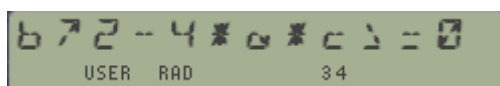


**EVAL?** uses a very fast MCODE function to scan the ALPHA text looking for valid combinations of the test operators. This function is located in the second FAT, and named **TST\$**. If no test operator is present or an invalid combination of them are found, the function will abort the FOCAL program execution and will show the "SYNTAX ERR" message.

Although you could use them if you want, there's no need to enclose the expressions between open and close parenthesis to delimit them. As always, you need to mind the maximum length of the ALPHA text, limited to 24 characters. Also *do \*not\* put a question mark at the end of the text.*

The Boolean result of **EVAL?** is given by the status of user flag 04 at the end of the execution: clear if FALSE and set if TRUE. You can use that as a control in your own routines. In manual mode (not running a program) it'll also show the message "TRUE" or "FALSE" for visual feedback to the user.

Let's see an example: for a more capable way to calculate the roots of the quadratic equation, add a test on the discriminant to determine if it has complex roots. If we write  $Q(X) = aX^2 + bX + c$ , then the evaluation syntax will be as follows:



Note: You can use **EVAL\$** directly on an expression that uses one conditional operator. The execution will be transferred to **EVAL?** for proper evaluation of the Boolean result.

See below the equivalent FOCAL routine listing below for details. This routine was further optimized in revision 1G for speed and simplified program flow – but as a FOCAL program it isn't comparable to the final MCODE function.

- **EVAL?** FOCAL routine used data registers {R00-R10}, and user flags F0-F4.
- **EVAL?** MCODE function uses data registers { R00 – R10 }, but no user flags are used.

1	LBL "EVAL?"		32	X=Y?	
2	STOS (00)	save expression	33	CLX	
3	ST>RG 04	Stack to Regs	34	GTO 00	
4	TSTS	Check for operators	35	LBL 03	
5	ALENG		36	X#Y?	
6	RCL Q(9)	position of character	37	CLX	True Line
7	STO 09	ready for RIGHTS	38	GTO 00	False Line
8	-		39	LBL 05	
9	E		40	FS? 01	double case?
10	-		41	X#Y?	yes, either one
11	FC? 01	was "=" there?	42	X<Y?	no, strictly "<"
12	GTO 01	no, skip adjustment	43	CLX	True Line
13	FC? 02	was ">" there?	44	GTO 00	False Line
14	FS? 00	no, was "<" there?	45	LBL 02	double case?
15	DSE X(3)	yes, adjust marker	46	FS? 01	
16	LBL 01		47	X#Y?	yes, either one
17	LEFT\$	Left ALPHA string	48	X>Y?	no, strictly ">"
18	RG>ST 04	Regs to Stack	49	CLX	True Line
19	EVAL\$	evaluate first part	50	LBL 00	False Line
20	X<> 09		51	"FALSE"	
21	RCL\$ (00)	restore expression	52	CF 04	for the record...
22	RIGHT\$	Right ALPHA string	53	X=0?	
23	RG>ST 04		54	"TRUE"	
24	EVAL\$		55	X=0?	
25	RCL 09		56	SF 04	for the record...
26	FS? 03	"#" ?	57	?RTN	ST# 27
27	GTO 03		58	AVIEW	show if not program
28	FS? 00	"<" ?	59	RG>ST 04	Regs to Stack
29	GTO 05		60	RCL\$ (00)	restore expression
30	FS? 02		61	END	
31	GTO 02				

Examples. Enter the values 4, 3, 2, 1 in the stack registers T, Z, Y, X, respectively.

Then test whether the following comparisons are true or false:

$(Y \uparrow Z - X) / Z \leq T$

$(Y - X) / (Z - T) \geq E(LG(Z * X))$

The MCODE function will show a Boolean YES/NO result message in the LCD if the function is used interactively from the keyboard, but not so during a program execution. It'll also reflect the Boolean status in the general rule "skip if false"..

Note: As of revision 3H, **EVAL?** has become a full-MCODE function. The main benefits are faster execution speed, the use of the standard "YES/NO" LCD messages in manual mode, and "Skip-if-False" rule in program execution - not using user flag 04 anymore.



WHILE we're at it: Putting **EVAL?** to work

**EVAL?** can be used in a FOCAL program to augment the basic testing capabilities provided by the standard stack register comparison functions, such as  $X=Y?$ ,  $X\leq 0?$ , etc. More sophisticated conditions provide greater power in the program flow automation.

The idea is to repeat a calculation (or subroutine) while the expression in ALPHA is true, moving off once the status has changed (obviously influenced by said subroutine); i.e. this is the standard DO/WHILE methodology in high-level languages.

The example below uses **EVAL?** to count until 5; note how a local label is used and the execution is transferred back to it while the count hasn't reached the target value.

01 <b>LBL "COUNT"</b>	07 "X#5" ; testing condition
02 CLX ; count starts at zero	08 <b>EVAL?</b> ; check the test
03 LBL 00 ; marker point	09 FS? 04 ; fulfilled?
04 VIEW X ; for information	10 GTO 00 ; nope, do again
05 1 ; actual code:	11 etc... ; yes, keep going
06 + ; increase counter	

A proper DO/WHILE implementation will use **DO** instead of the LBL instruction, and **WHILE** replacing lines 8-10 – with an automated decision made on the actual status of the test. So there is a combined action in two steps: the first one needs to record the address to return to (done by DO) and the second one needs to trigger the execution of EVAL?, and decide whether to return to the DO address or to continue depending on the test result.

Here's the same program using the brand-new functions:

01 <b>LBL "COUNT"</b>	06 + ; increase counter
02 CLX ; count starts at zero	07 "X#5" ; testing condition
03 <b>DO</b> ; marker point	08 <b>WHILE</b> ; repeat while true
04 VIEW X ; for information	09 etc... ; keep going
05 1 ; actual code:	

Note that in this case the test condition in ALPHA could have been placed outside of the loop, just before the DO instruction since the code within the loop does not alter ALPHA contents. This would be faster, but I've left it next to the WHILE statement for clarity - as in the general case the code within the loop may very well modify ALPHA.

Of course, you could move the central code (increasing the counter in this example) to a subroutine, which in this case makes no sense but in more complex calculations could be very convenient.

Note also that **DO** checks the presence of a matching **WHILE**, searching the program steps following itself until a WHILE statement is found – or until a global END is encountered, in which case it'll put up a "NO BOUND" error message:



## Nested "WHILE" levels are always possible

Each DO/WHILE loop requires two subroutine levels, therefore this implementation allows *up to three DO/WHILE calls in a nested structure*. The only glitch is that the pairing check within the functions won't cover nested configurations - *so the user must make sure that the DO's and WHILE's are matched!*

For example, the routine below will count up to five (in the X-Reg) **three times**, using the Y-register for the outer counter:

01 LBL "DODO"	09 "X#5"	; testing condition
02 CLST	10 WHILE	; repeat while true
03 DO	11 1	; resumes 1 <sup>st</sup> DO
04 CLX	12 ST+ Z	
05 DO	13 RDN	
06 1	14 VIEW Y	; for information
07 +	15 "Y#3"	; second condition
08 VIEW X	16 WHILE	; repeat while true
	17 etc...	; keep going

In summary, this implementation provides a simpler and more advanced program flow control, but it doesn't come gratis: Obviously both instructions need to be paired – mind you, this is also the case using the standard LBL, so it doesn't add overhead. More importantly, one additional return address is used by DO for the automated return from the **WHILE** step. Therefore, the user will only have FOUR return addresses available when the DO/WHILE method is used.

As you have noticed, **WHILE** provides an *MCODE encapsulation of the EVAL? FOCAL program*, plus the branching decisions based on the test result. This is transparent to the user – as long as the EVAL? execution is not stopped mid-way!

Remember that EVAL? uses the following resources internally – therefore they are not available for the FOCAL code within the DO/WHILE loop:

Data Registers:	{R00-R08}	- to preserve the initial Stack and ALPHA contents
User Flags:	F00-F03	- to signal the Boolean operator involved
	F04	- holds the TRUE/FALSE status

Using the RTN address to store the WHILE address has pros and cons. The disadvantage is of course that besides the default RTN level used by the call to **EVAL?**, one additional RTN level is used (or two or three if nested loops are configured). But the advantage is that no additional storage locations are needed for those WHILE addresses, so the complete {XYZTL} stack and {abcdeF} buffer variables are available for the test condition to use.

I trust you'd agree this is very neat stuff, bringing the programming resources up to a more abstract position, usually requiring high-level languages.

## What IF ?- Getting **EVAL?** money's worth!

The same methodology can be used for an IF.(ELSE).ENDIF structure, with only a little more effort to arrange the RTN addresses and inverting the sequence of things.

Here too we'll resort to the **EVAL?** function to determine the Boolean result of the test condition in ALPHA, acting accordingly depending of its TRUE/FALSE status. But contrary to the WHILE case, now the heavy-lifting is performed by **IF** up-front, foreseeing either Boolean result beforehand and arranging the RTN addresses accordingly to serve the desired program flow scheme.

Here's a succinct summary of the operation:

- **IF** will verify that there is a paired **ENDIF** statement following in memory, within the same Global Chain segment (i.e. before the next global END).
- It will then evaluate the test condition and continue normally if the status is TRUE or it will jump over to the instruction following **ENDIF** (or **ELSE** is present), in case the test condition was FALSE.
- Using **ELSE** is optional, and when it's included it will only be relevant if the test condition was FALSE.
- **ENDIF** really doesn't do a thing, apart from demarcating the end of the structure.

Note that unrestricted nested calling of IF.(ELSE).ENDIF is currently *\*not\** supported. This limitation stems from the fact that the technique employed (using the RTN addresses) cannot really match multiple ELSE/ENDIF statements to their matching IF's. Besides, the check for a closing ENDIF will need additional logic to foresee the contingency that multiple IF statements precede a single ENDIF step – getting too complex, the law of diminishing returns really kicks in!

However, it is possible to use DO/WHILE within any of its branches, and vice-versa i.e. it can be placed inside of a DO/WHILE loop.

The example below should illustrate the operation: Use it to calculate the roots of the second-degree equation,  $a.x^2 + b.x + c = 0$ ; with IF.ELSE branches for real or complex roots based on the discriminant. On entry the coefficients (a, b, c) are expected in {Z,T,X}. On exit the real roots (F04 Set); or conjugated complex roots (F04 Clear) are placed in Y,X (Im, Re for complex)

<div style="border: 1px solid black; padding: 2px; display: inline-block;"> <b>01 LBL "QRT#"</b> </div>	09 "(Q(b^2-4*a*c)-b)/2/a"
02 "00XYZ"	<b>10 ELSE</b>
<b>03 SHFL</b>	11 "Q(ABS(b^2-4*a*c))"
04 "b^2-4*a*c>=0"	12 " - )/2/a"
<b>05 IF</b>	<b>13 EVALY</b>
06 "#(Q(b^2-4*a*c)+b"	14 "#b/2/a"
07 " -))/2/a"	<b>15 ENDIF</b>
<b>08 EVALY</b>	<b>16 EVAL\$</b>
	17 END

Try it with  $X^2 = 1$  to obtain:  $x_1 = 1$ ,  $x_2 = -1$  in {Y,X}

Note that the program is intentionally not optimized, for the purpose of the example showing the expressions repeated several times.

## Nesting "IF" levels is not always possible.

There is a conceptual difference between the WHILE and IF implementations related to how (or rather when) the test comparison is made:

- With DO/WHILE the comparison is made at the bottom of the loop, at the WHILE statement. When true, the execution is sent back to the previous DO, and when False it simply continues along its merry way. This poses no issue with nested levels, as each one is self-contained as far as the reference address go (i.e. they don't "overlap" the individual brackets).
- With IF/ENDIF however the comparison is made atop the loop, at the IF statement.
  - When True, the execution follows suit until either a) the next ENDIF statement is found, or b) to the next ELSE statement, in which case it jumps to the next ENDIF below it to skip the code within ELSE and ENDIF.
  - When False, the execution jumps over to said ELSE/ENDIF skipping the top IF branch.

You can see that therein lies the problem: depending on the combination of ELSE/ENDIF statements and their Boolean results, the jumps will go to the first ENDIF grabbed, which won't necessarily be the one paired with the proper IF. This is inherent to the way the RTN addresses are being saved, sequentially at each encountering of the IF statement.

There are however a few supported configurations: you can nest IF/ENDIF groups provided that the subordinate group ends at the bottom of the lower branch of the main group, i.e. *when the two ENDIF statements are consecutive, and only if there are no program steps between them*. This fortunately includes the most common cases without ELSE branches.

Therefore, the subordinate group cannot be in the "True" branch if this has end ELSE step, as this will situate its ENDIF directly before the ELSE statement of the main group. Likewise, you can't place two subordinate groups within the same branch – it'll also break the "contiguous ENDIF's" rule.

Two examples with one subordinate group in blue are shown below, showing the explicit restriction to have contiguous ENDIF statements:

```

IF(1)
  True branch-1
ELSE(1)
  False branch-1
  IF(2)
    True branch-2
  ELSE(2)
    False branch-2
  ENDIF(2)
  —False branch-1 can't
ENDIF(1)
    
```



```

IF(1)
  True branch-1
  IF(2)
    True branch-2
  ENDIF(2)
  True branch-1 can't
ENDIF(1)
    
```



If you're interested to see the underpinnings of these functions refer to the Appendix#4 in page #59, with a detailed analysis of the code and discussion of the operation.

Let's see a few examples of utilization to illustrate the advantage of the new functions.

### Example 1: Fibonacci numbers and yet another Factorial

Starting with the Fibonacci numbers, we'll apply the recursive definition with  $F(0)=0$ ,  $F(1) = F(2) = 1$ :

$$F_n = F_{n-1} + F_{n-2},$$

Entering the initial values in the X,Y registers puts the input number in Z, so the condition will refer to that stack register this time – repeating the loop while its value is greater than 2. Note as well the use of a trivial IF/ENDIF condition to deal with the cases  $n=1$  and  $n=2$ .

Moving to the next example, perhaps the tritest application of the Do/While construction - let's prepare our version using these new functions. Obviously not rocket science, as it's very straightforward application of the definition. With n in X, we use it as a counter multiplying all the values by the partial result.

