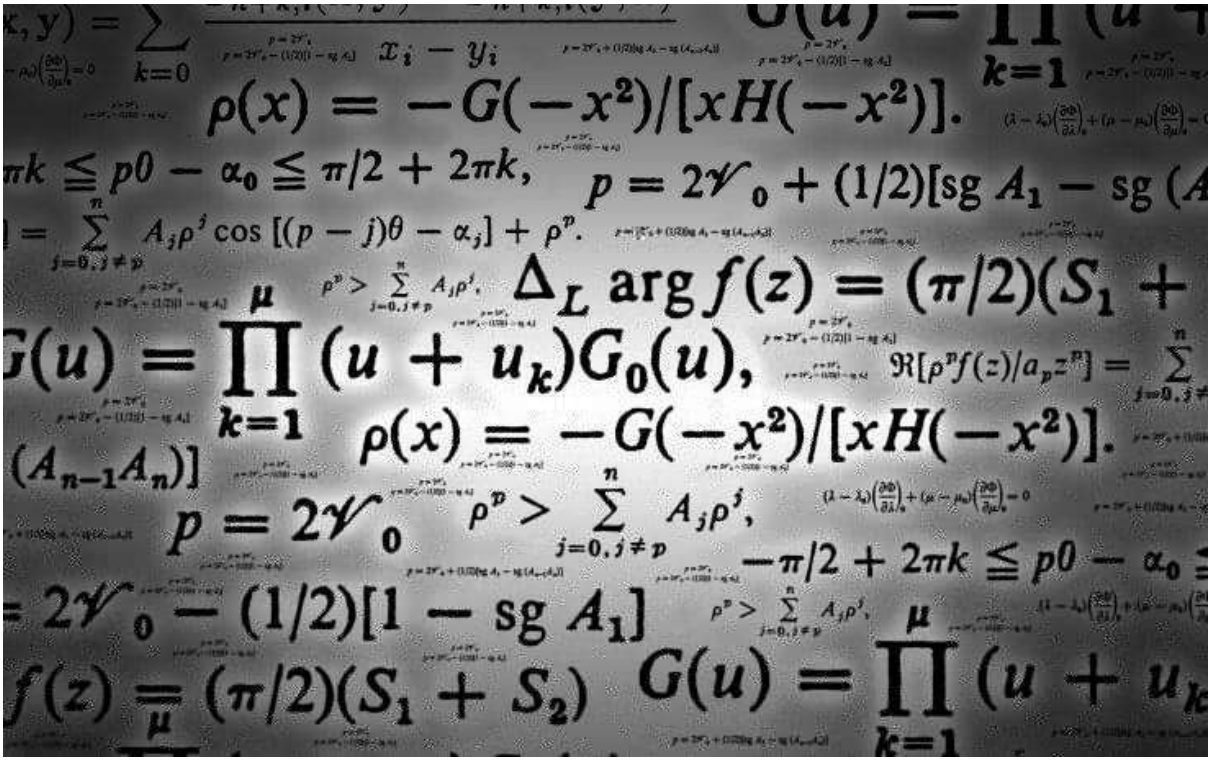
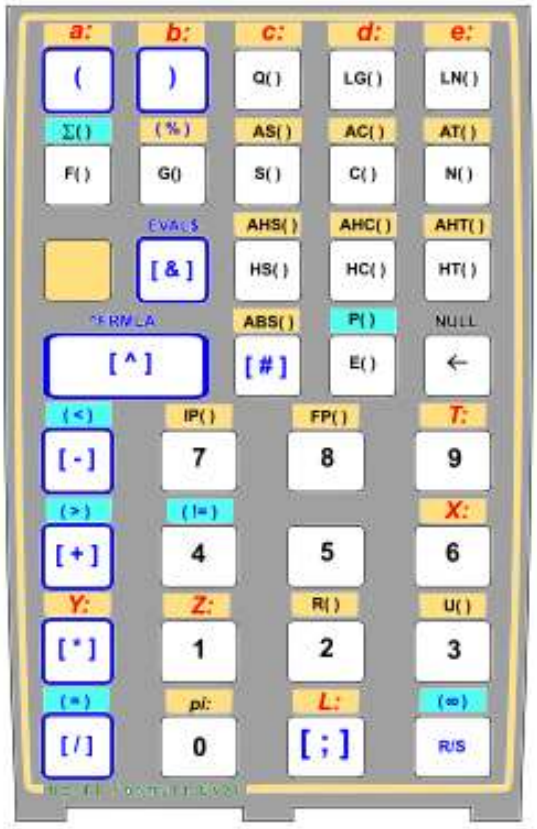
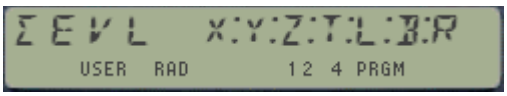
FORMULA EVALUATION ROM

HP-41 Module



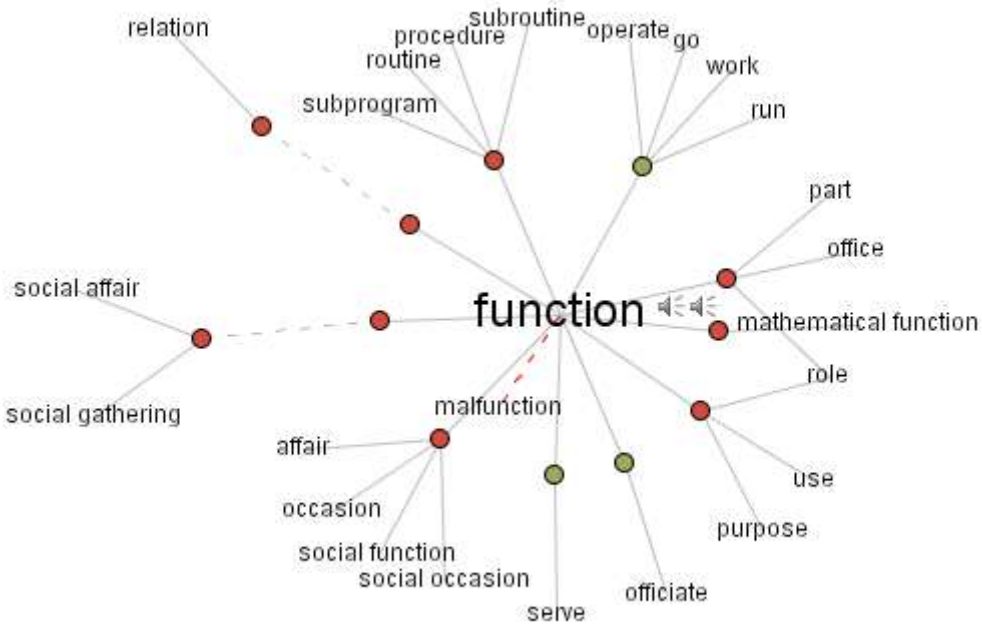




Written & Programmed by
Greg McClure and Ángel Martin

Revision VF++, May 2024

This compilation revision 1.7.3



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

FORMULA_EVALUATION-VF++

Table of Contents

1. Towards a Visual FOCAL Language: VF++
   a. Teaching new tricks to an ol' dog 4
   b. Scope, Intent and Dependencies 4
   c. Module function Summary 6

2. Syntax and Rules of Engagement
   a. Variables, constants and parameters 9
   b. Formula Entry remarks 9
   c. Formula Evaluation rules 10
   d. Syntax Table and keyboard Overlay 11
   e. Chained evaluations & Error handling 14
   f. EVAL Launcher and Alpha to Memory 15
   g. Other Utility functions 17

3. Example Programs
   a. Vector distances and Dot product 19
   b. Polynomial Evaluation using Honer's method 20
   c. Orthogonal Polynomials: Legendre, Hermite, and Chebyshev's . . 21
   d. Real Roots of Quadratic Equation 22
   e. Solve and Integrate Reloaded 23
   f. Use of EVAL$ with FINTG and FROOT 24
   g. Lambert Function 25

4. EVAL$ Advanced Applications
   a. Advanced test comparisons with EVAL? 26
   b. Evaluating Sums & Series with EVALΣ 28
   c. Evaluating Products with EVALP 29
   d. Appendix 1. Sub-functions in the auxiliary FAT 48
   e. Appendix 2. Eval$ Buffer Structure 49

5. VF++ Conditional Structures 33
   a. WHILE we're at it: Putting EVAL? to work 34
   b. What IF ?; Getting EVAL? money's worth 36
   c. Even more difficult: FOR…NEXT loops 42
   d. SELECT-CASE Structures 44

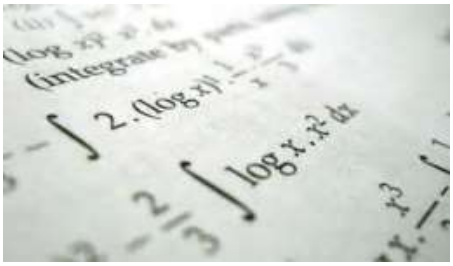
6. Eval_APPS Companion ROM 51
   a. Scripting Language facility using X-Mem 58

Appendix4. MCODE Underpinnings of VF++ Structures 71

Formula EvaluationROM

Visual FOCAL++

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.