#### **01 LBL "FBO"**

```

02 INT
03 ABS
04 X=0?      ; is x=0?
05 RTN       ; yes, abort
06 "X<3"
07 IF        ; is X<3?
08 1         ; yes, X=1
09 RTN       ; done.
10 ENDIF     ; no, go on
11 1
12 1
13 "Z>2"
14 DO        ; loop starst
15 +
16 LASTX
17 X<>Y
18 DSE Z
19 WHILE     ; repeat if >2
20 RTN
    
```

#### **21 LBL "FCT"**

```

22 INT
23 ABS
24 X=0?      ; is X=0?
25 RTN       ; yes, abort
26 1
27 X<>Y      ; f=1
28 "X>1"
29 DO        ; loop starts
30 ST* Y     ; f=f*n
31 1
32 -         ; n=n-1
33 WHILE     ; repeat if n>1
34 X<>Y
35 END
    
```

Note that these short programs are included in the companion module, "EVAL\_APPS".

### Example 2: Arithmetic-Geometric Mean

Here we're using EVAL\$ functions in the Do-WHILE loop for a good illustration of what this module is capable of:

$$a_{n+1} = \frac{1}{2}(a_n + g_n)$$

$$g_{n+1} = \sqrt{a_n g_n}.$$

01 LBL "AGM\$"	08 EVAL\$ ; an
02 "ABS(IP(Y))	09 "Q{X*Y}"
03 EVALY	10 EVALY ; gn
04 "ABS(IP(X))"	11 VIEW X ; show value
05 EVAL\$	12 "X#Y"
06 DO	13 WHILE ; in FIX 10
07 "(X+Y)/2"	14 END

Which isn't only a *very sort and compact routine*, but also very easy to read and troubleshoot – a far cry compared to the FOCAL program listings!

### Example 3: Ulam's Conjecture

A well-known subject that has been addressed before using different approaches (MCODE included), now in a completely new fashion on the 41 platform.

$$f(n) = \begin{cases} \frac{n}{2} & \text{if } n \equiv 0 \pmod{2} \\ 3n + 1 & \text{if } n \equiv 1 \pmod{2}. \end{cases}$$

This final example uses an IF/ELSE/ENDIF structure inside of a DO-WHILE loop. It really brings the point home: simplicity and legibility, albeit there's an obvious speed penalty using this approach.

01 LBL "ULAM\$"	13 "X/2" ; yes, halve it
02 "ABS(IP(X))"	14 ELSE
03 EVAL\$ ; naturalize input	15 "3*X+1" ; no, increase it
04 "X=0"	16 ENDIF
05 IF	17 EVAL\$ ; evaluate new
06 RTN ; abort if x=0	18 VIEW X ; show value
07 ENDIF	19 "Y+1"
08 "0"	20 EVALY ; increase count
09 EVALY ; reset count	21 "X#1"
10 DO	22 WHILE ; repeat while
11 "(X&2)=0" ; MOD (x,2)	23 VIEW Y ; show length
12 IF ; is x even?	24 END



## *The next logical case for FOR/NEXT*

The next logical structure is no doubt the FOR...NEXT loop, perhaps the most popular program flow control known by every programmer, no matter the level of expertise.

Obviously, the HP-41 platform comes with **ISG** and **DSE**, which combined with **LBL** and **GTO** perform a similar function to the FOR...NEXT loops. Indeed, their fundamental operation is very comparable, although the FOR...NEXT syntax offers more convenience – at the cost of more variables and memory requirements used of course.

The implementation presented here is a compromise between the native ISG/DSE and the most capable FOR...NEXT concept:

- **FOR** expects the **bbb.eee:ss** control word in X, defining the initial and final index values, as well as the step size. If the control word is positive (with  $bbb < eee$ ) then the STEP increments the index, whereas if negative (with  $bbb > eee$ ) then it decrements the index. By default, if  $ss=0$  the step is one (standard hp-41 convention for loop functions).
- The index variable is declared using the **SELECT** instruction in the WARP Core module. With it you can designate any data or stack register as the index - which is going to be fundamental for nested configurations if course. The default value is the X-register.
- **NEXT** does the index increment or decrement and loops back to the FOR location of a new iteration kkk until all of them are done (when  $kkk > eee$ ); in which case the execution continues with the program steps below it.

Note that **NEXT** increments (or decrements if the control word is negative) the value in the SELECT'ed register, not X – unless of course X is the selected register. It comes without saying that the instructions executed within the loop should not modify the contents of the SELECTed register – unless you're a power user and want to modify the index variable intentionally.

For example, the routine below uses R01 and R02 as SELECT'ed index to play three TONE 0 with another two TONE 1 instructions for each of them, with a BEEP to end- i.e. nine tones in total as follows: T0-T1-T1; T0-T1-T1; T0-T1-T1

### **36 LBL "TONES"**

```

37  SELECT 01 ; S1 variable
38  -2       ; = 1.003
39  FOR      ; S=01
40  TONE 0
41  SELECT 02 ; S2 variable
42  -1       ; =1.002
```

```

43  FOR      ; S=02
```

```

44  TONE 1
```

```

45  NEXT     ; Next S2
```

```

46  SELECT 01 ; S1 variable
```

```

47  NEXT     ; Next S1
```

```

48  BEEP
```

```

49  END
```

Unlike the previous two structures, FOR...NEXT doesn't make utilization of **EVAL?** – or any evaluation function for that matter. They can of course be used inside the loop, but if you do so remember that data registers {R00-R01} are used by **EVAL?** itself, therefore, they shouldn't be used as SELECT'ed registers for the loop index.

Each FOR..NEXT loop takes one subroutine level, thus you can build *up to six nested loops* – provided that there aren't any additional subroutines called inside any of them of course. Remember to always match the number of FOR and NETX instructions – this is not checked by the code.

The next example is slightly more useful than playing tones: Bubble Sorting (once again!) - although it's a non-practical solution due to the slow speed it is very indicated to document the operation of the functions.

Two versions are included one with data movement and another that makes the data comparison in-place. The input should be the FROM.TO control word (bbb,eee) delimiting the memory area to be sorted. The programs very much read like BASIC routines to do this job. Both versions show a nice implementation of a two nested FOR...NEXT loops and require functions from the WARP\_Core module (which is already needed for SELCT, anyway).

Version#1. Data in-place. We'll use stack registers Y and X as loop variables.

<pre> 01 LBL "BSORT1"; bbb.eee in X 02 -E-3          ; -0.001 03 X&lt;&gt;Y 04 +             ; bbb.(eee-1) 05 LASTX         ; bbb.(eee-1) 06 SELCT Y       ; outer loop 07 FOR           ; kkk.(eee-1) 08 E 09 +             ; (kkk+1).eee </pre>	<pre> 10 SELCT X       ; inner loop 11 FOR           ; (kkk+1).eee 12 SELCT IND X 13 ?S&lt; IND Y 14 S&lt;&gt; IND Y 15 SELCT X 16 NEXT          ; do next X 17 SELCT Y 18 NEXT          ; do next Y 19 END </pre>
---	--

Version #2. Data moved to the stack for the comparison. We'll use R00 and R01 as loop variables. It is slightly longer and obviously {R00-R01} are reserved (can't contain data to sort).

<pre> 01 LBL "BSORT2"; bbb.eee in X 02 E-3 03 - 04 SELCT 0 05 FOR           ; bbb.(eee-1) 06 RCL 00        ; kkk.(eee-1) 07 1.001 08 + 09 SELCT 1 10 FOR           ; (kkk+1).eee 11 RCL IND 01     ; R(eee) value 12 RCL IND 00     ; R(kkk+1) value </pre>	<pre> 13 X&lt;Y?          ; already sorted? 14 GTO 00        ; yes, skip 15 RCL 01        ; (kkk+1).eee 16 X&lt;I&gt;Y         ; (*) 17 LBL 00 18 NEXT          ; do next reg 19 SELCT 0 20 NEXT          ; do next Reg 21 END </pre>
---	---

(\*) Function **X<I>Y** does IND X <> IND Y. It is also in the WARP Core

Once again, these routines are very slow. For real-life applications (say more than 10 registers to sort) you really should be using a MCODE function like **SORTRG** in the SandMath, or a more intelligent routine such as **S2** and **S3** in the PPC ROM (and derivatives).

## 2. Evaluating Sums & Series with **EVALΣ**

This routine provides the capability to calculate sums or even (convergent) infinite series, just by direct repeat execution of the general term – either the number of terms specified for sums, or until the contribution to the partial sum is negligible for convergent series.

The syntax requires the initial and final values for the indexes (they *must be constants*), separated by semi-colons “;” plus the function to sum – which uses the X register as index parameter. The first character must be a Sigma and the complete expression must be enclosed by open and close parenthesis.

The complete syntax can be put together using **^FRMLA**, which has been upgraded to also handle the Sigma character and the semi-colons. For example, to calculate the harmonic number for n=25 we just call **^FRMLA** to type:

**Σ(1,25,1/X)**

Note that the general term does not need to be enclosed by parenthesis – the routine knows it starts right after the second semi-colon and ends right before the final parenthesis.

As an example of infinite series, let’s calculate the Erdos-Borwein constant 1.606695153 using the following syntax (note the letter “I” used in the final index for infinite, but any non-numeric character will work as well):

**Σ(1,I,1/(2^X-1))**

This routine uses the current decimal settings to determine the accuracy of the result. FIX 9 is the most accurate but will require the most number of terms (and longest time) to converge.

Setting user flag 10 provides a visual feedback of the result after each new term as been added to the sum. This is very useful if the convergence is slow (like in this case).

### Evaluation Functions as Power Series

You can also use **EVALΣ** to calculate functions expressed as power series. In that case the function variable is assumed in the X register on entry, but it gets moved to Y at the beginning of the routine execution. Therefore, it’s represented by “Y” in the evaluating syntax, and not by “X” - which is reserved for the index value (usually “n” or “k” in these formulas).

For example, to calculate the exponential function we’ll use the syntax below:

**Σ(0,I,Y^X/FT(X))**

Don’t forget to set the number of decimal places to the desired accuracy.

**EVALΣ** is a direct application of **EVAL\$** used in a loop. It leaves the result in X, and the initial argument in L – preserving the initial contents of the stack Y-Z-T registers. It uses data registers {R00-R10} and user flags F0, F1.

### 3. Evaluating Products with **EVALP**

The product counterpart is just a small modification of the same routine, and therefore shares the same general characteristics and data registers requirements. In this case the initial character must be a "P" instead of sigma, but the rest of the syntax is identical.

For example, let's calculate the Permutations of n elements taken k at a time:  $\frac{n!}{(n-k)!}$ .

Which can be calculated as the product of the last (n-k+1) terms of the numerator - from (n-k+1) to n - rather than using the FACT function – avoiding so "OUT OF RANGE" errors if n>69 and k>=1

Thus, the required syntax should be of the form: "P(n-k+1;n;(n-k))"

All we need is a way to place the correct values in the ALPHA string, and the perfect function to do that is **ARCLI** in the AMS\_OS/X module (or any of its equivalents like **AIN** or **AIP**). Say we start the routine with n in Y and k in X, then we use the small program below:

01 LBL "NPK\$"	08 "P("
02 STO Z(1)	09 AINT
03 CHS	10  -" ;"
04 E	11 LASTX      n
05 +            n-k	12 AINT
06 X<>Y      leaves k in L	13  -" ;X"
07 +            n-k+1	14 XROM "EVALP"
	15 END

The complementary routine to calculate the Combinations CNK is easy done using NPK as basis:

01 LBL "NCK\$"	$\frac{n!}{k!(n-k)!}$
02 XEQ "NPK\$"	
03 "X/F(Z)"	
04 EVAL\$	
05 END	

Examples:

52, ENTER^, 5, NPK\$	-> 311,875,200
52, ENTER^, 5, NCK\$	-> 2,598,960

The routine code for both **EVALΣ** and **EVALP** is shown below. As you can see only functions from this module are used – this makes the program a little longer but it's more convenient for compatibility reasons.

In the final versions of the module these functions are hybrid: FOCAL with MCODE header. The first part is MCODE, doing all the syntax verification and preparing the variables. The second part is FOCAL, doing the loop calculations as per the code in next page.

Note: As of revision 2H you can use **EVAL\$** directly on an expression that uses the sigma ("Σ") or product ("P") characters. The execution will be transferred to **EVALΣ** or **EVALP** automatically.

1	LBL "EVALΣ"		37	LBL 05	
2	CF 01		38	RCL 07	function argument
3	GTO 01		39	RCL 10	get current index
4	LBL "EVALP"		40	INT	stripe off limit
5	SF 01		41	EVAL\$	evaluate expression
6	LBL 01		42	FC? 01	sums?
7	STOS 00	save in {R00-R03}	43	ST+ 09	add to partial sum
8	ST>RG 04		44	FS? 01	products?
9	126	"Σ" character	45	ST* 09	factor in partial product
10	FS? 01		46	FS? 10	need to show?
11	80		47	VIEW 04	yes, oblige
12	XEQ 00	remove & check	48	FC? 00	infinite series?
13	40	"(" character	49	GTO 02	no, skip over
14	XEQ 00	remove & check	50	ISG 10	next index
15	RADEL	remove close paren	51	NOP	
16	ANUM	get initial index	52	FS? 01	products?
17	STO 10		53	DSE X(3)	
18	XEQ 03	advance to next field	54	NOP	
19	ANUM	get final index	55	RND	as per the dsp settings
20	ATOX	get first char of field	56	X#0?	was term null?
21	57	number limit	57	GTO 05	no, do next
22	X<=Y?	not a number?	58	GTO 01	yes, show final result
23	CLX	then clear it	59	LBL 02	finite SUM
24	X#0?	was a number?	60	ISG 10	next index
25	RCL Z	yes, recover value	61	GTO 05	repeat if not done
26	,1		62	LBL 01	
27	%	divide by 1,000	63	RCL 09	final result to X
28	ST+ 10	add to control word	64	X<> 07	
29	CF 00	default: SUM	65	STO L(4)	
30	X=0?	wasn't a number?	66	RG>ST 04	
31	SF 00	SERIES mode	67	RCL\$ 00	restore initial syntax
32	XEQ 03	move to next field	68	RTN	done.
33	CLX	initial value	69	LBL 00	
34	STO 09	reset sum	70	ATOX	remove char
35	FS? 01	products?	71	X=Y?	bad syntax?
36	ISG 09	yes, start = 1	72	RTN	no, return
			73	SYNERR	yes, show "SYNTAX ERR"
			74	LBL 03	
			75	ATOX	remove char
			76	59	"." character
			77	X#Y?	got it yet?
			78	GTO 03	no, do next
			79	END	yes, done.

Note also that as of Revision 2H the programs have a MCODE header instead of a FOCAL one. This facilitates the execution transfers from EVAL\$ in case that special characters are found in the string.

Always remember that the index values used in these two functions need to be constant values, i.e. you cannot use a variable for them. The examples included in the manual show how to circumvent this restriction using AINT – which adds the current value of the "X" variable to ALPHA.

### Examples: Gamma and Digamma functions.

Armed with the routines described before, it is relatively simple to write short FOCAL programs to calculate the Gamma and Digamma functions. To that effect we'll use the Lanczos approximation for Gamma, with the well known formula:

$$\Gamma(z) = \frac{\sum_{n=0}^N q_n z^n}{\prod_{n=0}^N (z+n)} (z+5.5)^{z+0.5} e^{-(z+5.5)}$$

$q_0 =$	75122.6331530
$q_1 =$	80916.6278952
$q_2 =$	36308.2951477
$q_3 =$	8687.24529705
$q_4 =$	1168.92649479
$q_5 =$	83.8676043424
$q_6 =$	2.5066282

Note the product in the denominator, which will be calculated using **EVALP**.

Examples: 1, XEQ "GAM\$" -> 1.000000000  
 PI, XEQ "GAM\$" -> 2.288037797  
 -5.5, XEQ "GAM\$" -> 0.010912655

As you can see the program also works for values  $x < 0$  (not integers), including support for these arguments using the reflection formula:

$$\Gamma(1-z) \Gamma(z) = \frac{\pi}{\sin \pi z}.$$

On the other hand the formula for the Digamma function (Psi) is a combination of a logarithm and a pseudo polynomial expression in  $u = 1/x$

$$\Psi(x) = \log(x) - \frac{1}{2x} - \frac{1}{12x^2} + \frac{1}{120x^4} - \frac{1}{252x^6} + O\left(\frac{1}{x^8}\right)$$

programmed as:  $u^2\{[(u^2/20-1/21)u^2 + 1/10]u^2 - 1\}/12 - [\ln u + u/2],$

The implementation also makes use of the analytic continuation to take it to arguments greater than 9, using the following recurrence relation to relate it to smaller values - which logically can be applied for negative arguments as well, as required.

$$\Psi(x+1) = \Psi(x) + \frac{1}{x}.$$

Note the Summation in this expression (with as many terms as delta between the argument and 9), which will be calculated using **EVALΣ**.

### Examples:

PI, XEQ "PSI\$" -> 0.977213308  
 1, XEQ "PSI\$" -> -0.577215666 (opposite of Euler's constant)  
 -7.28, XEQ "PSI\$" -> -4.651194214

And here's the program listing for these functions. Note we're using several tricks and executing repeated times the **EVAL\$** functions, taking care of partial expressions of the formula each time.

1	LBL "GAMS"	$x > 0$	1	LBL "PSI"	
2	CF 04		2	STO 00	x
3	X<0?		3	0	initial delta
4	X=0?		4	X<>Y	
5	GTO 04		5	9	accuracy limit
6	RAD		6	X<>Y	
7	SF 04		7	LBL 00	
8	"1-X"	$X = (1-X)$	8	X<Y?	
9	EVAL\$		9	X=Y?	$x \geq 9$ ?
10	LBL 04		10	GTO 01	yes, exit loop
11	"P(0;6;Y+X)"	denominator	11	E	
12	EVALP	saves x in R09	12	ST+ T(0)	increase delta
13	STO 09	save for later	13	ST+ Y(2)	increase argument
14	5,5		14	RDN	fix stack
15	LET= a		15	GTO 00	do next
16	LASTX	x	16	LBL 01	
17	"(X+a)^(X+1/2)/E"	transcendent term	17	"LN(1/X)+1/2/X"	
18	-(X+a)"		18	EVALY	
19	EVAL\$		19	"R(1/X)"	
20	STO 10	partial result	20	EVAL\$	
21	75122,63315	q0	21	"((X/2--1/21)*X+	
22	LET= a		22	-"1/10)*X-1"	
23	80916,62789	q1	23	EVAL\$	
24	LET= b		24	"X*L/12-Y"	
25	36308,29514	q2	25	EVAL\$	
26	LET= c		26	RCL Z(1)	delta
27	8687,245297	q3	27	X=0?	was $x \geq 9$ ?
28	LET= d		28	GTO 01	yes, skip adjustment
29	1168,926495	q4	29	E	
30	LET= e		30	-	
31	83,86760434	q5	31	"P(0;"	
32	ENTER^		32	AIN	
33	2,5066282	q6	33	RDN	
34	LASTX	x	34	RCL 00	x
35	"c+X*(d+X*(e+X*("	polynomial term	35	-"1/(Y+X))	
36	>"Z+X*Y)))"	part-1	36	EVALP	
37	EVAL\$		37	*	ok, I cheated here...
38	"a+b*L+X*L^2"	part-2	38	X<>Y	
39	EVAL\$		39	LBL 01	
40	RCL 10	get partial result	40	X<>Y	
41	*	factor it in	41	CLD	
42	RCL 09	get denominator	42	END	
43	/	divi by it			
44	FC? 04	negative argument?			
45	GTO 04	no, skip			
46	RCL 08	get (1-x)			
47	"pi/Y/S(pi*(1-X))"	reflection formula			
48	EVAL\$				
49	LBL 04				
50	CLD	clear display			
51	END	done.			



## *A new scripting language using Extended Memory*

One of the goals for the final version of this module was to allow a series of steps to be stored to automatically run as a scripted language. The perfect place for such a script would be an ASCII file in Extended Memory. After entering the steps into the ASCII file, all the module user would need to do is initialize the stack and buffer variables (X, Y, Z, T, L, a, b, c, d, e) if required, enter the name of the file into the Alpha register, and execute the script reading program.

This goal has been realized in this version of the Function Evaluation module! All the EVAL functions (including **EVAL?**, **EVALΣ**, and **EVALP**) have been brought together to create a scripted language including a primitive "GOTO" function, labels for the "GOTO", and decision making statements.

Note that two versions of the program exist, the standard **EVALXM** and a more capable **EVLXM+**. Either one requires that the WARP\_CORE module be plugged in, as they make extensive use of its **?SELECT/CASE** functions in control branches..

Before describing each type of script line, a few definitions are needed:

- **'Variable'** can represent any of the stack variables used by **EVAL\$** (or its siblings), i.e. X, Y, Z, T, or L; or it can also be one of the buffer variables a, b, c, d, e, or F.
- **"\_"** represents a blank space character
- **'StackVar'** is restricted to one of the stack variables X, Y, Z, T, or L.
- **'Formula'** represents any of the strings used by **EVAL\$** or its siblings as a line to evaluate.
- **'Value'** represents any valid real value that can be read from the Alpha register via ANUM.
- **{Condition}** represents any conditional operator understood by **EVAL?**, i.e. **<**, **<=**, **=**, **>=**, **>**, or **≠**, as described in the EVAL? section.
- **'Label'** represents any single character – even special chars.
- **'RegNumber'** represents any valid memory register number, it does NOT require a leading zero 0.
- **'Params'** represents the parameters supplied inside **"("** and **")"** used by **EVALΣ** and **EVALP**.

With those definitions in mind, here is the syntax used by the scripting language. Each record in the ASCII file in extended memory should be one of the following:

1. **'Variable'\_Value'** [the space in between "Variable" and "Value" is required]
2. **'StackVar'=Formula'** [the equal sign is required]
3. **'Label':** [the colon after the "Label" is required].
4. **G\_Label'** [the space between the G and 'Label' is required, "GOTO" statement]
5. **'Variable'S'RegNumber'** [stores value at "Variable" into memory register "RegNumber"]
6. **'Variable'R'RegNumber'** [stores value from register "RegNumber" into location "Variable"]
7. **??Formula'{Condition}'Formula'** [conditional statement, skips next statement if FALSE]
8. **ΣΣ('Params')** [for using summation function EVALΣ, value of sum replaces X, prev X to L]
9. **PP('Params')** [for using product function EVALP, value of product replaces X, prev X to L]

And **EVLXM+** adds the following additional capabilities:

10. **F** 'Variable'='Params' [Space between the F and 'Variable' is required, "FOR" statement]
11. **NX** [Next statement that goes back to FOR statement above]
12. **DO** [Beginning of while statement loop]
13. **W** 'Formula' {Condition} 'Formula' [While condition is true, repeat DO loop]
14. **IT** ('Params') ["Params" represents "Divisions;From;To;Equation" for IT\$]
15. **SV** ('Params') ["Params" represents "Guess1;Guess2;Equation" for SV\$]
16. **GF** 'Label' [Forward only GOTO search for "Label"]
17. **GB** 'Label' [Backward only GOTO search for "Label"]

*Note1:* The assignment statement (form at #1 above) accepts any real value (i.e. -1.2345E-67), but the evaluation statements (Format #2 above) formulas can only contain integers, not real values.

*Note2:* The last eight statements are only available in the new **EVLXM+** program. In addition to these new statements, "Params" in **ΣΣ**, **PP**, **IT** and **SV** can ALL be formulas: the integer indexes are no longer restricted to being integers (as **EVLXM+** uses enhanced versions of EVALΣ and EVALP, aptly named **EVLΣ+** and **EVLΠ+**). "I" for "infinite" looping in **ΣΣ**, **PP** is still supported, and since this parameter can also be a formula, must be "I" by itself to represent infinity.

Solve and integrate.

For the **IT** statement, the parameters are the Z, Y, and X values needed for **IT\$**, and the formula put into the alpha register. However, for the IT statement these values can be formulas that will be evaluated, and the results put on the stack for **IT\$** to use. The first parameter is evaluated and placed in Z, then the second parameter is evaluated and placed in Y, finally the third parameter is evaluated and placed in X. During execution phase, X, Y, and Z are pushed into Y, Z, and T.

For the **SV** statement, the parameters are the Y and X values needed for **SV\$**, and the formula put into the alpha register. Again, for the **SV** statement these values can be formulas that will be evaluated, and the results put on the stack for **SV\$** to use. The first parameter is evaluated and placed in Y, then the second parameter is evaluated and placed in X. During execution phase, X, Y, and Z are pushed into Y, Z, and T.

## Loop Control

For/Next statements should be used as follows:

```
F variable=begin;end
... statements in the loop
NX
```

On finding the F statement, the variable selected is initialized to the begin value. So, for example the statement **F\_X=5.5;7.5** will initialize X to 5.5, then continue on.

On finding the NX statement, the F statement will be search for, and then the variable will be incremented by 1. It will then be checked to see if it is greater than the end value. If so, the next statement executed will be that after the NX statement, otherwise it will be the next statement following the F statement (it will repeat all statements in the loop).

Do/While statements should be used as follows:

### DO

... statements in the loop

**W** formula{condition}formula

On finding the **W** statement, the condition is evaluated. If true, then the **DO** statement will be searched for, and the next statement following the DO will be executed. Otherwise, it will continue with the next statement after the **W**.

All this may seem confusing, so an example might be in order.

Create an ASCII file in extended memory named "TEST", size it to 20 registers (oversized so you can play with the example afterward). Put the following records into the file using your favorite editor:

```
00 Y 1.0           ; initial Y value
01 X 1.0           ; initial X value
02 A:              ; label A
03 X=X+Y           ; add it to sum in X
04 Y=L             ; recall LastX to Y
05 ??X<100         ; les that 100?
06 G A             ; yes, goto A:
```

Put **TEST** into the Alpha register and XEQ "**EVALXM**". What you have done is find the first Fibonacci number above 100. Note the steps show up as they are executed, and the GOTO statement will show the goose as it searches for the label.

An explanation of the steps follows:

```
Y_1.0      puts 1.0 into the Y register [you could have also just used Y 1]
X_1.0      puts 1.0 into the X register [you could have also just used X 1]
A:        this is a label, it will be used by the GOTO statement at the end
X=X+Y      replaces X register with the sum of X and Y, L becomes the previous value of X
Y=L        replaces Y register with contents of L
??X<100    this tests to see if X is less than 100, if NOT, the GOTO statement is skipped
G_A        go to label A
```

Warning: Direction of search.

The GOTO statement *will search from the beginning of the ASCII script file* for the label. This means some searches could take a long time if the label is far down into a program. The **EVLXM+** program includes two additional statements to make this search faster (**GF** and **GB**).

Let's see how to rewrite the Fibonacci example with DO/WHILE statements instead:

```

00 Y 1.0
01 X 1.0
02 DO                ; start of DO/WHILE
03 X=X+Y
04 Y=L
05 W X<100           ; end of WHILE loop, keeps looping until X>=100

```

The DO statement replaces the label, and we take one less statement in the program.

#### More scripting examples:

To store [Z] in R11 use:	<b>ZS</b> 11
To recall [R15] to [a] use:	a <b>R</b> 15
To goto label A searching backwards only use:	<b>GBA</b>
To find the sum of X for X=1 to 5 use:	<b>ΣΣ</b> (1;5;X)
To find the product of X for X=1 to 5 use:	<b>PP</b> (1;5;X)

Variables vs. Integer indexes.

If you are using the new **EVLM+** program, then in place of the last two examples...

If Y = 2 and Z = 6 you could use instead:

**ΣΣ**(Y-1 ; Z-1 ; X) or **PP**(Y-1 ; Z-1 ; X)

Notice that Y and Z are not moved until the execution phase (where the formula "X" is used).

#### Warning: Record Lengh.

Even if the ASCII records can hold up to 256 characters, the complete strings cannot exceed 24 characters, as they'll be put in ALPHA and handled by **EVLM**. This restriction does include the leading control characters at the beginning of the record, like "**X**=", "**??**", etc. So in this respect the scripts are a bit more restrictive than if you use the individual functions in FOCAL programs.