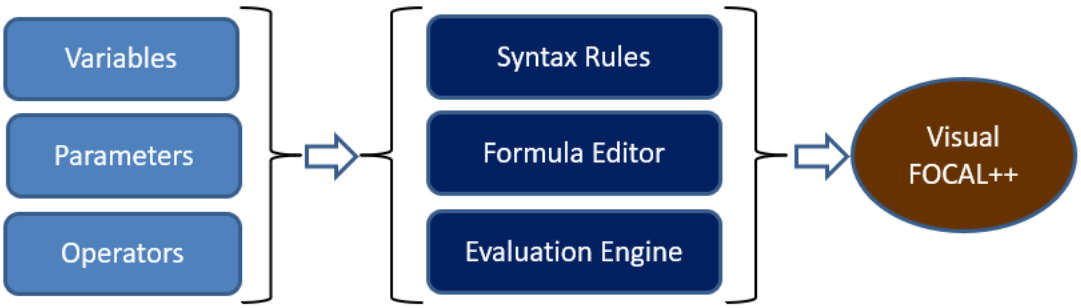
It is generally accepted that Symbolic Algebra and CAS are well beyond the scope of a venerable machine like the HP-41, quickly approaching40-year-old architecture and design. Some pioneering attempts were made in the old days, but their practical applicability (and very slow performance) would render 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. Also remember that supporting all math are the 13-digit OS routines doing the number crunching.

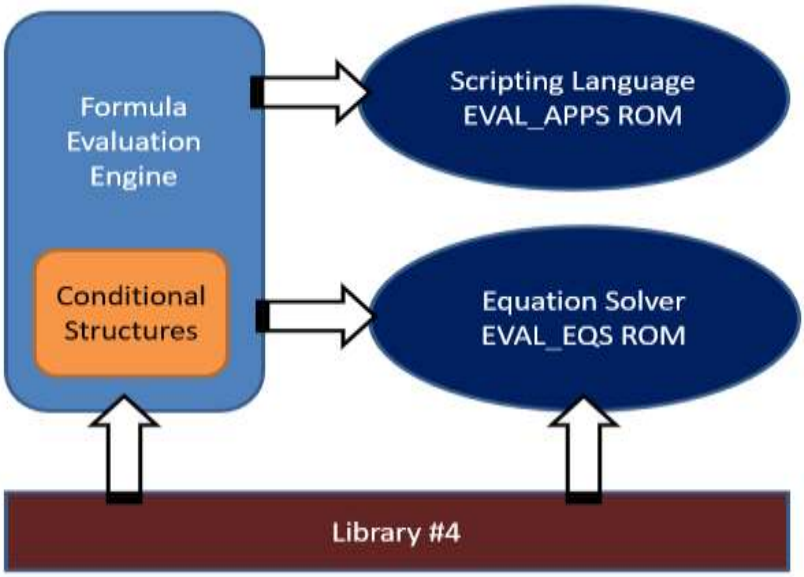
The initial design had very modest goals but was soon enough extended to include a comprehensive set of functions and operations, only restricted by the inherent limitations of the LCD display, the keyboard and other design aspects. **EVALΣ** and **EVALP** have added support for *Direct evaluation of formulas with sums and products,* and **EVAL?** provided a *general-purpose conditional testing* based on expressions combining multiple variables and math logic between them.



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

Subsequent revisions added to the mix an intriguing set of new 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. These were followed by daring implementations of FOR…NEXT and SELECT/CASE/ENDSLCT structures to complete the set of Conditional Structures.

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. See the diagram below for a conceptual summary of the interdependencies between the VF++ plug-in modules



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.

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 !

Formula Evaluation ROM – Function Summary

The table below lists all functions available in the module. The Main FAT section comprises 39 functions, while the Auxiliary FAT section adds another set of 32 functions. All of them are programmable and directly accessible by the user.

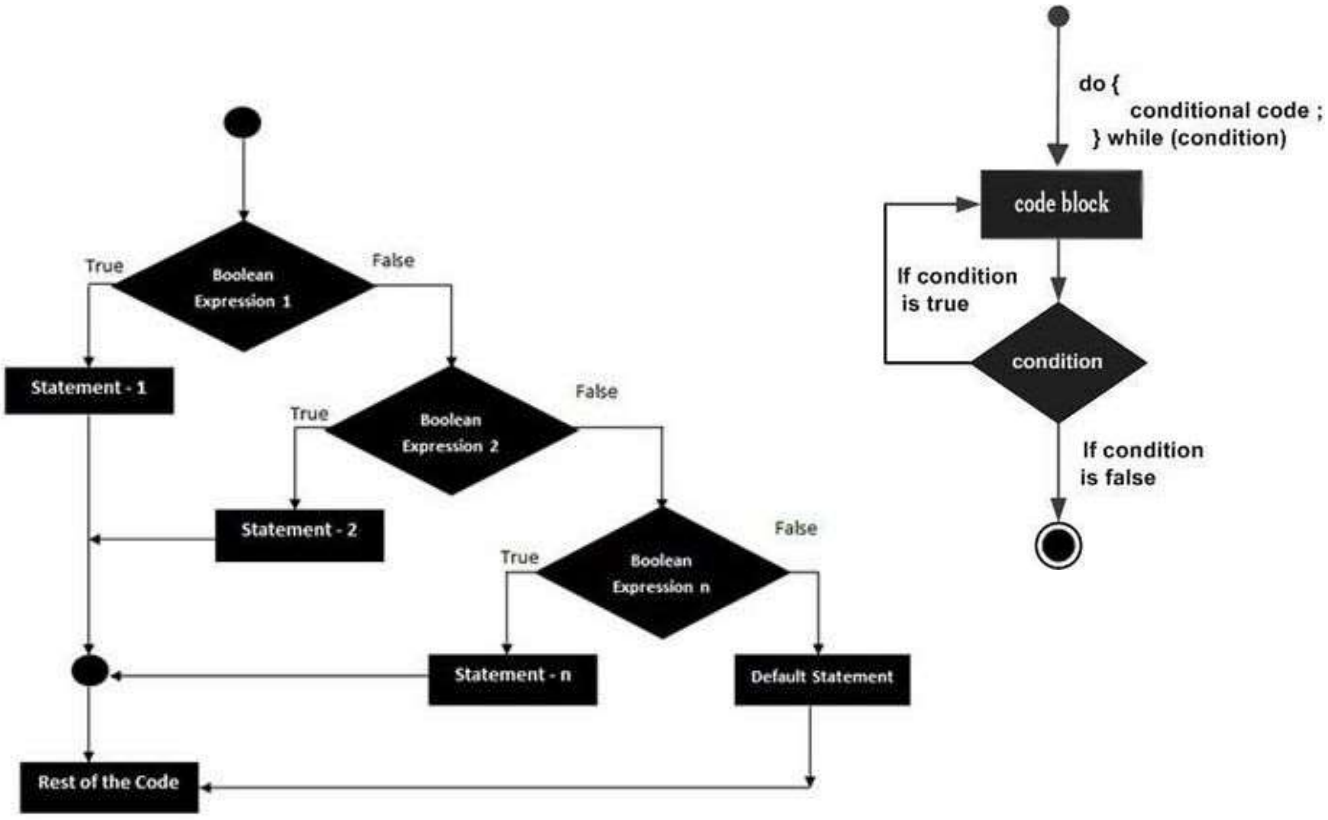
#	Name	Description	Input	Author
00	**-FORM EVAL+**	Section header	n/a	*n/a*
01	**^FRMLA _**	Enters Formula in ALPHA	Uses Custom Keyboard	*Ángel Martin*
02	**EVAL$**	Evaluates Formula -> X	Expression in ALPHA	*Greg McClure*
03	**EVALY**	Evaluates Formula ->Y	Expression in ALPHA	*Martin-McClure*
04	**EVALZ**	Evaluates Formula ->Z	Expression in ALPHA	*Martin-McClure*
05	**EVALT**	Evaluates Formula ->T	Expression in ALPHA	*Martin-McClure*
06	**EVALR _ _ _**	Evaluates -> Data Register	Expression in ALPHA	*Martin-McClure*
07	**GET= _**	Recalls parameter value	a,b,c,d,e in prompt	*Ángel Martin*
08	**LET= _**	Sets Parameter Value	a,b,c,d,e in prompt	*Ángel Martin*
09	**SHOW= _**	Shows Parameter Value	a,b,c,d,e in prompt	*Ángel Martin*
10	**SWAP= _**	Swaps Parameter and X	a,b,c,d,e in prompt	*Ángel Martin*
11	**SF# _ _ _**	Sub-function by index	sub-fnc. Index#	*Ángel Martin*
12	**SF$ _**	Sub-function by Name	sub-fnc. Name	*Ángel Martin*
13	**A-PM**	ALPHA to Program Memory	String in ALPHA	*Ángel Martin*
14	**ΣEVL _**	Eval Launcher	Prompts for destination	*Ángel Martin*
15	**-EVAL$ FNS**	Section header	n/a	*n/a*
16	**RCL$ _ _**	Recalls Formula to ALPHA	Prompts for Rg#	*Ángel Martin*
17	**RG>ST _ _**	Registers to Stack	Prompts for Re#	*Ángel Martin*
18	**ST<>RG _ _**	Swaps Stack and Regs	Reg# in Prompt	*Ken Emery*
19	**ST>RG _ _**	Stack to Regs	Prompts for Rg#	*Ken Emery*
20	**STO$ _ _**	Stores Formula in Memory	Prompts for Rg#	*Ángel Martin*
21	**SWAP$ _ _**	Swaps Alpha and Regs	prompts for Rg#	*Ángel Martin*
22	**"EVAL?"**	Evaluates Boolean Tests	Expressions in ALPHA	*Ángel Martin*
23	**EVALΣ**	Sums and Series	Expression in ALPHA	*Ángel Martin*
24	**EVALP**	Products	Expression in ALPHA	*Ángel Martin*
25	**LEFT$**	Extracts Left text	#Chars in X	*Ross Colling*
26	**RIGHT$**	Extracts right text	#Chars in X	*Ross Colling*
27	**SWAP$ _ _**	Swap ALPHA and Regs	Reg# in prompt	*Ángel Martin*
28	**DO**	Begins While Loop	WHILE statement below	*Ángel Martin*
29	**WHILE**	Ends While Loop	Expression in ALPHA	*Ángel Martin*
30	**IF**	Begins IF group	Expression in ALPHA	*Ángel Martin*
31	**ELSE**	Branches IF	ENDIF statement below	*Ángel Martin*
32	**ENDIF**	Ends IF group	none	*Ángel Martin*
33	**FOR _ _**	Begins For/Next loop	Bbb.eee in X	*Ángel Martin*
34	**NEXT _ _**	Ends For/Next loop	none	*Ángel Martin*
35	**SELECT _ _**	Opens SELECT Structure	Prompts for Reg#	*Ángel Martin*
36	**CASE _ _ _**	Individual CASE option	Prompts for Value	*Ángel Martin*
37	**CASELSE**	Unconditional Clause	none	*Ángel Martin*
38	**ENDSLCT**	Closes SELECT Structure	none	*Ángel Martin*
0	**-AUX FNS**	Section header	n/a	*n/a*
1	**FILL**	Fills Stack w/ X-value	value in X	*J.D. Dodin*
2	**SKIP**	Skips Next PRGM Line	Program code	*Erik Blake*
3	**EVALb _**	Evaluates -> Buffer Register	Expression in ALPHA	*Ángel Martin*
4	**EVALL**	Evaluates Formula -> L	Expression in ALPHA	*Martin-McClure*
5	**EVAL#**	EVAL by index	Index in R00	*Greg McClure*
6	**LADEL**	Left ALPHA delete	Text in ALPHA	*Ross Colling*
7	**RADEL**	Right ALPHA delete	Text in ALPHA	*Ross Colling*