**Example: Gamma ASCII script.**

Now gilding the lily, here you have an ASCII script to calculate Gamma(x) for  $x > 0$  using **EVALXM**.

Note that to include the special characters like parenthesis you'll need to first place them in ALPHA (either using **XTOA** or the direct entry feature from the AMC\_OS/X Module); and then append them to the ASCII record using **APPCHR** (or **APPREC** if you write in ALPHA the complete record). See the **GMXM** routine in the companion EVAL\_APPS ROM to see how this can be done programmatically.

Warning: this script *assumes your radix is set to decimal point, not comma*. Use SF 28 if needed.

This script is also based in the Lanczos formula. The listing is similar to the FOCAL program in the previous section, although modified due to the 24-chars limitation in ALPHA for the evaluating expressions. Only arguments  $x > 0$  are supported, you can go ahead and include the reflection formula as an exercise ;-)

```

00 PP(0;6;Y+X)
01 XS9
02 X=L
03 a 5.5
04 X=(X+a)^(X+1/2)/E(X+a)
05 XS10
06 a 75122.63315
07 b 80916.62789
08 c 36308.29514
09 d 8687.245297
10 e 1168.926495
11 Z 83.86760434
12 Y 2.5066282
13 X=L
14 X=X*(d+X*(e+X*(Z+X*Y)))      ; had to leave off c due to length
15 Z=L                          ; use Z for L
16 X=X+c                        ; add it in here
17 X=a+b*Z+X*Z^2                ; changed this formula accordingly
18 YR10
19 X=X*Y
20 YR9
21 X=X/Y

```

Below you can see the code for the **EVALXM** routine. Note the repeated use of functions **?SELECT** and **CASE** from the WARP\_Code Module. Note as well the use of two auxiliary functions from FAT-2, **EVAL#** and **TRIAGE**. The first one is a "wild-card" to help select which of the EVAL\$ functions to use, while **TRIAGE** expedites the value assignment to variables. They use R02 as repository for the index to designate the variable.

1	LBL "EVALXM"	ASCII File Evaluation	75	LBL 10	' GOTO LABEL
2	CF 21	non-stop AVIEW	76	STO 02	
3	STO 00	' SAVE X	77	CLX	disable stack lift
4	CLX	disable stack lift	78	ATOX	
5	SEEKPTA	beginning of file	79	X<> 02	
6	X<> 00	' RESTORE X	80	STO 03	
7	LBL H	' MAIN LOOP	81	CLX	
8	SF 25	' TURN ON TO HANDLE END	82	SEEKPTA	beginning of file
9	GETREC	get current record	83	LBL 13	
10	FC?C 25	' END OF FILE CLEARS FLAG	84	CLD	clear LCD
11	GTO 19	end of program	85	XROM "+REC"	advance record
12	AVIEW	; SHOW RECORD	86	XROM "A01"	puts first Chars in R00-R01
13	XROM "A01"	puts first Chars in R00-R01	87	SELECT 01	second char triage
14	SELC 01	second char (operator)	88	?CASE 58	":" for LABEL
15	?CASE 126	"Σ" for EVAL Σ	89	GTO 00	
16	GTO 10		90	GTO 13	
17			91	LBL 00	
18	?CASE 80	"P" for EVALP	92	X<> 02	
19	GTO 12		93		
20	GTO 12		94	?X= (00)	
21	?CASE 61	"=" for EVAL	95		
22	GTO 20		96	GTO 00	
23	GTO 20		97	X<> 02	
24	?CASE 32	" " for ASSIGN	98	GTO 13	
25	GTO 21		99	LBL 00	
26	GTO 21		100	X<> 03	
27	?CASE 58	":" for LABEL	101	GTO H (109)	
28	GTO H (109)	next record	102	LBL 23	COMPARISONS
29	GTO H (109)	next record	103	EVAL?	make comparison
30	?CASE 63	"?" for COMPARE	104	GTO H (109)	yes, next record
31	GTO 23		105	XROM "+REC"	advance record
32	GTO 23		106	GTO H (109)	next record
33	?CASE 82	"R" FOR RECALL	107	LBL 24	RECALL
34	GTO 24		108	STO 02	
35	GTO 24		109	CLX	
36	?CASE 83	"S" FOR STORE	110	ANUM	
37	GTO 25		111	STO 03	
38	GTO 25		112	CLX	
39	?CASE (00)	blank record case	113	RCL IND 03	
40	GTO 19	end of program	114	X<> 02	
41	GTO 19	end of program	115	TRIAGE	SF# 8
42	SYNERR	show "SYNTAX ERR"	116	GTO H (109)	next record
43	LBL 10	; SUMS & SERIES	117	LBL 25	STORE
44	-"Σ"		118	STO 02	X to R02
45	STO 02		119	CLX	
46	CLX	disable stack lift	120	LASTX	
47	-1		121	STO 04	L to R04
48	AROT		122	CLX	
49	X<> 02		123	ANUM	
50	X<> 02		124	STO 03	RG# to R03
51	EVALΣ	perform the sum	125	CLA	
52	GTO H (109)	next record	126	X<> 00	chr#(0)
53	LBL 12	; PRODUCTS	127	XTOA	
54	-"P"		128	CLX	
55	STO 02		129	RCL 02	restore X
56	CLX		130	EVALS	evaluate expression
57	-1		131	STO IND 03	store in RG#
58	AROT		132	X<> 04	recall L
59	X<> 02		133	STO L (4)	restore L
60	EVALP	perform the product	134	X<> 02	restore X
61	GTO H (109)		135	GTO H (109)	next record
62	LBL 20	; EVALUATE FORMULA	136	LBL 19	CODA
63	EVAL#	SF# 7	137	CLD	
64	GTO H (109)	next record	138	WORKFL	SF# 9
65	LBL 21	; OTHERS INCL ASSIGNMENT	139	END	
66	STO 02				
67	CLX	disable stack lift			
68	ANUM				
69	X<> 02				
70	SELECT 00	first char triage			
71	?CASE 71	"G" for GOTO			
72	GTO 10				
73	TRIAGE	SF# 8			
74	GTO H (109)	next record			

## Script Reading Programs Comparison Summary

The table below summarizes the most important differences between the two scrip reading programs included in the Formula Evaluation Module.

<i>Program</i>	<i>Condition Routine</i>	<i>Summing Routine</i>	<i>Multiplying Routine</i>	<i>Other Routines</i>
<b>EVALXM</b>	<b>EVAL?</b>	<b>EVALΣ</b>	<b>EVALP</b>	n/a
<b>EVLXM+</b>	<b>EVAL?</b>	XROM "EVLΣ+"	XROM "EVL P+"	XROM "IT\$" XROM "SV\$"

Note that besides being capable of using integral and solve commands directly in the scripts, **EVLXM+** includes the following additional Statements not supported by EVALXM:

- GoTo Forward
- GoTo Backward
- For / Next Loop
- Do / While Loop
- Solve & Integrate calls

Besides, **EVALΣ** and **EVALP** require actual integer values in their syntax, whereas **EVLΣ+** and **EVL P+** also allow an expression formula that would calculate the index value during the execution of the programs.

## Appendix 0. Evaluating Variables - Equation Solver.

As you know by now, the Formula Evaluation EVAL\_3K is a 4k-Module and can be extended with an upper page with the examples (EVAL\_APPS). The contents of the two pages are largely self-contained, so it's possible to *only load the lower page* in the calculator.

An optional ROM is available that employs the formula evaluation techniques to solving for unknown variables in equations, a.k.a. an equation SOLVER add-on. The **EVAL\_EQNS** add-on can be plugged along the Formula Evaluation module, instead of the EVAL\_APPS upper page (this saves room in the I/O bus if you don't need the provided examples anymore),

The diagram below shows these options; the top configuration with both modules alongside, and the bottom one where the EVAL\_EQNS has replaced the AVAL\_APPS:

Module	Formula Eval	Equation Solver
Lower Page	EVAL_3K	EVAL_3K
Upper Page	EVAL_APPS	<b>EVAL_EQNS</b>



## Appendix 1. Sub-functions in the auxiliary FAT

This module includes a set of low-level routines in the auxiliary FAT that can become very useful for troubleshooting and diagnostics. Some are subsets of the **EVAL\$** and **^FRMLA** functions (the two pillars of the ROM), made available as independent functions as well. Let's describe them briefly.

To execute sub-functions, you need to use either one of the launcher functions, **SF#** (using the function index) or **SF\$** (spelling the function name). **LASTF** will repeat the last executed function.

Use **CAT+** to enumerate the sub-functions. [R/S] halts the listing, [SST]/[BST] navigates the list, and [XEQ] executes it straight from the catalog.

- The underpinnings of **EVAL\$** make usage of a memory buffer, with id#6. This buffer stores all information from the formula: operators, functions, and data. During the execution of **EVAL\$** there are calls to buffer routines to push and pop values, as well as to initialize (clear) it. The available functions are: **CLRB6**, **PSHB6**, and **POPB6**

The buffer#6 header also holds the information on the currently selected buffer register (pointer in digit 9) and the destination register for the result (marker in digits 4,5,6). **POPB6** and **PSHB6** automatically decrement and increment the buffer register pointer. The buffer is 16 registers long, which should allow for any combination of data, operators, and variables in the formula string. See the following chapter for more details.

- Another group of routines have to do with advancing the character selection within the text. This allows the main code to scan all characters in the ALPHA string using a loop which is executed multiple times until the complete formula has been processed. These sub-functions are: **NXTCHR**, and **PRVCHR**.
- The next one is very helpful for error prevention and correction. **CHK\$** checks for non-matching number of open and close parenthesis, correcting the unbalance in case that close parenthesis were missing.
- The next pair **B6?** and **B7?** interrogates for existence of buffers #6 and #7 – creating them on the fly if they don't yet exist. This action is always performed by all functions accessing these buffers, but these functions provide a manual access to the functionality.
- In case you miss the HP-48SX, also included in this group is a trivial **BLIP** sound to reinforce the error messages with an acoustic warning – don't we all love those obnoxious beeps ;-)
- The last group does clever manipulation of the RTN stack addresses, popping or killing specific ones. They are used by the high-level DO/WHILE and IF.ELSE.ENDIF structures to keep track of the return-to addresses – which are temporarily stored in the RTN stack as well. These sub-functions are **XQ>GO**, **KRTN2**, **RTNS**, and **?RTN**

## Appendix 2. EVAL\$ Buffer Structure.

Buffer #6 is LIFO, sort of like a buffer stack. EVAL\$ handles the buffer interactions, as it works its way down the expression in ALPHA, character after character. EVAL\$ also does checks for empty buffer (nothing to pop from the buffer stack) or full buffer (can't push anything more onto the buffer stack). The lowest value register in the buffer set (i.e. the buffer header) has a pointer that tells which one was the last value pushed (or 0 if a clear buffer)

Buffer 6 when used for EVAL\$ purposes has three possible formats:

1. If the sign digit is 1, it is the internal function code (found in the table below) for the operation to be saved on left parenthesis operation. It would be either a dyadic or unary operation code with format "1|xxxxxxxx|CCC", where CCC is the three-digit code in the internal table.
2. If the sign digit is 2, it is the saved status of the precedence flags for last left parenthesis operation (used by right parentheses routine). It restores the flags back to the ST internal register. This is formatted on the register as "2|xxxxxxxx|xSS", where SS is the saved ST register contents.
3. If the sign digit is 0 or 9, it is the saved value (decimal) of the last saved operand. This is the usual numeric representation of a 10-digit operand (BCD). Dyadic functions save two values, unary functions save one (and most unary functions save the function number using format 1 after that because they are followed by a left parenthesis).

If the sign digit is anything else, EVAL\$ will bail out and yield an error.

Key	LCD Symbol	id#	Name	Key	LCD Symbol	id#	Name
[ + ]	+	02B	Sum	[STO]	HS	253	Hyperbolic Sin
[ - ]	-	02D	Subtraction	[RCL]	HC	243	Hyperbolic COS
[ * ]	*	02A	Product	[SST]	HT	254	Hyperbolic TAH
[ / ]	/	02F	Division	[ ][LBL]	AHS	353	Hyperbolic ASIN
[ENTER^]	^	01F	Power	[ ][GTO]	AHC	343	Hyperbolic ACOS
[Σ+]	(	028	Open Paren	[ ][BST]	AHT	354	Hyperbolic ATAN
[1/X]	)	029	Close Paren	[ ][a]	a	061	parameter
[ % ]	%	025	Percentage	[ ][b]	b	062	parameter
[RDN]	&	026	Modulus	[ ][c]	c	063	parameter
[CHS]	#	023	Negative value	[ ][d]	d	064	parameter
[ ][CHS]	ABS	315	Absolute value	[ ][e]	e	065	parameter
[ ][CAT]	IP	219	Integer Part	[ ][π]	π	050	pi
[ ][RTN]	FP	216	Fractional Part	[ 0 ]	0	030	integer
[SQRT]	Q	051	Square Root	[ 1 ]	1	031	integer
[ ][ENG]	U	055	Cube Power	[ 2 ]	2	032	integer
[EEX]	E	045	Exponential	[ 3 ]	3	033	integer
[X<>Y]	F	046	Factorial	[ 4 ]	4	034	integer
[ ][CLΣ]	G	047	Sign	[ 5 ]	5	035	integer
[ ][SCI]	R	052	Square Power	[ 6 ]	6	036	integer
[LOG]	LN	24E	Natural Log	[ 7 ]	7	037	integer
[LN]	LG	247	Decimal Log	[ 8 ]	8	038	integer
[SIN]	S	053	Sine	[ 9 ]	9	039	integer
[COS]	C	043	Cosine	[ ][X]	X	058	Variable
[TAN]	N	04E	Tangent	[ ][Y]	Y	059	Variable
[ ][ASIN]	AS	153	Arc Sine	[ ][Z]	Z	05A	Variable
[ ][ACOS]	AC	143	Arc Cosine	[ ][T]	T	054	Variable
[ ][ATAN]	AT	154	Arc Tangent	[ ][LastX]	L	04C	Variable

In summary, buffer #6 is used during the execution of the EVAL\$ functions in a dynamic manner, populating first its registers with both the arguments and the operations (coding the ALPHA characters as described above), and decoding the registers later to calculate the value of the expression written in ALPHA.