#	Name	Description	Input	Author
8	TRIAGE	Variable assignment	ASCCI file record	*Martin-McClure*
9	WORKFL	Current File Name	Appended to ALPHA	*Sebastian Toleg*
10	CLRB6	Clear Buffer#6	Buffer#6 in Memory	*Greg McClure*
11	CHK$	Checks Syntax	Expression in ALPHA	*Ángel Martin*
12	TST$	Test ALPHA operators	{ "!=", "=". "<", ">"}	*Ángel Martin*
13	PSHB6	Push X to Buffer#6	Data in X	*Greg McClure*
14	POPB6	Pop data from Buffer#6	Data in buffer reg	*Greg McClure*
15	NXTCHR	Get Next Char	Text in ALPHA	*Greg McClure*
16	PRVCHR	Get Previous Char	Text in ALPHA	*Greg McClure*
17	B7>ST	Copies buffer to Stack	None	*Ángel Martin*
18	ST>B7	Copies Stack to Buffer	None	*Ángel Martin*
19	BLIP	Make a Sound	None	*Ángel Martin*
20	B6?	Buffer #6 Check	Data in I/O	*Greg McClure*
21	B7?	Buffer#7 Check	Data in I/O	*Ángel Martin*
22	ΣDGT	Sum of Mantissa Digits	Number in X	*Ángel Martin*
23	ZOUT	Shows Complex value	Re in X, Im in Y	*Ángel Martin*
24	CAT+ _	Sub-function CATalog	R/S, SST, BST. XEQ	*Ángel Martin*
25	XQ>GO	Pops the first RTN addr	Skips the 1st. return	*HåkanThörngren*
26	DRTN2	Duplicate 2nd RTN addr	Overwrites 1st RTN	*Ángel Martin*
27	KRTN2	Kills 2nd. RTN addr	Skips the 2nd. Return	*Ángel Martin*
28	?RTN	Tests for pending RTN	Skips next line if False	*Doug Wilder*
29	RTNS	Number of pending RTN	Data in RTN stack	*Ángel Martin*
30	DTST	Display Test	none	*Chris Dennis*
32	$KY?N _	Bulk Key Assignments	Prompts Y/N, Cancel	*HP Co.*

Additionally, the EVAL_APPS ROM has a library of pre-programmed applications, as follows:

00	-EVAL APPS	Section header	n/a	*n/a*
01	AINT	ALPHA integer part	Value in X	*Fritz Ferwerda*
02	"ARPXY"	Alpha Replace Y by X	Pos in Y, Chr$ in X	*Greg McClure*
03	"IT$	Integrates	[a,b] and N in stack	*Ángel Martin*
04	"SV$"	Solves f(x)=0	Guess in X	*PPC Members*
05	"AGM"	Arithm-Geom. Mean	x, y in X, Y	*Ángel Martin*
06	"d2$"	2D-Distance	P1, P2 in Stack	*Martin-McClure*
07	"d3$"	3D-Distance	Prompts for Vectors	*Martin-McClure*
08	"DOT$"	Dot Product 3x3	Prompts for Vectors	*Martin-McClure*
09	"CL$"	Ceiling Function	Argument in X	*Ángel Martin*
10	"FL$"	Floor Function	Argument in X	*Ángel Martin*
11	"HRON$"	Triangle Area (Heron)	A, b, c in Y,Z,T	*Angel Martin*
12	"LINE$"	Line equation thru 2 points	Y2,X2,Y1,X1 in Stack	*Angel Martin*
13	"NDF$"	Normal Density Function	μ in Z, σ in Y, x in X	*Ángel Martin*
14	"P4$"	Polynomial Evaluation	Prompts for Coefficients	*Ángel Martin*
15	"QRT$"	Quadratic Equation Roots	Coefficients in Z, Y, X	*Martin-McClure*
16	"R$S"	Rectangular to Spherical	{x, y, z} in X, Y, Z	*Ángel Martin*
17	"S$R"	Spherical to Rectangular	{R, phi, theta} in X, Y, Z	*Ángel Martin*
18	-$AND MTH	Section header	n/a	*n/a*
19	"KK$"	Elliptic Integral 1st. Kind	argument in X	*Ángel Martin*
20	"NCK$"	Combinations	n in Y, k in X	*Ángel Martin*
21	"NPK$"	Permutations	n in Y, k in X	*Ángel Martin*
22	"LEG$"	Legendre Polynomials	order in Y, argument in X	*Ángel Martin*
23	"HMT$"	Hermite's Polynomials	order in Y, argument in X	*Ángel Martin*
24	"TNX$	Chebyshev's Pol. 1st. Kind	order in Y, argument in X	*Ángel Martin*
25	"UNX$"	Chebyshev's Pol. 2nd. Kind	order in Y, argument in X	*Ángel Martin*
26	"e^X"	Exponential function	Argument in X	*Ángel Martin*