Because of this, looking into Buffer #6 at any other moment (not during the execution of EVAL\$) will typically not show any relevant information. However, during the actual execution it will have a configuration like the one represented below with a variable number of registers used depending on the actual formula being worked on.

Buffer id#	Buffer Reg	Type	Used for:
06	b7	Hex code	Operator-3
	B6	BCD value	4 <sup>th</sup> argument
	b5	Hex code	Operator-2
	b4	BCD value	3 <sup>rd</sup> argument
	b3	Hex code	Operator-1
	b2	BCD value	2 <sup>nd</sup> argument
	b1	BCD value	1 <sup>st</sup> argument
	b0	admin	Header



## Appendix 3. EVAL Applications ROM

The companion to the EVAL\_3K ROM, this is a collection of examples and applications of the different EVAL functions. Some were used as examples in the manual, but others are added in for completion. It also includes the EVAL\$-aware versions of SOLVE and INTEG that were mentioned in previous sections of this manual.

Here's a list of the included routines. Mostly they're short drivers for the core EVAL functions, which do all the heavy lifting. You're encouraged to look at the formulas used in the listings for those examples you find interesting to your needs.

Name	Description	Inputs	Author
<b>-EVAL_APPS</b>	Section Header	n/a	n/a
<b>AIINT</b>	ALPHA integer part	Value in X	Fritz Ferwerda
<b>"ARPLY"</b>	ALPHA Replace Y by X	Old in Y, new in X	Greg McClure
<b>"IT\$"</b>	Integration Routine	Interval in {Y,X}, #iter in Z	UPLE#
<b>"SV\$"</b>	Solves $f(x)=0$	Guess in X	PPC Members
<b>"AGM"</b>	Arithmetic-Geometric Mean	x, y in X, Y	Ángel Martin
<b>"d2\$"</b>	2D-Distance	P1, P2 in Stack	Martin-McClure
<b>"d3\$"</b>	3D-Distance	Prompts for Vectors	Martin-McClure
<b>"DOT\$"</b>	Dot Product 3x3	Prompts for Vectors	Martin-McClure
<b>"CL\$"</b>	Ceiling Function	Argument in X	Ángel Martin
<b>"FL\$"</b>	Floor Function	Argument in X	Ángel Martin
<b>"HRON\$"</b>	Triangle Area (Heron)	a, b, c in Y,Z,T	Ángel Martin
<b>"LINE\$"</b>	Line equation thru points	Y2,X2,Y1,X1 in Stack	Ángel Martin
<b>"NDF\$"</b>	Normal Density Function	$\mu$ in Z, $\sigma$ in Y, x in X	Ángel Martin
<b>"P4\$"</b>	Polynomial Evaluation	Prompts for Coefficients	Ángel Martin
<b>"QRT\$"</b>	Quadratic Equation Roots	Coefficients in Z, Y, X	Martin-McClure
<b>"RS\$"</b>	Rectangular to Spherical	{x, y, z} in X, Y, Z	Ángel Martin
<b>"SSR\$"</b>	Spherical to Rectangular	{R, phi, theta} in X, Y, Z	Ángel Martin
<b>-\$AND MTH</b>	Section header	n/a	n/a
<b>"NCK\$"</b>	Combinations	n in Y, k in X	Ángel Martin
<b>"NPK\$"</b>	Permutations	n in Y, k in X	Ángel Martin
<b>"KK\$"</b>	Elliptic Integral 1 <sup>st</sup> . Kind	argument in X	Ángel Martin
<b>"LEG\$"</b>	Legendre Polynomials	order in Y, argument in X	Ángel Martin
<b>"HMT\$"</b>	Hermite's Polynomials	order in Y, argument in X	Ángel Martin
<b>"TNX\$"</b>	Chebyshev's Pol. 1 <sup>st</sup> . Kind	order in Y, argument in X	Ángel Martin
<b>"UNX\$"</b>	Chebyshev's Pol. 2 <sup>nd</sup> . Kind	order in Y, argument in X	Ángel Martin
<b>"e^X"</b>	Exponential function	Argument in X	Ángel Martin
<b>"ERDOS"</b>	Erdos-Borwein constant	None	Ángel Martin
<b>"FHB\$"</b>	Generalized Faulhaber's	N in Y, x in X	Ángel Martin
<b>"HRM\$"</b>	Harmonic Number	N in X	Ángel Martin
<b>"GAM\$"</b>	Gamma function (Lanczos)	Argument in X	Ángel Martin
<b>"JNX"</b>	Bessel J integer order	n in Y, x in X	Ángel Martin
<b>"LNG\$"</b>	Log Gamma	Argument in X	Ángel Martin
<b>"PSI\$"</b>	Digamma function	Argument in X	Ángel Martin
<b>"WL\$"</b>	Lambert W Function	Argument in X	Ángel Martin
<b>"CI\$"</b>	Cosine integral	Argument in X	Ángel Martin
<b>"SI\$"</b>	Sine Integral	Argument in X	Ángel Martin
<b>"ERF\$"</b>	Error Function	Argument in X	Ángel Martin
<b>"JDN"</b>	Julian Day Number	MDY Date in {Z,Y,X}	Ángel Martin
<b>"CAL\$"</b>	Calendar Date	JND in X	Ángel Martin
<b>-SCRIPT EVL</b>	Section header	n/a	n/a
<b>EVALXM</b>	Script Evaluation	File Name in ALPHA	Greg McClure
<b>EVLXM+</b>	Enhanced Script Eval	File Name in ALPHA	Martin-McClure

<b>EVLΣ+</b>	Enhanced Sum Eval	String in ALPHA	<i>Martin-McClure</i>
<b>EVLΠ+</b>	Enhanced Product Eval	String in ALPHA	<i>Martin-McClure</i>
<b>"GMXM"</b>	Makes GAMMA Script	none	<i>Martin-McClure</i>
<b>^01</b>	Puts Chars in R00-R01	Strings in ALPHA	<i>Martin-McClure</i>
<b>+REC</b>	Advances one Record	File Name in ALPHA	<i>Martin-McClure</i>
<b>"FCT#"</b>	Factorial w/ DO.WHILE	Argument in X	<i>Ángel Martin</i>
<b>"FIB#"</b>	Fibonacci Number	Argument in X	<i>Ángel Martin</i>

A few numerical examples:-

2, ENTER^, 3, ENTER^, 4, XEQ "HRON\$"	=> 2.904737510
25, ENTER^, 2, XEQ "FHB\$"	=> 5,525.000000
25, XEQ "HRM\$"	=> 3.815958178
PI, XEQ "FL\$"	=> 3.000
PI, XEQ "CL\$"	=> 4.000
3, ENTER^, 2, ENTER^, 1, XEQ "R\$S"	=> 3.741657386,
RDN	=> 0.640522313, (in RAD mode)
RDN	=> 1.107148718 (in RAD mode)
XEQ "S\$R"	=> 3, 2, 1 in { Z, Y, X }
77, ENTER^, 27, XEQ "NCK\$"	=> 4.3838771 E20
77, ENTER^, 27, XEQ "NPK\$"	=> 4.7735466 E48
5, XEQ "WL\$"	=> 1.326724665
3, XEQ "PSI\$"	=> 0.922784334
75, XEQ "LNG\$"	=> 247.5729141
0.5, XEQ "ERF\$"	=> 0.520499888
1, ENTER^, 1, XEQ "JNX\$"	=> 0.440050584
07, ENTER^, 21, ENTER^, 1959, XEQ "JDN\$"	=> 2,436,784.000
XEQ "CAL\$"	=> 1959
RDN	=> 21
RDN	=> 7
1.4, XEQ "CI\$"	=> 0.462006566
1.4, XEQ "SI\$"	=> 1.256226762
24, ENTER^, 6 XEQ "AGM\$"	=> 13.45817148
0.5, XEQ "KK\$"	=> 1.854074677

Routine listings for NDF, LINE\$, NCK/NPK, HRMX, HRON\$, and JNX

1	LBL "NDF"			1	LBL "HRMX"	
2	"Y*Q(2*\p)*E(R((X"			2	LBL 01	
3	"-Z)/Y)/2)"			3	"Σ(1;"	
4	EVAL\$			4	AIN	
5	1/X			5	">;1/X)"	
6	END			6	XROM "EVALΣ"	
				7	RTN	
				8	GTO 01	
1	LBL "LINE\$"			9	LBL "ERDOS"	
2	"(T-Y)/(Z-X)"			10	LBL 02	
3	EVAL\$			11	"Σ(1;1/(2^X-1)"	
4	"Y-X*L"			12	">)"	
5	EVALY			13	XROM "EVALΣ"	
6	"Y="			14	RTN	
7	ARCL X(3)			15	GTO 02	
8	J-"*X"			16	LBL "e^X"	
9	X<>Y			17	LBL 03	
10	X<0?			18	"Σ(0;1;Y^X/F(X))"	
11	X=0?			19	XROM "EVALΣ"	
12	">+"			20	RTN	
13	X<>Y			21	GTO 03	
14	ARCL Y(2)			22	END	
15	AVIEW	shows line				
16	END					
1	LBL "NCK\$"			1	LBL "JNX"	
2	LBL 00			2	LET=	
3	XROM "NPK\$"			3	1	
4	"X/F(L)"			4	RDN	
5	EVAL\$			5	LET=	
6	RTN			6	2	
7	GTO 00			7	"C(b*X-a*S(X))"	
8	LBL "NPK\$"			8	E1	
9	LBL 02			9	STO 13	
10	"P("			10	0	
11	RCL Y(2)			11	PI	
12	RDN			12	XROM "ITS"	
13	-			13	PI	
14	ISG X(3)	leaves k in L		14	/	
15	NOP			15	END	
16	AIN					
17	J-";"			1	LBL "HRON\$"	
18	R^			2	"(X+Y+Z)/2"	
19	AIN			3	EVALT	
20	J-";X)"			4	"Q(T*(T-X)*(T-Y)"	
21	XROM "EVALP"			5	">*(T-Z))"	
22	RTN			6	EVALT	
23	GTO 02			7	R^	
24	END			8	END	

Routine listings for QRT\$ (updated to support complex roots) and P4\$.

1	LBL "QRT\$"		1	LBL "P4\$"	
2	LBL 00	new start	2	4	
3	LET= c		3	LBL 00	
4	3		4	"a"	
5	RDN		5	ARCL I	
6	LET= b		6	"I-=?"	
7	2		7	PROMPT	
8	RDN		8	STO IND Y	
9	LET= a		9	RDN	
10	1		10	DSE X	
11	"b^2-4*a*c<0"	discriminant	11	GTO 00	
12	XROM "EVAL?"	test for complex roots	12	"a(0)=?"	
13	"Q{ABS(b^2-4*a*c		13	PROMPT	
14	- ) ) / 2 / a"	avoids DATA ERROR	14	LET=	
15	EVAL\$		15	4	
16	FS? 04		16	RCL 01	
17	GTO 04		17	LET=	
18	"X-b/2/a"		18	3	
19	EVALY		19	RCL 02	
20	"X1="		20	LET=	
21	ARCL Y	show x1	21	2	
22	AVIEW		22	RCL 03	
23	"#X-b/2/a"		23	LET=	
24	EVAL\$		24	1	
25	"X2="		25	"e+X*(d+X*(c+X*(	
26	ARCL X	show x2	26	" -b+X*a)))"	
27	PROMPT		27	STO\$	
28	GTO 01		28	LBL 01	
29	LBL 04		29	"X=?"	
30	"#b/2/a"		30	PROMPT	
31	EVALY		31	RCL\$ (00)	
32	X<>Y		32	EVAL\$	
33	"Z1,2="		33	"P="	
34	ARCL X	shos results,	34	ARCL X	
35	- "#J"	both combined	35	PROMPT	
36	ARCL Y		36	GTO 01	
37	PROMPT		37	END	
38	END				



Routine listings for AGM\$, KK\$, CL\$, FL\$, FHB\$ and S\$R / R\$S.

1	LBL "AGM\$"			1	LBL "R\$S"	
2	LBL 00			2	"Q(X^2+Y^2)"	
3	"Q(X*Y)"			3	EVALT	
4	EVALY			4	"AT(T/Z)"	
5	"(X+L)/2"			5	EVALY	
6	EVAL\$			6	"AT(L/X)"	
7	RND			7	EVALZ	
8	X<>Y			8	"Q(L^2+T^2)"	
9	RND			9	EVAL\$	
10	X=Y?			10	RTN	
11	RTN			11	LBL "S\$R"	
12	GTO 00			12	"X*C(Y)"	
13	LBL "KK\$"			13	EVALT	
14	LBL 01			14	"X*S(Y)*S(Z)"	
15	"Q(1-X)"			15	EVALY	
16	EVAL\$			16	"X*S(L)*C(Z)"	
17	E			17	EVAL\$	
18	XROM "AGM"			18	"T"	
19	"π/2/X"			19	EVALZ	
20	EVAL\$			20	END	
21	RTN					
22	GTO 01					
23	END					
1	LBL "FHB\$"			1	LBL "FL\$"	
2	"Σ(1;"			2	CF 00	
3	X<>Y			3	GTO 00	
4	AINT			4	LBL "CL\$"	
5	-";"			5	SF 00	
6	-"X^Y)"			6	LBL 00	
7	X<.Y			7	"X-(X&"	
8	XROM "EVAL\$"			8	FS? 00	
9	END			9	-"#"	
				10	-"1)"	
				11	EVAL\$	
				12	END	

Note that **AGM\$** relies on the decimal settings of the calculator for the accuracy of the results (steps 9 and 11 perform a rounding of the X,Y values).

Formulas:  $N = \text{IDN} - 1,721,119$

if Gregorian:

$$C = \text{int} \{ (N - 0.2) / 36,524.25 \}$$

$$N' = N + C - \text{int}(C/4)$$

if Julian:

$$N' = N + 2$$

$$Y' = \text{int}[(N' - 0.2) / 365.25] \quad \rightarrow N'' = N' - \text{int}(365.25 * Y')$$

$$M' = \text{int}[(N'' - 0.5) / 30.6] \quad \rightarrow D = \text{int} [N'' - 30.6 * M' + 0.5]$$

$$\text{JDN} = \text{int} \{ \text{int} [ [ D + \text{int}(367 * X) - \text{int}(X) ] - 0.75 * \text{int}(X) ] - 0.75 * \text{int}[\text{int}(X)/100] \} + 1,721,115$$

$$\text{where: } X = Y' + (M - 2.85) / 12$$

Routine listings for ARPLXY and JDN\$ / CAL\$.

1	LBL "ARPLXY"		23	LBL "CAL\$"	
2	X<>Y		24	365.25	
3	POSA		25	LET=	
4	E		26	5	"e"
5	+		27	CLX	
6	X=0?		28	30.6	
7	GTO 00		29	LET=	
8	AROT		30	4	"d"
9	X<>Y		31	CLX	
10	SF#		32	36524.25	
11	6	RADEL	33	LET=	
12	XTOA		34	2	"b"
13	X<>Y		35	CLX	
14	CHS		36	1721119	
15	AROT		37	"Y-X+2"	
16	LBL 00		38	FS? 00	Julian Cal?
17	END		39	GTO 00	
			40	"IP((Y-X-1/5)/b)"	
			41	EVALT	
			42	"Y-X+T-IP(T/4)"	
			43	LBL 00	
			44	EVAL\$	
			45	LET=	
			46	3	"c"
			47	RDN	
			48	"IP((c-1/5)/e)"	
			49	EVAL\$	
			50	"c-IP(e*X)"	
			51	EVALT	
			52	"IP((T-1/2)/d)"	
			53	EVALZ	
			54	"IP(T+1/2-d*Z)"	
			55	EVALY	
			56	"Z<=9"	
			57	XROM "EVAL?"	
			58	FC? 04	True?
			59	GTO 04	
			60	"Z+3"	
			61	GTO 00	
			62	LBL 04	
			63	E	
			64	+	
			65	"Z-9"	
			66	LBL 00	
			67	EVALZ	
			68	CLD	
			69	END	
1	LBL "JDN\$"				
2	0.75				
3	LET=				
4	2	"b"			
5	CLX				
6	2.85				
7	LET=				
8	1	"a"			
9	RDN				
10	"X+(Z-a)/12"				
11	EVALT				
12	"b*IP(IP(T)/100)"				
13	FS? 00	Julian Cal?			
14	"2*b"				
15	EVALZ				
16	"IP(367*T)-IP(T)"				
17	" -"-b*IP(T)"				
18	EVAL\$				
19	"IP(IP(Y+X)-Z)+1"				
20	" -"721115"				
21	EVAL\$				
22	RTN				

Routine listings for SV\$ and IT\$.

1	LBL IT\$	<i>N, a, b, in stack</i>	1	LBL "SV\$"
2	STO\$ 07		2	STO\$ 07
3	ENTER^	<i>b</i>	3	LBL 00
4	EVAL\$		4	EVALZ
5	STO 11	<i>F(b)</i>	5	X<>Y
6	RDN		6	EVALT
7	"(X-Y)/Z/2"	<i>(a-b)/2N</i>	7	"Y-Z*(Y-X)/(Z-T)"
8	EVAL\$		8	EVAL\$
9	RCL\$ 07		9	FS? 10
10	RCL Y(2)	<i>a</i>	10	VIEW X(3)
11	EVAL\$	<i>F(a)</i>	11	RCL\$ 07
12	ST+ 11	<i>F(a) + F(b)</i>	12	X#Y?
13	LBL 00		13	GTO 00
14	CLX		14	END
15	E			
16	ST- T(0)	<i>decrement N</i>		
17	XEQ 09			
18	ST+ X(3)	<i>4x</i>		
19	ST+ 11	<i>add to sum</i>		
20	R^			
21	X=0?			
22	GTO 01			
23	RDN			
24	XEQ 09			
25	ST+ 11	<i>add to sum</i>		
26	GTO 00			
27	LBL 09			
28	RDN			
29	ST+ Y(2)			
30	RCL Y(2)			
31	EVAL\$			
32	ST+ X(3)	<i>2x</i>		
33	RTN			
34	LBL 01			
35	RCL 11			
36	"X*T/3"			
37	EVAL\$			
38	RCL\$ 07			
39	END			

Note that both routines use {R07-R10} to save the formula of integrand function. These routines are prepared to be used by EVALXM and EVLXM+ (data registers usage is compatible). This however, makes them unsuitable for nested execution, i.e. either using SV\$ in the integrand function, or IT\$ in the function to solve the roots for.

## Appendix. EVLXM+ Program Listing

Extended version supports DO-WHILE, FOR-NEXT, GOTO, IT\$ AND SV\$  
Non-merged instructions are listed in condensed form to save real state and for improved legibility.

01 LBL "EVLXM+" 02 CF 21 03 STO 00 04 CLX 05 SEEKPTA 06 X<> 00 07 LBL H ;new record 08 SF 25 09 GETREC 10 FC?C 25 11 GTO 19 12 AVIEW 13 XROM "^01" 14 SELECT 1 16 ?CASE 126 ;Σ 18 GTO 10 19 ?CASE 80 ;P 21 GTO 12 22 ?CASE 61 ;= 24 GTO 20 25 ?CASE 32 ;space 27 GTO 21 28 ?CASE 70 ;F 30 GTO 21 31 ?CASE 66 ;B 33 GTO 21 34 ?CASE 79 ;D 36 GTO H 37 ?CASE 58 ;: 39 GTO H 40 ?CASE 63 ;? 42 GTO 23 43 ?CASE 82 ;R 45 GTO 24 46 ?CASE 83 ;S 48 GTO 25 49 ?CASE 88 ;X 51 GTO 27 52 ?CASE 84 ;T 54 GTO 28 55 ?CASE 86 ;V 57 GTO 29 58 ?CASE 0 59 GTO 19 60 SYNERR	61 LBL 10 ;ΣUM 62 "┌Σ" 63 STO 02 64 CLX 65 -1 66 AROT 67 X<> 02 68 XROM "EVLΣ+" 69 GTO H 70 LBL 12 ;PROD 71 "┌P" 72 STO 02 73 CLX 74 -1 75 AROT 76 X<> 02 77 XROM "EVLΠ+" 78 GTO H 79 LBL 20 80 SF# 7 (EVAL#) 82 GTO H 83 LBL 21 84 SELECT 0 85 ?CASE 71 ;G 87 GTO 10 88 ?CASE 87 ;W 90 GTO 17 91 ?CASE 70 ;F 93 GTO 26 94 STO 02 95 CLX 96 ANUM 97 X<> 02 98 SF# 8 (TRIAGE) 100 GTO H 101 LBL 10 102 CLD 103 ST>RG 3 105 ATOX 106 STO 02 107 SELECT 0 108 E 109 ?CASE 70 ;F 111 GTO 11	112 ?CASE 66 ;B 114 GTO 15 115 CLX 116 SEEKPT 117 GTO 11 118 LBL 15 119 RCLPT 120 INT 121 X=0? 122 SYNERR 123 DSE X 124 LBL 00 125 SEEKPT 126 XROM "+REC" 127 XROM "^01" 128 SELECT 1 130 ?CASE 58 ;: 132 GTO 00 133 GTO 15 134 LBL 00 135 RCL 00 136 ?X= 2 138 GTO 00 139 GTO 15 140 LBL 00 141 RG>ST 3 143 GTO H 144 LBL 11 145 XROM "+REC" 146 XROM "^01" 147 SELECT 1 149 ?CASE 58 ;: 151 GTO 00 152 GTO 11 153 LBL 00 154 RCL 00 155 ?X= 2 157 GTO 00 158 GTO 11 159 LBL 00 160 RG>ST 3 162 GTO H 163 LBL 17
--	--	---

164 **EVAL?**

165 GTO 04

166 GTO H

166 LBL 04

167 CLD

168 **ST>RG** 2

170 LBL 22

171 RCLPT

172 INT

173 X=0?

174 **SYNERR**

175 DSE X

176 LBL 00

177 SEEKPT

178 **XROM "+REC"**

179 **XROM "^01"**

180 **SELCT**

181 **?CASE** 68 ; D

183 GTO 00

184 GTO 22

185 LBL 00

186 **RG>ST** 2

188 GTO H

189 LBL 23

190 **EVAL?**

191 GTO H

193 **XROM "+REC"**

194 GTO H

195 LBL 24

196 **ST>RG** 4

198 ANUM

199 STO 03

200 RCL IND 03

201 STO 02

202 **RG>ST** 4

204 **SF#** 8 (TRIAGE)

206 GTO H

207 LBL 25

208 **ST>RG** 4

210 ANUM

211 STO 03

212 CLA

213 RCL 00

214 XTOA

215 RCL 07

216 **EVAL\$**

217 STO IND 03

218 **RG>ST** 4

220 GTO H

221 LBL 26

222 **ST>RG** 3

224 ATOX

225 STO 00

226 ATOX

227 **RG>ST** 3

229 **1ST**

230 **RG>ST** 3

232 **EVAL\$**

233 STO 02

234 X<> 06

235 **SF#** 8 (TRIAGE)

237 GTO H

238 LBL 27

239 CLD

240 **ST>RG** 2

242 RCLPT

243 STO 11

244 GTO 00

245 LBL 30

246 RCLPT

247 LBL 00

248 INT

249 X=0?

250 **SYNERR**

251 DSE X

252 LBL 00

253 SEEKPT

254 **XROM "+REC"**

255 **XROM "^01"**

256 SELCT

257 **?CASE** 70 ; F

259 GTO 00

260 GTO 30

261 LBL 00

262 **STO\$** 7

264 ATOX

265 STO 00

266 CLA

267 XTOA

268 "└="

269 XTOA

270 "└+1"

271 **RG>ST** 2

273 **SF#** 7 (EVAL#)

275 **ST>RG** 2

277 **RCL\$** 7

279 CF 00

280 **2ND**

281 RCL 00

282 XTOA

283 "└<="

284 -3

285 AROT

286 **RG>ST** 2

288 **EVAL?**

289 FS? 04

290 GTO H

291 X<> 11

292 INT

293 SEEKPT

294 **XROM "+REC"**

295 X<> 11

296 GTO H

297 LBL 29

298 SF 01

299 GTO 01

300 LBL 28 ; integrate

301 CF 01

302 LBL 01

303 **ST>RG** 2

305 **SF#** 6 (RADEL)

307 **STO\$** 7

309 **1ST**

310 **RG>ST** 2

312 **EVAL\$**

313 FC? 01

314 STO 03

315 FS? 01

316 STO 04

317 **RCL\$** 7

319 SF 00

320 **2ND**

321 **RG>ST** 2

323 **EVAL\$**

324 FC? 01

325 STO 04

326 FS? 01

327 STO 05

328 **RCL\$** 7

330 FS? 01

331 CF 00

332 **3RD**

333 FS? 01

334 GTO 00

335 **EVAL\$**

336 STO 05

337 **RCL\$** 7

339 **4TH**

340 LBL 00

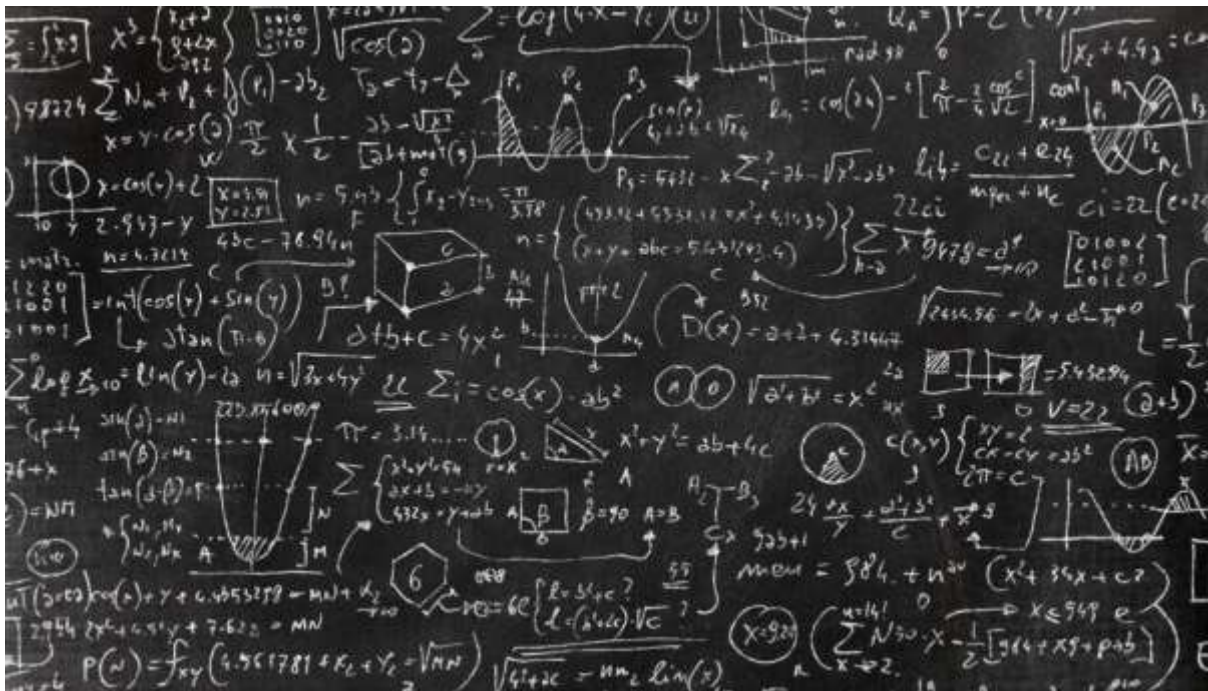
341 RG>ST 2  
343 STO 06  
344 FC? 01  
345 XROM "IT\$" 346 FS? 01

347 XROM "SV\$"  
348 STO 05  
349 RG>ST 2  
351 GTO H  
368 LBL 19

369 SF# 9 (WRKFILE)  
371 CLD  
372 END

*And finally, see the two subroutines shared by EVALXM and EVLXM+*

1	LBL "A01"	puts first Two Chars in R00-R01			
2	STO 00				
3	CLX	disable stack lift			
4	ATOX	left char to X			
5	X<> 00		11	LBL "+REC"	
6	STO 01		12	SF 25	
7	CLX	disable stack lift	13	GETREC	
8	ATOX	left char to X	14	FC?C 25	end of File?
9	X<> 01		15	SYNERR	show "SYNTAX ERR"
10	RTN		16	END	



## Appendix4. The underpinnings of DO-WHILE and IF,ELSE.ENDIF

These functions are a great example of what can be done with a good idea and a robust knowledge of the operation of the calculator. The recipe for success is a skillful manipulation of the FOCAL RTN addresses to coerce the routine flow to obey the results of the conditional evaluations, and not the sequential scheme provided by the standard FOCAL rules.