#	Name	Description	Input	Author
27	**"ERDO$"**	Erdos-Borwein constant	None	*Ángel Martin*
28	**"FHB$"**	Generalized Faulhaber's	N in Y, x in X	*Ángel Martin*
29	**"HRM$"**	Harmonic Number	N in X	*Ángel Martin*
30	**"GAM$"**	Gamma function (Lanczos)	Argument in X	*Ángel Martin*
31	**"JNX$"**	Bessel J integer order	n in Y, x in X	*Ángel Martin*
32	**"LNG$"**	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$"**	Calendar Date	JND in X	*Ángel Martin*
40	**-SCRIPT EVL**	Section Header	n/a	*n/a*
41	**"EVALXM"**	Evaluates an XM File	ASCII File Script	*Greg McClure*
42	**"EVLXM+"**	Executes Script File	File Name in ALPHA	*Greg McClure*
43	**1ST**	1st. Position	Program usage	*Greg McClure*
44	**2ND**	2nd Position	Program usage	*Greg McClure*
45	**3RD**	3rd Position	Program usage	*Greg McClure*
46	**4TH**	4th Position	Program usage	*Greg McClure*
47	**"EVLΣ+"**	Enhanced EVALΣ	Formula in ALPHA	*Martin-McClure*
48	**"EVLP+"**	Enhanced EVALP	Formula in ALPHA	*Martin-McClure*
49	**"GMXM"**	Makes GAMMA Script	none	*Martin-McClure*
50	**^01**	Puts chars in R00-R01	String in ALPHA	*Martin-McClure*
51	**+REC**	Advance File Record	Selected XM File	*Martin-McClure*
52	**"FCT#"**	Factorial using Do/While	Argument in X	*Ángel Martin*
53	**"FIB#"**	Fibonacci using Do/While	Argument in X	*Ángel Martin*
54	**"ULAM$**	Collatz' Conjecture	Argument in X	*Ángel Martin*



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 π, 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 MOD(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 modules 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 ;-)

<u>In summary:</u>

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

Key	LCD Symbol	Name
[ ][SCI][<-]	Rn	Data Register {R0-R9}
[ ][a]	a	parameter
[ ][b]	b	parameter
[ ][c]	c	parameter
[ ][d]	d	parameter
[ ][e]	e	parameter
[ ][CLΣ]	F	parameter
[ ][π]	π	pi
[0]	0	integer
[1]	1	integer
[2]	2	integer
[3]	3	integer
[4]	4	integer
[5]	5	integer
[6]	6	integer
[7]	7	integer
[8]	8	integer
[9]	9	integer
[ ][X]	X	Variable
[ ][Y]	Y	Variable
[ ][Z]	Z	Variable
[ ][T]	T	Variable
[ ][LastX]	L	Variable

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

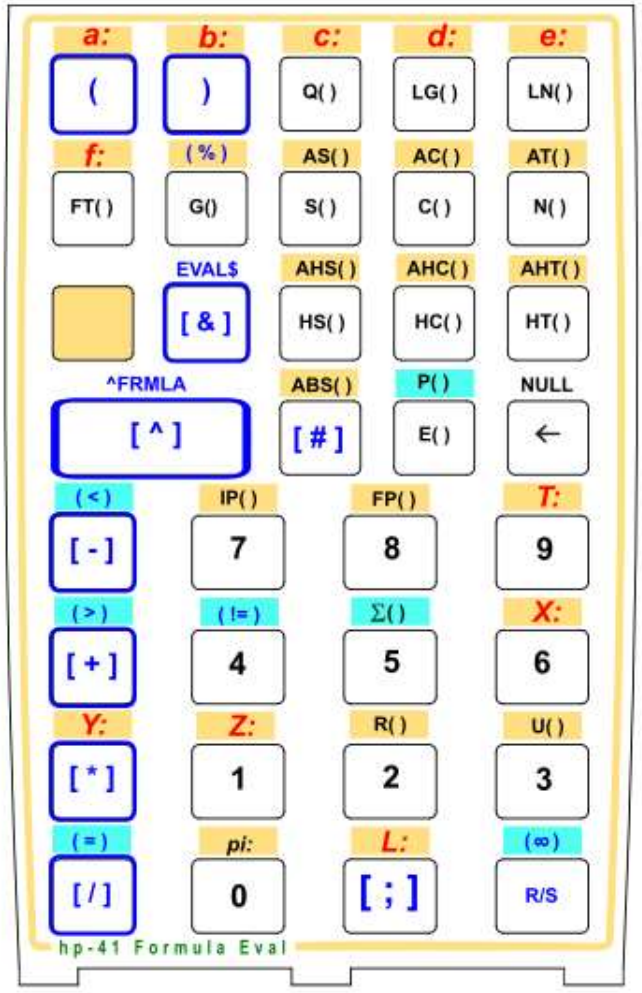
Remember the precedence rules as covered in the previous paragraphs; some will take you a little to get used to but very soon you'll feel comfortable and be putting them to its paces.

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. Usethe 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

Note that **FRMLA^** is not programmable, thus when editing a program the expression entered in ALPHA will be added as text lines steps performing an automated transfer to program memory (using the function **A-PM** under the scenes; see description in a later section.

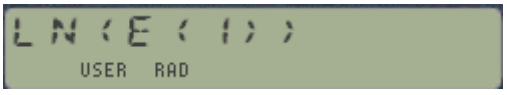
Show Me. (Missourians rejoice!)

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:

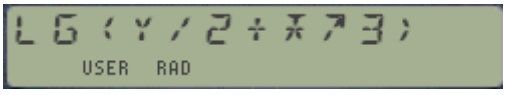
X=1, Y=2, Z=3, T=4. Then **EVAL$** returns the following results:

= 10.0000000000
easy does it….

= 1.000000000
a more rugged test!



= 1.0000000000
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

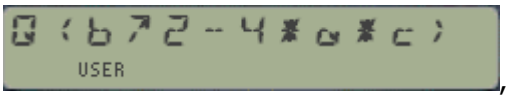
The Quintuple Twins. Chained Evaluations

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 **EVALL** to choose from, depending on the cases. Note that all except EVALL (for obvious reasons) *will save the previous contents of the destination register in LastX*– which then it effectively 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 thereal 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 =>

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.

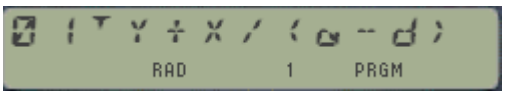
Entering Formulas in program memory with **A-PM**

No doubt using **^FRMLA** is a powerful and convenient way to enter formula expressions in the ALPHA register, and now it is also capable of entering those expressions in program memory by itself. To achieve that it triggers behind the scenes the function **A-PM** – which transfers 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, switch to program mode and execute **^FRMLA**. Since it isn't programmable it'll prompt for the text string to be entered. The transfer will occur automatically when you press R/S or ALPHA to terminate the formula string. Note that **A-PM** 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:"

" 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:

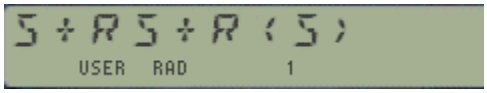
for **EVALR**_ _ _ the prompt specifies the data register number, and
for **EVALb**_ the buffer register letter (a – f) – or number (1 – 6) in program use.

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 main FAT entries.

Supporting Data Registers {R00 to R09} as data variables.

Revision VF+ adds support for the first ten data registers in memory, that is {R00 to R09}, as variables for your formulas. The syntax for these is a capital "R" followed by the one-digit register number, i.e. from "R0" to "R9". The way to enter these using **^FRMLA** is a work-around of the Square Power shortcut: [SHFT] [2] puts "R(" on the LCD, so all you need now is hit the back arrow to remove the left parenthesis, and type the index number using the numeric pad.

Example: Calculate the sum of 5+5^2+5^3, storing 5^3 in R05 previously



EVAL$ quickly returns with the result:

125, STO 05, **^FRMLA** "5+R5+R(5)", **EVAL$** => 155.000000

The data registers can therefore be part of your formulas as well, and thus numeric values from them are used as input just by typing their coded number. Furthermore, you can user the **EVALR** function to store the result in any data register of choice, a very clean and convenient way to do register math not altering the stack or requiring the infamous RCL/STO combination.