Starting with the easier one, the DO-WHILE source code is shown below.

**DO's** mission is to search for a **WHILE** statement downstream; done in the subroutine [?END0], and to push *its own location address* in the [ADR1] position of the RTN stack – so the **WHILE** code will know where to send the execution back to..

Header	A8C4	08F	"O"		does PC>RTN
Header	A8C5	004	"D"		Ángel Martín
DO	A8C6	39C	PT= 0		
	A8C7	130	LDI S&X		
	A8C8	09C	CON:		WHILE 2nd. byte
and	A8C9	058	G=C @PT,+ ←		
	A8CA	379	PORT DEP:		Search for matching WHILE
	A8CB	03C	XQ		doesn't change the PC
	A8CC	088	->A888		[?END0]
NOTFND	A8CD	0BB	JNC +23d		Show "NO_" msg
FOUND	A8CE	141	?NC XQ		get current PC address
	A8CF	0A4	->2950		[GETPC] -points at byte after DO
GO>XQ	A8D0	31D	?NC XQ		backtrack one byte
at [DO]	A8D1	0A4	->29C7		[DECAD]
h iteration	A8D2	31D	?NC XQ		backtrack one byte
r in RTN stack	A8D3	0A4	->29C7		[DECAD]
GO_XQ2	A8D4	0CC	?FSET 10		pointer in ROM?
	A8D5	02F	JC +05		yes, skip adjustment
ss	A8D6	042	C=0 @PT		PC is now at the DO step
le is zero	A8D7	0A2	A<>C @PT		pack the RAM address
	A8D8	3CA	RSHFC PT<-		so the leftmost nybble is zero
	A8D9	156	A=A+C XS		"0XXX" in A<3:0>
PRTN2	A8DA	338	READ 12(b)←		"R3 ADR2 ADR1 PCNT"
V1	A8DB	0AA	A<>C PT<-		replace PC with 0XXX
	A8DC	0FC	RCR 10		"R2 ADR1 addr R3 AD"
	A8DD	0AA	A<>C PT<-		save b's leftover in A<3:0>
	A8DE	328	WRIT 12(b)		"R2 ADR1 0XXX PCNT"
	A8DF	2F8	READ 11(a)		"ADR6 ADR5 ADR4 AD"
	A8E0	0FC	RCR 10		"ADR5 ADR4 AD ADR6"
	A8E1	0AA	A<>C PT<-		rescue leftover for a
	A8E2	2E8	WRIT 11(a)		"ADR5 ADR4 ADR3 AD"
	A8E3	3E0	RTN		done.
NOEND	A8E4	321	?NC XQ ←		Show "NO_" msg
	A8E5	10C	->43C8		[NOMSG4]
	A8E6	005	"E"		
	A8E7	00E	"N"		"NO END"
	A8E8	204	"D"		
	A8E9	1F1	?NC GO		Left!, Show and Halt
	A8EA	0FE	->3F7C		[APEREX]



At this point let's refresh our understanding of the RTN stack registers:

**h(12):**

R	3	A	D	R	2	A	D	R	1	P	C	N	T
13	12	11	10	9	8	7	6	5	4	3	2	1	0

**a(11):**

A	D	R	6	A	D	R	5	A	D	R	4	A	D
13	12	11	10	9	8	7	6	5	4	3	2	1	0

Once DO has performed its task, the program continues executing all following instructions until WHILE is reached, and [UCRUN] is called to run EVAL? as a FOCAL instruction to perform the conditional evaluation defined in ALPHA and act accordingly:

- **If true**, the execution needs to be sent back to the DO program step. This we accomplish by removing the first RTN address (*placed there by [UCRUN]*) so that the second one takes its position, hence the execution will go to DO after the RTN. The popping is done by [XQ>GO]
- **If false**, the execution should continue below the WHILE step, in which case we don't need the DO address anymore, thus ADR2 is purged off the RTN stack, shifting the higher RTN addresses (ADR5 to ADR3) down one position. Note that ADR1 is not messed with at all, as it holds the location of the WHILE program step - pushed there by [UCRUN]. The purging is done by [KADR2]

Header	A8DD	085	"E"		
Header	A8DE	00C	"L"		Check test and decide
Header	A8DF	009	"I"		
Header	A8E0	008	"H"		
Header	A8E1	017	"W"		Ángel Martín
WHILE	A8E2	178	READ 5(M)		
	A8E3	2EE	?C#0 ALL		anything in Alpha?
	A8E4	329	?NC GO		no, show "SYNTAX ERROR"
	A8E5	12A	->4ACA		[SYNERR]
PORTAL	A8E6	1B9	?NC XQ		yes, transfer to FOCAL
	A8E7	100	->406E		[UCRUN] - pushes RN addr
FOCAL	A8E8	1A7	XROM 30,21		evaluate the test condition
	A8E9	095	A7:95		EVAL?
TRUE	A8EA	1B1	GTO 00		TRUE, pop the first RTN adr
	A8EB	082	<Distance>		2 bytes
FALSE	A8EC	1B1	GTO 00		FALSE, kill 2nd. RTN and end.
	A8ED	085	<Distance>		5 bytes
POPRT1	A8EE	1A7	XROM 30,11		pop the first RTN adr (WHILE)
	A8EF	08B	A7:8B		SF# --
	A8F0	112	2		
	A8F1	015	5		XQ>GO
	A8F2	185	RTN		it'll return to "DO"
KILLRT2	A8F3	1A7	XROM 30,11		kill the 2nd. RTN adr
	A8F4	08B	A7:8B		SF# --
	A8F5	112	2		
	A8F6	016	6		KRTN2
	A8F7	1C2	END		it'll return to "WHILE"
FOCAL	A8F8	004	CHAIN - ... whatever		
FOCAL	A8F9	22F	<End od Program>		



To complete this review, see the listing for the subroutines mentioned so far:

Header	ACB0	0CF	"O"	
Header	ACB1	047	"G"	<u>Pops first RTN address</u>
Header	ACB2	03E	">"	
Header	ACB3	051	"Q"	
Header	ACB4	058	"X"	<i>Hakan Thörgren</i>
XQ>GO	ACB5	2F8	READ 11(a)	get upper register
	ACB6	0EE	C<>B ALL	save it in B
	ACB7	338	READ 12(b)	get lower register
	ACB8	0AE	A<>C ALL	save it in A
	ACB9	29C	PT= 7	field delimiter
	ACBA	3EA	LSHFA PT<-	
	ACBB	3EA	LSHFA PT<-	shift A<0:7> four nybbles left
	ACBC	3EA	LSHFA PT<-	from: "R3 ADR2 ADR1 PCNT"
	ACBD	3EA	LSHFA PT<-	to: "R3 ADR2  PCNT 0000 "
	ACBE	0AE	A<>C ALL	bringit to C
	ACBF	01C	PT= 3	
	ACC0	0CA	C=B PT<-	"R3 ADR2 PCNT  R4 AD "
	ACC1	07C	RCR 4	" R4 AD  R3 ADR2 PCNT"
	ACC2	328	WRIT 12(b)	ADR1 is gone!
	ACC3	0CE	C=B ALL	"ADR6 ADR5 ADR4 AD"
	ACC4	04A	C=0 PT<-	"ADR6 ADR5 AD 00 00 "
	ACC5	07C	RCR 4	"0000  ADR6 ADR5 AD"
	ACC6	2E8	WRIT 11(a)	update upper register
	ACC7	3E0	RTN	done.

and:

Header	A333	0B2	"2"	
Header	A334	04E	"N"	<u>Kills the 2nd. RTN adr</u>
Header	A335	054	"T"	
Header	A336	052	"R"	
Header	A337	04B	"K"	<i>Ángel Martín</i>
KRTN2	A338	338	READ 12(b)	
	A339	0E0	SLCT Q	
	A33A	2DC	PT= 13	
	A33B	0A0	SLCT P	
	A33C	11C	PT= 8	
	A33D	3D2	RSHFC P-Q	
	A33E	3D2	RSHFC P-Q	get rid of 2nd. RTN
	A33F	3D2	RSHFC P-Q	one nybble at a time
	A340	3D2	RSHFC P-Q	"0000  R3 ADR1 PCNT"
	A341	10E	A=C ALL	
	A342	2F8	READ 11(a)	get upper RTN stack
	A343	07C	RCR 4	rotate for transfer
	A344	0DC	PT= 10	delimit new field
	A345	112	A=C P-Q	puts "R4 AD" to A<13:10>
	A346	052	C=0 P-Q	wipe it off from upper
	A347	2E8	WRIT 11(a)	"0000  ADR6 ADR5 AD"
	A348	0AE	A<>C ALL	"R4 ADR3 ADR1 PCNT"
	A349	328	WRIT 12(b)	re-write-lower part
	A34A	3E0	RTN	done.

And we've left the initial subroutine for last, which is also a good seg way for the IF.ELSE.ENDIF description following next.-

The routine expects the second byte of the sought-for instruction in the G register. A successful hit consists of a match of the first byte ("171") and the second byte (in G). They are of course determined by the XROM id# and their position in the FAT.

?END0	A888	384	CLRF 0	
?END	A889	141	?NC XQ	get current PC address
	A88A	044	->2950	[GETPC] - puts adr in A<3:0>
NXTBYT	A88B	08A	B=A PT<-	byte adr in B<3:0>
	A88C	3CC	?KEY	safety abort
	A88D	360	?C RTN	
OVER?	A88E	38C	?FSET 0	was INDIF found?
	A88F	023	JNC +04	no, skip
	A890	198	C=M ALL	get ENDIF adr back
	A891	36A	?A#C PT<-	are we here again?
	A892	3A0	?NC RTN	yes, abort ELSE search!
FIRST	A893	019	?NC XQ	get next byte from A<3:0> in C<1:0>
	A894	0B4	->2D06	[NBYTAB]
	A895	08E	B=A ALL	save adr in B<3:0>
	A896	056	C=0 XS	
	A897	106	A=C S&X	put byte in A<1:0> for compares
	A898	130	LDI S&X	
	A899	0A7	CON:	WHILE 1st. byte
	A89A	366	?A#C S&X	could it be WHL_1?
	A89B	077	JC +14d	NO, keep checking
WHILE	A89C	019	?NC XQ	yes, get next byte
	A89D	0B4	->2D06	[NBYTAB]
	A89E	08E	B=A ALL	
	A89F	056	C=0 XS	clear the "1" if there
	A8A0	106	A=C S&X	save 2nd. Byte in A.X
	A8A1	39C	PT= 0	
	A8A2	098	C=G @PT,+	get sought for value
	A8A3	01C	PT= 3	watch out!
	A8A4	366	?A#C S&X	is is WHL_2?
	A8A5	14D	?NC GO	yes, WHILE found
	A8A6	032	->0C53	[SKIP1] - address left in B<3:0>
NXTBT	A8A7	06A	A<>B PT<-	no, put adr in A
	A8A8	31B	JNC -29d	and loop for next byte
NOWHL	A8A9	130	LDI S&X	upper bound
	A8AA	0CD	CON:	"CE" = X<>F _; "CF" = LBL _
	A8AB	306	?A<C S&X	trouble child?
	A8AC	3DB	JNC -05	no, next byte plz.
	A8AD	386	RSHFA S&X	move nybble right
	A8AE	130	LDI S&X	the remaining "Cx" bytes
	A8AF	00C	.END. Nybble	excluding CE and CF
	A8B0	366	?A#C S&X	
	A8B1	3B7	JC -10d	
	A8B2	3E0	RTN	search failed!

Two entry points exist: the first one at 0xA888 clears F0 and it's used by both DO and IF in the search for and ENDIF statement (that must always exist). The second entry point is only used by IF in the search for an ELSE statement (which may exist or not), a condition that sets F0 so the routine knows to do an address check – ensuring that the current address checked *does not go beyond that of the ENDIF statement found in the first pass*. This shortens the searched segment and avoids finding an ELSE *outside* of the IF.ENDIF we're working within.

When the execution encounters an IF statement the code searches for the matching ENDIF, as well as for a possible ELSE statement within the same structure. Its address is immediately pushed in the ADR2 location of the return stack. Next it performs the evaluation of the conditional in ALPHA, and depending on its result it will: (a) continue with the program step after when TRUE, or (b) branch to the ENDIF (or ELSE if it exists) when FALSE.