Example: Put the sum of R01, R02, and R03 in data register R10

"R1+R2+R3", **EVALR** 10 => sum in R10, previous R10 value in LastX

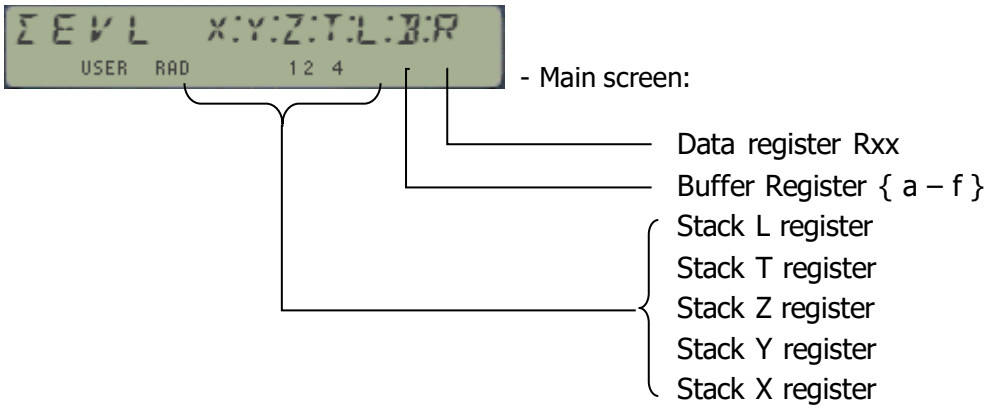
Example: Calculate R06*(1+R05) and using **EVALR** save the result in R07:

plus:

So there you have it, an even more seamless integration between the classic data storage locations and the new formula-driven operations. Note however that **EVALΣ**, **EVALP** and **EVAL?** make internal usage of data registers {R00-R10}, thus the data registers syntax will conflict in these cases.

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

Revision 3K ads 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**

Remember that *the formula syntax will automatically trigger* the **EVALP, EVALΣ**, and **EVAL?** functions if needed when selecting any of the available choices -overriding so the nominal destination - Thus they don't need additional choices in the launcher

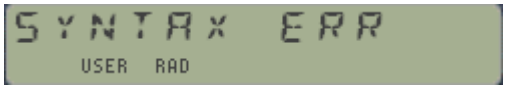
Caveat Emptor: Error Handling

The EVAL$ functions do a very reasonable job at error handling, but as any cost-effective software implementation cannot be 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 practical 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) registersused by **STO$**, **RCL$** since this will change the contents of the registers and **RCL$** will not be able to restore the alpha string correctly. 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 the5 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. Theyuse 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**.

- **FILL** is a sweet & short routine which basically fills the stack with the value in X. So it is equivalent to **SHFL** "XXXXL" in the **WARP_Core** module. This short routine was first published by J.D. Dodin, one of the advanced capabilities pioneers.

- **LET=**, **GET=**, **SHOW=**, and **SWAP=** work on the buffer register variables {a, b, c, d, e, F, and G }. Use them to assign values to them, recall their current value to X, exchange it with X value, or just to view their current value – I let you figure out which function is for each action ;-)

  These functions operate on the individual registers of the buffer, but you can also use the pair below for a group action using the complete stack as data source.

- **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** s a very good approach to assign values to buffer variables. The reverse direction **B7>ST** populates the stack with the values in those variables.

  The table below shows the actual correspondence between the stack and buffer registers.

Buffer id#	Buffer Reg	Type	Used for:
07	b6	BCD value	n/a / "F"
	b5	BCD value	X-Reg / "e"
	b4	BCD value	Y-Reg / "d"
	b3	BCD value	Z-Reg / "c"
	b2	BCD value	T-Reg / "b"
	b1	BCD value	L-Reg / "a"
	b0	admin	Header

The table below shows the indexes needed for the non-merged instructions described above.

Argument	Shown as:
100	00
101	01
102	A
103	B
104	C
105	D
106	E
107	F
108	G
109	H
110	I
111	J

Argument	Shown as:
112	T
113	Z
114	Y
115	X
116	L
117	M
118	N
119	O
120	P
121	Q
122	|-
123	a

Argument	Shown as:
124	b
125	c
126	d
127	e
128	IND 00
129	IND 01
130	IND 02
131	IND 03
132	IND 04
133	IND 05
134	IND 06
135	IND 07

(!) 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.

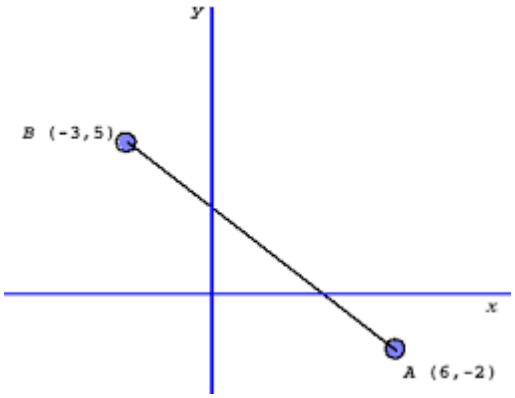
Example Programs.

1. Vector Distances, and Dot Product.

Three more easy examples follow (included in the EVAL_APPS ROM) to calculate the distance between two points, (2D and 3D cases) 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:  Q((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 **ST>B7**, 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:

```
d3:Q(R(X-e)+R(Y-d)+R(Z-c))
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:32.000000000`

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)"	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	"XZYTL"
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 (23 characters exactly) and proceed to evaluate it:

^FRMLA	=>	e + X * (d + X * (c + X * (b + X * a)))
2,**EVAL$**	=>	57.000000000
-2, **EVAL$**	=>	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$"		21	3	
2	4		22	RCL 03	
3	LBL 00		23	LET=	
4	"a"		24	2	
5	ARCLI		25	RCL 04	
6	"|-=?"		26	LET=	
7	PROMPT		27	1	
8	STO IND Y		28	"e+X*(d+X*(c+X*("	
9	RDN		29	"|-b+X*a)))"	
10	DSE X		30	STO$	
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 1st. Kind	Tn(x) = 2x.Tn-1(x) - Tn-2(x)	T0(x) = 1	T1(x) = x
Chebyshev 2nd. 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$" => 16,598,444.019 P7
LastX => 188,641.3852 P6

7, ENTER, 3.14, XEQ"HMT$" => 73,726.243390 H7
LastX => 21,659.280490 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 discriminantbeforehand – 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. Note also that 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
12RDN
 13 1          ;next index
14+            ; placed in X
15"(Y/2)^(2*X+Z)"
16FC? 00       ; is it J?
17>"*#1^X"     ;alternate
18EVALT        ;first part
19"T/FT(X)/FT(X+Z)"
20EVALT        ;second part
21R^
22ST+ 00       'add to sum
23FS? 10
24VIEW 00      'show partial
25RND          ;rounding
26X#0?
27GTO 01
28RCL 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$** instructions. Apart from that everything else is very similar to other well-known routines, like SV and IT from the PPC ROM. Just enter the two root guesses (must be different) in Y and X for **SV$**; 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$" => 1.09866 12289

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	**LBL ÏT$"**	*N, a, b, in stack*		27	LBL 09	
2	**STO$ 07**			28	RDN	
3	ENTER^	*b*		29	ST+ Y(2)	
4	**EVAL$**			30	RCL Y(2)	
5	STO 11	*F(b)*		31	**EVAL$**	
6	RDN			32	ST+ X(3)	2x
7	"(X-Y)/Z/2"	*(a-b)/2N*		33	RTN	
8	**EVAL$**			34	LBL 01	
9	**RCL$ 07**			35	RCL 11	
10	RCL Y(2)	*a*		36	"X*T/3"	
11	**EVAL$**	*F(a)*		37	**EVAL$**	
12	ST+ 11	*F(a) +F(b)*		38	**RCL$ 07**	
13	LBL 00			39	**END**	
14	CLX			1	**LBL "SV$"**	
15	*E*			2	**STO$ 07**	
16	ST- T(0)	*decrement N*		3	LBL 00	
17	XEQ 09			4	**EVALZ**	
18	ST+ X(3)	4x		5	X<>Y	
19	ST+ 11	*add to sum*		6	**EVALT**	
20	R^			7	"Y-Z*(Y-X)/(Z-T)"	
21	X=0?			8	**EVAL$**	
22	*GTO 01*			9	FS? 10	
23	RDN			10	VIEW X(3)	
24	XEQ 09			11	**RCL$ 07**	
25	ST+ 11	*add to sum*		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 "S(X)+C(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 and then 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, 135.000000 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).

Note that with the function **A-PM** *you can enter the formula directly in a program step*, just as if you were using the AMC_OS/X module, 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  "S(X)+C(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.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 iterationsis small, leading to a relatively fast convergence.

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

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

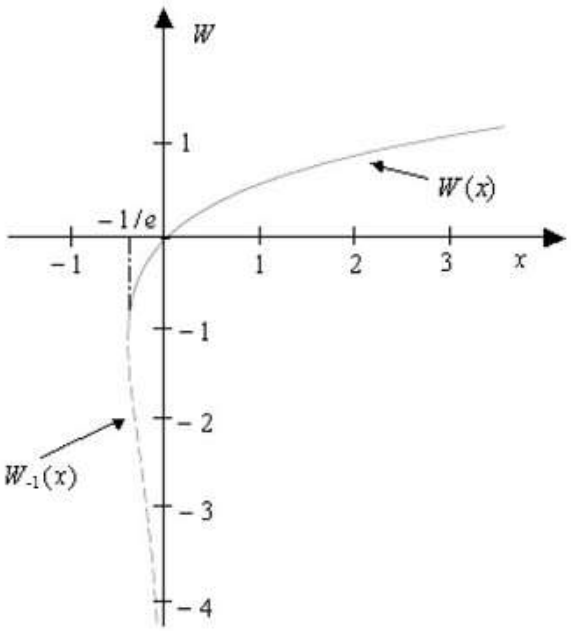
Which with a bit of ingenuity can be written in exactly 24 characters, and therefore only one call to **EVAL$** is required per each iteration. Assuming the current value w is in X, and the argument z is in Z the expression for the next value (w") is as follows:

```
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.

```
01 LBL "WL$
02 FIX 9
03 STO Z
04 LN1+X
05 "X-(X*E(X)-Z)/(1"
06 "|-+X)/E(X)"
07 LBL 00
08 STO Y
09 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.

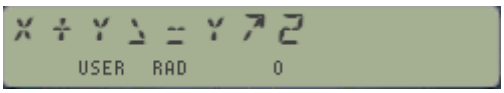
The module includes a set of functions and FOCAL routines designed to make general-purpose test comparisons, and to evaluate finite and infinite Products, Sums and Series. The FOCAL routines are designed pretty much as if they were MCODE functions, in that they preserve the contents of the stack registers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

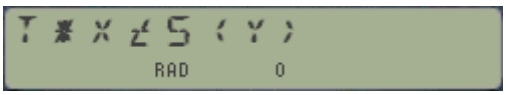
1. Advanced Test Comparisons with **EVAL?**

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≠Y? and X≠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, andnamed **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 used to skip the following program line if FALSE, and do nothing 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 "YES" or "NO" 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? the** FOCAL routine used data registers {R00-R10}, and user flags F0-F4.
- **EVAL?** the MCODE function uses data registers { R00 – R10 }, but no user flags are used.

1	LBL "EVAL?"		32	X=Y?	
2	STO$ (00)	save expression	33	CLX	
3	ST>RG 04	Stack to Regs	34	GTO 00	
4	TST$	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 RIGHT$*	38	GTO 00	*False Line*
8	-		39	LBL 05	
9	E		40	FS? 01	*double case?*
10	-		41	X#Y?	yes, either one
11	FC? 01	was "=" there?	42	X<Y?	*no, strictly "<"*
12	GTO 01	no, skip adjustment	43	CLX	*True Line*
13	FC? 02	*was ">" there?*	44	GTO 00	*False Line*
14	FS? 00	*no, was "<" there?*	45	LBL 02	*double case?*
15	DSE X(3)	*yes, adjust marker*	46	FS? 01	
16	LBL 01		47	X#Y?	yes, either one
17	LEFT$	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 ^2 -- X ) / Z   2 --   T

( Y -- X ) / ( Z -- T )   1   E ( LG ( 2 * 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: Since revision 3H, **EVAL?** became 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.

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

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

```
01  LBL "NCK$"
02  XEQ "NPK$"
03  "X/F(Z)"
04  EVAL$
05  END
```

$\frac{n!}{k!(n-k)!}$

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.

Program listing.

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	STO$ 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 of the module, these 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

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

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

Gamma, with the well-known formula:

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$" ->0.109 12655

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$**" -> 0.9772 13308
1, XEQ "**PSI$**" ->- 0.5772 15666 (opposite of Euler's constant)

-7.28, XEQ "**PSI$**" ->4.65 1 1942 14

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 "GAM$"	*x>0*
2	CF 04	
3	X<0?	
4	X=0?	
5	GTO 04	
6	RAD	
7	SF 04	
8	"1-X"	*X=(1-X)*
9	EVAL$	
10	LBL 04	
11	"P(0;6;Y+X)"	*denominator*
12	EVALP	*saves x in R09*
13	STO 09	*save for later*
14	5,5	
15	LET= a	
16	LASTX	*x*
17	"(X+a)^(X+1/2)/E"	*trascendent term*
18	|-"(X+a)"	
19	EVAL$	
20	STO 10	*partial result*
21	75122,63315	*q0*
22	LET= a	
23	80916,62789	*q1*
24	LET= b	
25	36308,29514	*q2*
26	LET= c	
27	8687,245297	*q3*
28	LET= d	
29	1168,926495	*q4*
30	LET= e	
31	83,86760434	*q5*
32	ENTER^	
33	2,5066282	*q6*
34	LASTX	*x*
35	*"c+X*(d+X*(e+X*("*	*polynomial term*
36	*>"Z+X*Y)))"*	*part-1*
37	EVAL$	
38	*"a+b*L+X*L^2"*	*part-2*
39	EVAL$	
40	RCL 10	get partial result
41	*	factor it in
42	RCL 09	get denominator
43	/	divi by it
44	FC? 04	negative argument?
45	GTO 04	no, skip
46	RCL 08	get (1-x)
47	"π/Y/S(π*(1-X))"	*reflection formula*
48	EVAL$	
49	LBL 04	
50	CLD	clear display
51	END	done.

1	LBL "PSI$"	
2	STO 00	x
3	*0*	*initial delta*
4	X<>Y	
5	*9*	*accuracy limit*
6	X<>Y	
7	LBL 00	
8	X<Y?	
9	X=Y?	*x>=9?*
10	*GTO 01*	*yes, exit loop*
11	*E*	
12	ST+ T(0)	*increase delta*
13	ST+ Y(2)	*increase argument*
14	RDN	fix stack
15	*GTO 00*	do next
16	LBL 01	
17	*"LN(1/X)+1/2/X"*	
18	EVALY	
19	*"R(1/X)"*	
20	EVAL$	
21	*"((X/2--1/21)*X+*	
22	*|-"1/10)*X-1"*	
23	EVAL$	
24	*"X*L/12-Y"*	
25	EVAL$	
26	RCL Z(1)	*delta*
27	X=0?	*was x>=9?*
28	*GTO 01*	*yes, skip adjustment*
29	*E*	
30	-	
31	*"P(0;"*	
32	AINT	
33	RDN	
34	RCL 00	*x*
35	*|-";1/(Y+X))*	
36	EVALP	
37	*	*ok, I cheated here...*
38	X<>Y	
39	LBL 01	
40	X<>Y	
41	CLD	
42	END	

VF++ Conditional Structures

The remaining of this chapter describes the VB-like functions included in the Formula Evaluation module. This function set brings the classic FOCAL programming up one notch, so things begin to resemble a higher-level programming language – even if the same intrinsic platform limitations are still there of course.

These functions are associated in groups or structures, with each of them being either statements or defining independent clauses within the structure. As such, they often include integrity checks built-in to verify the presence of other functions needed to complete the structure. These checks scan the body of the FOCAL program looking for the expected pairing functions, saving their location in memory for later use depending on the result of conditional expressions, etc. Indeed, a new kind of functions and a first on the HP-41 platform.

Function table:

Structure	Function	Description	Technique
DO/WHILE	**DO**	Starts structure	**EVAL?** tests
	WHILE	Repeat while True	Uses RTN Stack
IF.ELSE.ENDIF	**IF**	Does if expression is True	**EVAL?** tests
	ELSE	Does if expression is False	Uses RTN Stack
	ENDIF	End of structure	
FOR..NEXT	**FOR _ _**	Starts loop	bbb.eee:ss parameters in SELCT'ed register
	NEXT _ _	Repeats if not matched	
	LOOPT	Recalls current pointer to X	
SELECT-CASE	**SELECT _ _**	Activates it and Selects Register	SELCT'ed register in Header
	CASE _ _ _	Does clause if True	Case value in Header
	CASELSE	Does Clause regardless	Deactivates search flag
	ENDSLCT	Deactivates and Ends structure	Deactivates flag and clears

As mentioned before, the defining functions { DO, IF, FOR, and SELECT }scan the body of the program to check the integrity of the structure, looking for the pairing function that defines the end of the structure - WHILE, ENDIF, NEXT, and ENDSLCT respectively. When that's not found the error message "NO BOUND" is displayed and the program execution halts:

The VF++ structures are designed to be used in a running program. The individual functions can also be executed manually but the structure integrity check will likely throw the "NO BOUND" error message – unless the program pointer is coincidentally positioned in a memory segment (defined by a global label and END instructions) that includes the pairing function closing the structure – not very likely indeed.

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

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

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"
02 CLST      ;counts starts at zero
03 DO        ;1st marker point
04 CLX       ;reset inner count
05DO         ;2nd marker point
06 1         ;actual code:
07 +         ;increase counter
08 VIEW X    ;for info
09 "X#5"     ;testing condition
10 WHILE     ;repeat while true
11 1         ;resumes 1st DO
12 ST+ Z
13 RDN
14 VIEW Y    ; for information
15 "Y#3"     ;second condition
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 a focal *encapsulation of the **EVAL?**FUNCTION*, plus the branching decisions based on the test result. This is transparent to the user, with he only caveat that the FOCAL program including WHILE cannot be single stepped.

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
	R09,R10	- for Scratch
User Flags:	F00-F03	- to signal the Boolean operator involved

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 FOCAL call to **EVAL?** , oneadditional 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.

WhatIF ?- 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 then will evaluate the test condition and continue normally if the status is TRUE orit 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 pair 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)