Header	A8FA	086	"F"			does PC>RTN
Header	A8FB	009	"I"			Ángel Martin
IF	A8FC	39C	PT= 0			
	A8FD	130	LDI S&X			
	A8FE	09F	CON:			ENDIF 2nd. byte
	A8FF	058	G=C @PT,+			
	A900	379	PORT DEP:			Search for matching ENDIF
	A901	03C	XQ			somewhere below
	A902	088	->A888			[?END0]
NOTFND	A903	29B	JNC -45d			Show "NO_" msg
FOUND	A904	0CA	C=B PT<-			get ENDIF address to C<3:0>
	A905	158	M=C ALL			safeguard in M<3:0>
	A906	388	SETF 0			flags ENDIF found!
	A907	39C	PT= 0			
	A908	130	LDI S&X			
	A909	09E	CON:			ELSE 2nd. byte
cannot exceed	A90A	058	G=C @PT,+			
	A90B	379	PORT DEP:			Search for matching ELSE
	A90C	03C	XQ			
	A90D	089	->A889			[?END]
NOTFND	A90E	043	JNC +08			ELSE not found
FOUND	A90F	046	C=0 S&X			
	A910	270	RAMSLCT			
M<3:0>	A911	06A	A<>B PT<-			put ELSE address in A<3:0>
	A912	379	PORT DEP:			Push ELSE adr+2 in RTN#1
M<3:0>	A913	03C	XQ			in case it's FALSE
	A914	0C3	->A8C3			[GO>XQ2]
	A915	043	JNC +08			CHECK CONDITION
NOTELSE	A916	046	C=0 S&X			found, it'll be used if FALSE
	A917	270	RAMSLCT			
	A918	198	C=M ALL			get ENDIF adr back
	A919	10A	A=C PT<-			put address in A<3:0>
	A91A	379	PORT DEP:			Push ENDIF adr in RTN#1
	A91B	03C	XQ			in case it's FALSE
	A91C	0BF	->A8BF			[GO>XQ]
PORTAL	A91D	1B9	?NC XQ			transfer to FOCAL
	A91E	100	->406E			[UCRUN] - pushes RN addr
FOCAL	A91F	1A7	XROM 30,21			evaluate the test condition
	A920	095	A7:95			EVAL?
TRUE	A921	1B1	GTO 00			TRUE, kill 2nd. RTN and end.
	A922	030	<Distance>			-48 bytes
FALSE	A923	1B1	GTO 00			FALSE, pop the first RTN adr
	A924	037	<Distance>			-55 bytes

Note that, like it was the case for WHILE, the conditional evaluation is done in a FOCAL code stub triggered by [UCRUN]. The TRUE/FALSE results direct the execution to the same [XQ>GO] and [KRTN2] routines but in reverse:

- TRUE now removes the ENDIF address from ADR2
- FALSE pops the first RTN adr so that the ENDIF address in ADR2 becomes ADR1

So far so good, the last piece of this puzzle is to equip ELSE with the capability to jump to the ENDIF statement, so then the TRUE branch is completed the program will skip all the instructions between ELSE and ENDIF. This requires a new search for ENDIF, i.e. a third call to [?END0] as can be seen below:

Header	A925	085	"E"	
Header	A926	013	"S"	tricky little one...
Header	A927	00C	"L"	
Header	A928	005	"E"	Ángel Martin
ELSE	A929	39C	PT= 0	
	A92A	130	LDI S&X	
F processes	A92B	09F	CON:	ENDIF 2nd. byte
cy	A92C	058	G=C @PT, +	
y done	A92D	379	PORT DEP:	Search for matching ENDIF
n (!)	A92E	03C	XQ	somewhere below
	A92F	088	->A888	[?END0]
NOTFND	A930	29B	JNC -45d	Show "NO_" msg
FOUND	A931	06A	A<>B PT<-	put address in A<3:0>
	A932	31D	?NC XQ	backtrack one
	A933	0A4	->29C7	[DECAD]
	A934	0B1	?NC GO	[DECAD] and [PUTPC]
	A935	08E	->232C	[PUTPCD]
Header	A936	086	"F"	
Header	A937	009	"I"	End of IF
Header	A938	004	"D"	Does NOTHING!
Header	A939	00E	"N"	
Header	A93A	005	"E"	Ángel Martin
ENDIF	A93B	39C	PT= 0	
	A93C	3D8	C<>ST XP	
	A93D	058	G=C, PT+	
	A93E	046	C=0 S&X	
	A93F	270	RAMSLCT	
	A940	130	LDI S&X	
e"	A941	0C6	CON: 198	higher pitch
	A942	375	?NC XQ	messes up all status bits!
	A943	058	->16DD	[TONEB]
	A944	098	C=G @PT, +	
	A945	358	ST=C XP	
	A946	3E0	RTN	

And finally, the **ENDIF** instruction – which by itself does nothing but must exist to demarcate the IF/ENDIF structure. I've added a short beep just for kicks, so the user knows the execution has completed the IF.ELSE.ENDIF structure successfully.

PS. It'll be good to expedite the execution by saving the ENDIF address in a permanent location, but such isn't a trivial proposition since there's no way to know what is going to happen within the ELSE.ENDIF branch and thus there's no way to tell what resources are going to be needed. The solution may involve using the buffer header register... to be continued?

### Even more difficult now: FOR...NEXT loops

This is of course the next logical step, that despite its assumed simplicity it has required a more involved wizardry to wedge it in the module.

The FOR...NEXT loop requires a variable and two loop pointers (beginning and end). The variable is the SELECT'ed register, and the pointers are combined in bbb.eee form as contents of such register.

In terms of the internal operation the **FOR** instruction performs a dual role: (1) storing the bbb.eee:ss control word in X into the SELECT'ed register (first execution only), done in subroutine [SELSLC], and (2) pushing its own location address in the RTN stack (for the **NEXT** statement consumption later on). This second task is identical to [DO]'s mission, thus the execution is transferred to the same point for that.

Header	A8B3	092	"R"	Saves X in SLCT and does PC>RTN
Header	A8B4	00F	"O"	
Header	A8B5	006	"F"	Ángel Martín
<b>FOR</b>	A8B6	0F8	READ 3(X)	get control word
	A8B7	070	N=C ALL	
	A8B8	379	PORT DEP:	Selects SLCT register, and puts
	A8B9	03C	XQ	header addr in Q, content in M
le with	A8BA	188	->A988	[SELSLC]
!!	A8BB	198	C=M ALL	get header content (before resetting)
small advance?	A8BC	2F6	?C#0 XS	SLCT update disable?
	A8BD	01F	JC +03	yes, skip update!
ANEW	A8BE	0B0	C=N ALL	recall control word
	A8BF	2F0	WRTDATA	put cnt'l word in SLCT reg
	A8C0	39C	PT= 0	prepare char# to search
	A8C1	130	LDI S&X	
	A8C2	0A1	CON:	NEXT 2nd. byte
	A8C3	033	JNC +06	merge w/ DO code

For the first task [SELSLC] is faced with the interesting problem of telling whether it's the first time it's being executed. We can't use CPU or user flags as semaphores for that purpose, because there's no way to know what instructions are being ran inside of the FOR...NEXT loop – which can alter any or all of them. System flags like PRIVATE status (F12) is available and could be used to this purpose (in fact that's how it was done in the first version), but the drawback is that SST-mode execution is not possible (the O/S detects the fake-private status and doesn't play ball). The stack-lift flag (F11) was also a potential candidate, but it was discarded because it gets reset by the [XRUN] routine used by NEXT to return to FOR.

The solution has been to set a marker in buffer#7's header (the one that holds the pointer to the SELECT'ed register). The marker is always zero except when **NEXT** transfers the execution back to **FOR**. Reading that digit solves the problem: the control word is only saved in the SELECT'ed reg the first time. See code lines above at 0xA8BB and 0xA8BC, with copy of the header register in M (and the marker in its XS digit) at this point.

[SELSLC] is also used by **NEXT** to read the current control word kkk.eee, to increment (or decrement) the current index kkk, and to perform the comparison to determine an exit of loop condition (kkk>=eee). The header is read into M and the marker is cleared back immediately to avoid leaving a false first-time status as result of left-over value after errors between this point and the moment of the checking in FOR.

SELSLC	A988	369	?PNC XQ	Check buffer id#7 -> header in C
	A989	124	->49DA	[CHKBF#7] - returns addr in A.X
	A98A	086	B=A S&X	save buffer addr in B.X
	A98B	0A6	A<>C S&X	
	A98C	270	RAMSLCT	
ELECT" token	A98D	038	READATA	get header content
CT update	A98E	158	M=C ALL	save header contents in M
	A98F	056	C=0 XS	
	A990	2F0	WRTDATA	clear re-select token
	A991	046	C=0 S&X	
	A992	270	RAMSLCT	
	A993	0C6	C=B S&X	get buffer addr again
	A994	268	WRIT 9(Q)	save buf addr in Q(9)
	A995	03C	RCR 3	yes, shift group
	A996	266	C=C-1 S&X	remove the padding
	A997	01B	JNC +03	if zero, replace with default
	A998	130	LDI S&X	
	A999	073	X register	defaults to X if no info
	A99A	106	A=C S&X	put in A.X for vetting
	A99B	321	?PNC XQ	Check Existence - IND, STK
	A99C	138	->4EC8	[EXISTS3] - adr in A.X
	A99D	0A6	A<>C S&X	
	A99E	270	RAMSLCT	select SELCT register
	A99F	3E0	RTN	

Let's now look into the NEXT instruction code next (sorry I couldn't resist). The first part after calling [SELSLC] is the routinary stuff to read the values, increment them and compare them: see code segment 0xA95F to 0xA97B in next page.

Depending on the comparison the execution is transfer back to the FOR statement (if kkk<eee), or to the instruction following NEXT if the limit has been reached (kkk>=eee). Note that we cover both contingencies (equal or larger than) to trap error condition cases when the user inputs bbb.eee such that bbb>eee.

The transfer back to **FOR** is handled by the [XRTN] procedure, an elaborate routine that resets F11 and manages all the O/S requirements for a subroutine return. Remember that at this point **FOR**'s location was still in the RTN stack, ok? Well, [XRTN] does all its voodoo magic and hands it out to the O/S with **FOR**'s location as Program Pointer (PC) – thus **FOR** kicks in again, saving its own address into ADR1, but *this time the control word won't be updated* since the last thing **NEXT** did (right before the call to [XRTN]) was to *mark the buffer header with the "don't update" flag* (see code segment 0xA97E to 0xA984).

Finally, the termination when the loop needs exiting is no other than a call to our known [XQ>GO] routine to pop **FOR** address off the RTN stack, since it won't be needed anymore.

Note that because FOR...NEXT doesn't involve **?EVAL**, it is an all-MCODE routine, and thus the strategy did not require using [UCRUN] to transfer the execution to FOCAL as it was the case for DO/WHILE and IF/ELSE/ENDIF – which was needed to run **?EVAL** as a FOCAL program step!



Header	A958	094	"T"	<i>Increase Cnt'l word and decide</i>
Header	A959	018	"X"	<i>whether to return to FOR</i>
Header	A95A	005	"E"	
Header	A95B	00E	"N"	<i>Ángel Martín</i>
NEXT	A95C	379	PORT DEP:	<i>Selects SLCT register, and puts</i>
	A95D	03C	XQ	<i>header addr in Q, content in M</i>
	A95E	18B	->A98B	<i>[SELSLCT]</i>
	A95F	038	READATA	<i>read SLCT register content</i>
	A960	070	N=C ALL	<i>kkk,eee; kkk&gt;=bbb</i>
	A961	2A0	SETDEC	
	A962	1E1	?NC XQ	<i>Increment Count</i>
	A963	100	->4078	<i>[INCC10]</i>
	A964	035	?NC XQ	<i>write result to register</i>
	A965	124	->490D	<i>[WRTSEL] - selects Chip0</i>
	A966	0B0	C=N ALL	<i>kkk,eee; kkk&gt;=bbb</i>
	A967	088	SETF 5	<i>Take Integer part</i>
	A968	0ED	?NC XQ	<i>kkk</i>
	A969	064	->193B	<i>[INTFRC]</i>
	A96A	0F0	C<>N ALL	<i>IP to N</i>
	A96B	084	CLRF 5	<i>take fractional part</i>
	A96C	0ED	?NC XQ	<i>0.eee</i>
	A96D	064	->193B	<i>[INTFRC]</i>
	A96E	2FA	?C#0 M	<i>check for zero (!)</i>
	A96F	023	JNC +04	<i>skip if so</i>
	A970	226	C=C+1 S&X	
	A971	226	C=C+1 S&X	<i>multiply by 1,000</i>
	A972	226	C=C+1 S&X	
ZERO	A973	10E	A=C ALL	<i>eee</i>
	A974	0B0	C=N ALL	<i>kkk</i>
	A975	36E	?A#C ALL	<i>kkk=eee?</i>
	A976	08B	JNC +17d	<i>yes, exit loop</i>
	A977	2BE	C=-C-1 MS	<i>-eee</i>
	A978	000	NOP	<i>let carry settle</i>
	A979	01D	?NC XQ	<i>kkk-eee</i>
	A97A	060	->1807	<i>[AD2-10]</i>
	A97B	260	SETHX	
	A97C	2FE	?C#0 MS	<i>is eee &gt; bbb ?</i>
	A97D	057	JC +10d	<i>no, exit loop</i>
	A97E	278	READ 9(Q)	<i>get buffer header addr</i>
	A97F	270	RAMSLCT	<i>select it</i>
	A980	038	READATA	<i>get header content</i>
	A981	2B6	C=-C-1 XS	<i>SELCT update disable "flag"</i>
	A982	2F0	WRTDATA	<i>mark the header!</i>
	A983	046	C=0 S&X	
	A984	270	RAMSLCT	
	A985	00D	?NC GO	<i>no, force a FOCAL return to FOR</i>
	A986	09E	->2703	<i>[XRTN] = will set F11</i>
REPEAT	A987	260	SETHX	
	A988	3AD	PORT DEP:	<i>yes, kill the [FOR] RTN addr</i>
	A989	08C	GO	<i>and merrily go on...</i>
	A98A	0B5	->ACB5	<i>[XQ&gt;GO]</i>

So, there you have it - the underpinnings of the BASIC-like instructions explained in all gory-detail. If nothing else, it'll be very helpful for me the next time I need to revise the code, but I also Hope it was of interest to you as well.

## Appendix5. AOS Simulator

Written by Greg McClure, this FOCAL program was first released in the GJM ROM and is added here for completion.

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. So "(" and "\*" are performed before "+", etc).

### B.1 AOS Overview

The Algebraic Operating System emulator is designed to act like non-RPN 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

[A]: AOS +	[B]: AOS -	[C]: AOS *	[D]: AOS /	[E]: AOS ^
[F]: AOS (	[G]: AOS )			[J]: 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 Annun.s)	Annunciators (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 +	01234	1.0000
2	G	2 ), + performed	01234	3.0000
	C	*	01234	3.0000
	F	(, * with value saved	01234	3.0000
3	D	3 /	01234	3.0000
4	G	4 ),/ performed, * with value recalled	01234	0.7500
	A	+, * performed	01234	2.2500
	F	(	01234	2.2500
5	E	5 ^	01234	5.0000
	F	(, ^ with value saved	01234	5.0000
1	D	1 /	01234	1.0000
2	G	2 ), / performed, ^ with value recalled	01234	0.5000
	G	), ^ performed, + with value recalled	01234	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 annunciators show which operations and how many stack registers will be stored (only one register is required for the operations saved).