```
01  LBL"QRT#"
02  "00XYZ"
03  SHFL
04  "b^2-4*a*c>=0"
05  IF
06  "#(Q(b^2-4*a*c)+b"
07  "|-))/2/a"
08  EVALY
09  "(Q(b^2-4*a*c*)-b)/2/a"
10  ELSE
11  "Q(ABS(b^2-4*a*c)"
12  "|- )/2/a"
13  EVALY
14  "#b/2/a"
15  ENDIF
16  EVAL$
17  END
```

Try it with X^2 =1 to obtain: x1 =1, x2 = -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 groupends at the bottom of thelower 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 mostcommon 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 con't~~
ENDIF(1)

IF(1)
True branch-1
IF(2)
True branch-2
ENDIF(2)
~~True branch-1 con'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 definitionwith 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 straight-forward application of the definition. With n in X, we use it as a counter multiplying all the values by the partial result.

```
1.  LBL "FBO#"
2.  INT
3.  ABS
4.  X=0?          ; is x=0?
5.  RTN           ; yes, abort
6.  "X<3"
7.  IF            ; is X<3?
8.  1             ; yes, X=1
9.  RTN           ; done.
10. ENDIF         ; no, go on
11. 1
12. 1
13. "Z>2"
14. DO            ; loop starts
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

No calculator should lack a good MCODE implementation of the AGM, a fundamental relationship very useful in elliptic functions theory. The SandMath has its own, but 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#"
02 "ABS(IP(Y))
03 EVALY
04 "ABS(IP(X))"     ; INT, ABS
05 EVAL$
06 DO
07 "(X+Y)/2"
08 EVAL$            ; an
09 "Q{L*Y)"         ; L, not X !
10 EVALY            ; gn
11 VIEW X           ; show value
12 X<>Y
13 RND
14 X<>Y
15 RND
16 "X#Y"
17 WHILE            ; in FIX 10
18 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!

Note that the convergence may at times run into issues if all 10 decimal places are used due to oscillations. This can be avoided using rounding values to the desired accuracy, see steps 13-15.

Example: AGM(8, 23) = 14.5 16 19896

As a corollary it is very simple to obtain the **Geometric-Harmonic mean** (GHM), derived from the AGM as per the below relationship and code:

```
01 LBL "GHM#"
02 X<>y
03 1/X
04 X<>Y
05 1/X
06 XROM "AGM#"
07 1/X
08 END
```

$$M(x,y) = \frac{1}{AG(\frac{1}{x}, \frac{1}{y})}$$

Definition of the sequence:

$$\begin{cases} g_{n+1} = \sqrt{g_n h_n} \\ h_{n+1} = \dfrac{2}{\frac{1}{g_n} + \frac{1}{h_n}} \end{cases}$$

Example 3: Ulam's Conjecture

Also known as the Collatz'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.

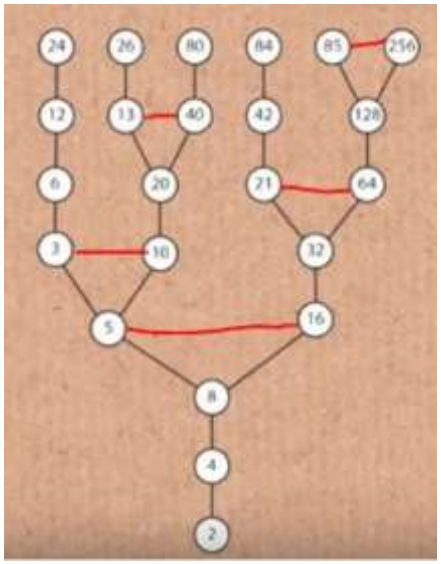
$$f(n) = \begin{cases} \frac{n}{2} & \text{if } n \equiv 0 \pmod 2 \\ 3n+1 & \text{if } n \equiv 1 \pmod 2. \end{cases}$$

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

Multiple visual representations of the Collatz conjecture can be found, some of them more elaborate than others – and many looking very intriguing and of a beautiful nature. Watch Numberphile to see an in-depth discussion of how these visuals were created.
(https://www.youtube.com/watch?v=LqKpkdRRLZw)

Here, each step where you divide by 2 is represented by a vertical black line and each step where you multiply by 3 and 1 is represented by a red horizontal line.



PS. I couldn't resist the temptation to prepare a MCODE version of this program, see the code in next page. FWIW, the speed isn't really the point here due to the visualization of the intermediate results, but nevertheless this showcases the trade-offs between the lowest-level language and the more convenient yet slower versions.

Header	A8EF	*0CD*	*"M"*	
Header	A8F0	*041*	*"A"*	
Header	A8F1	*04C*	*"L"*	
Header	A8F2	*055*	*"U"*	
ULAM	**A8F3**	**0F8**	**READ 3(X)**	
	A8F4	128	WRIT 4(L)	
	A8F5	*149*	*?NC XQ*	*Integer & Positive*
	A8F6	*134*	*->4D52*	*[CHKZI]*
	A8F7	268	WRIT 9(Q)	
	A8F8	04E	C=0 ALL	
	A8F9	0E8	WRIT 3(X)	*reset the counter*
LOOP1	A8FA	00E	A=0 ALL	
	A8FB	35C	PT= 12	*Builds "1" in A*
	A8FC	162	A=A+1 @PT	
	A8FD	278	READ 9(Q)	
	A8FE	36E	?A#C ALL	*end of the path?*
	A8FF	**3A0**	**?NC RTN**	*yes, end here.*
	A900	0F8	READ 3(X)	
	A901	2A0	SETDEC	
	A902	*01D*	*?NC XQ*	*increase counter*
	A903	*060*	*->1807*	*[AD2_10]*
	A904	0E8	WRIT 3(X)	*update value*
	A905	278	READ 9(Q)	*get current n*
	A906	*3CD*	*?NC XQ*	*C= MOD[int(C),2]*
	A907	*100*	*->40F3*	*[MOD2]*
	A908	2EE	?C#0 ALL	*it is odd?*
	A909	***02F***	***JC +05***	*yes, skip*
	A90A	278	READ 9(Q)	
EVEN	A90B	*3CD*	*?NC XQ*	*{A,B} = {C} /2*
	A90C	*13C*	*->4FF3*	*[DIVTWO]*
	A90D	***053***	***JNC +10d***	*show result*
ODD	A90E	04E	C=0 ALL	
	A90F	35C	PT= 12	
	A910	0D0	LD@PT- 3	
	A911	10E	A=C ALL	
	A912	278	READ 9(Q)	
	A913	*135*	*?NC XQ*	*3*n*
	A914	*060*	*->184D*	*[MP2_10]*
	A915	*001*	*?NC XQ*	*3*n+1*
	A916	*060*	*->1800*	*[ADDONE]*
MERGE	A917	268	WRIT 9(Q)	
	A918	*099*	*?NC XQ*	*Sends C to display - sets HEX*
	A919	*02C*	*->0B26*	*[DSPCRG]*
	A91A	*1FD*	*?NC XQ*	*wait a little - CL compatible*
	A91B	*12C*	*->4B7F*	*[WAIT4L] - Enables RAM*
	A91C	*1FD*	*?NC XQ*	*wait a little - CL compatible*
	A91D	*12C*	*->4B7F*	*[WAIT4L] - Enables RAM*
	A91E	*2E3*	*JNC -36d*	*[LOOP1]*

The next logical case for FOR/NEXT (STEP)

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 _ _** prompts for the register number to be used as index variable, which becomes the Selected Variable (selvar#). It also 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) thenthe STEP increments the index, whereas if negative (with bbb>eee) then itdecrements the index. By default, if ss=0 the step is one (standard hp-41 convention for loop functions).

- **NEXT _ _** does the index increment or decrement and loops back to the FOR location of a new iteration kkkuntil all of them are done (when kkk>eee); in which case the execution continues with the program steps below it. Note that the function prompts for the register index variable as well, and thus allowing nested levels.

- The index variable isdeclaredin the prompt, but due to the prompting technique (using OS routines) the value is not automatically entered in program mode, thus *you need to add it manually*. The indexes follow the same convention used all throughout the system. Adding 112 for stack registers and 128 for INDirect addressing. See the table in page #18.

Note that **NEXT** increments (or decrements if the control word is negative) the value in the SELCT'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 SELCT'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. -2          ;= 1.003
38. FOR01       ; S1=01
39. TONE 0
40. -1          ;=1.002
41. FOR 02      ; S=02
42. TONE 1
43. NEXT02      ; Next S2
44. NEXT 01     ; Next S1
45. BEEP
46. 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 SELCT'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 that 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, even if the second one requires functions from the WARP_Core module.

Version #1. Data moved to the stack for the comparison. We'll use R00 and R01 as loop index variables. It is slightly longer and obviously {R00-R01} are reserved (can't contain data to sort).

```
01 LBL "BSORT1"; bbb.eee in X
02 E-3
03 -
04 FOR 00          ; bbb.(eee-1)
05 RCL 00          ; kkk.(eee-1)
06 1.001
07 +
08 FOR 01          ; (kkk+1).eee
09 RCL IND 01      ; R(kkk+1) value
10 RCL IND 00      ; R(kkk) value
11 X<Y?            ; already sorted?
12 GTO 00          ; yes, skip
13 RCL 00          ; kkk.(eee-1)
14 RCL 01          ; (kkk+1).eee
15 X<I>Y           ; (*)
16 LBL 00
17 NEXT 01         ; do next reg
18 NEXT 00         ; do next Reg
19 END
```

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

Version#2. Data in-place.We'll use stack registers Y and X as loop index variables. Note that *it's ok to select other registers inside the loops because NEXT changes the selection back to the index variable*, so perfectly compatible.

```
01 LBL "BSORT2"; bbb.eee in X
02 E-3             ; 0.001
03 -               ; bbb.(eee-1)
04 ENTER^          ; bbb.(eee-1)
05 FOR Y(114)      ; kkk.(eee-1)
06 CLX
07 RCL Y
08 1.001
09 +               ; (kkk+1).eee
10 FOR X(115)      ; (kkk+1).eee
11 SELCT IND X
12 ?S>= IND Y      ; (242)
13 GTO 00
14 S<> IND Y       ; (241)
15 LBL 00
16 NEXTX ; do next X
17 NEXTY ; do nextY
18 END
```

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).

A complete SELECT-CASE Structure

We've just seen how to use the **SELCT** function in the WARP_Core module as an ancillary element in the FOR…NEXT loops. This is very efficient in that it allows selecting data registers, Stack registers and even buffer registers for the selected variable – as well as their INDirect variants.

SELCT is paired with **?CASE**, a yes/no function that skips next line if the content of the selected register doesn't match the value entered as its argument. See the **EVALXM** program later in the manual for a superb example of utilization of this pair of functions.

That's very good al well but it ain't quite the same as a full-fledge SELECT structure with multiple CASE clauses within it, which is now included in the Formula_Eval module and described below.

The new structure has four elements:

- **SELECT_ _** prompts for the selected register number and activates the structure flag. *It must always exist to delimit the beginning of the structure.*

- **CASE_ _ _** clauses prompt for a target value of the selected register. If there's a match the following program lines are run, right up to the next SELECT statement or the end of the structure. It also deactivates the structure flag so that other CASE clauses below it won't get executed. CASE checks for the existence of ENDSLCT downstream in the program code, showing the "NO BOUND" error message if it's not found.

- **CASELSE** is a special kind of CASE that doesn't impose any value-matching condition. It is an optional clause but if it exists *it should be placed after the last CASE statement in the structure*. Like CASE, this function also checks for an active selected variable and the presence of the ENDSLCT clause below it.

- **ENDSLCT** delimits the end of the structure, thus *it too must always exist*. It deactivates the flag and clears the selected register variable.

The trivial example below should illustrate usage. The routine plays a different tone depending on the input value in X, from 0 to 3 – or a BEEP if the value isn't one of those four. For convenience it takes the value in X into the register pointed at by R00 and selects that very register for the structure usage. Just make sure that R00 points at an existing register in memory, of course.

```
01  LBL "SCE"
02  STO IND 00
03  SELECT IND 00 (128)
04  CASE 0
05  TONE 0
06  CASE 1
07  TONE 1
08  CASE 2
09  TONE 2
10  CASE 3
11  TONE 3
12  CASELSE
13  BEEP
14  ENDSLCT
15  END
```

Like any good thing in life, some restrictions: apply - let's review them next:

1. As a consequence of the prompting mechanism used (based on OS routines), there's no automated support for stack registers {X,Y,Z,T,L} or buffer registers (A,B,C,D,E,F,G} in the variable selection. Therefore, when editing a program the selected register number needs to be input manually (i.e. it's not stored by the prompt). Here we can also include Stack registers and Indirect registers, simply by adding to the index the decimal value 112 (0x70 hex) for stack addressing, or 128 (0x80 Hex) for indirect addressing, i..e: 115 = Stack X; 128 = IND 00; 129 = IND 01; 243 = IND ST X. and so on. See the table in next page..

2. Even if these functions can be called in manual mode they're meant to be used in a program. In manual (interactive) mode there's no checks for a matching function, so calling SELECT outside of program mode will not trigger the "NO BOUND" message.

3. Nested SELECT-CASE structures are *onlyfully supported in CASELSE clauses* (which is always last in the structure). This is because the selected register number is stored in the header of buffer#7 (there's only one of them), and as a consequence of the cursory search of the CASE instructions when the conditional is not matched. This restriction is however immaterial if the execution engages the last,non-conditional clause so it'll always get done. Any other situation can run into conflict or false matches resulting in computational errors.

4. It's perfectly ok to include any of the other program-flow controlling groups as part of any SELECT clause – as long as their own nesting rules are respected.For example, the snippets below show three different flow control groups, one on each CASE clause, The value in R00 will determine which of the three branches to execute.

```
SELECT 00
CASE 0          ; FORT..NEXT
1,004           ; play 4 times
FOR
TONE 0
NEXT

CASE 1          ; IF/ELSE
"4*x<3*y^2"
IF
BEEP
ELSE
TONE 9
ENDIF

CASELSE         ;Not 0, not 1
SELECT 1        ; nested level
CASE 01
 TONE 7
CASE 1
 TONE 2
 ENDSLCT
ENDSLCT
```

Note that like the FOR..NEXT group, the SELECT/CASE structure doesn't need to do any formula evaluation (i.e. embedded call to **EVAL?**), thus they're much faster than the DO/WHILE and IF.ELSE.ENDIF counterparts.

Warning: do not mistake the **SELCT** and **SELECT** functions. **SELECT** is part of the new structure, while **SELCT** is a function in the WARP_Code module. **SELCT** is more general-purpose because it features automated entry of the non-merged program step, supporting data registers, stack and buffer addressing. On the other hand, **SELECT** requires manual editing of the non-merged program step – and besides that, it doesn't support buffer registers.

Compatibility with FOR..NEXT structures.

As described already, the SELECT/CASE structure uses buffer#7 header to store the selected register number (selvar#). This is used by the CASE instructions to check matching values with their targets;thus,it follows that the selected register number should be the same at every CASE instance.

Say now that a FOR..NEXT loop is included within a CASE clause, with a modification of the selected register number. This is going to alter the selvar# in the buffer header but *such an alteration is inconsequential to the SELECT/CASE structure because the go-to address of the ENDSLCT instruction is maintained* (i.e. not changed by FOR..NEXT), and thus the following SELECT instruction knows where to transfer the program execution to.

For example:

```
SELECT 01          ; structure begins

CASE XX            ; clause begins
bbb.eee:ss         ; control word
   FOR 02          ; changes selvar# (!)
<code here>
   NEXT 02

CASE YY            ; next clause begins
<code here>

ENDSLCT            ; end of structure.
```

Note that the other way around (a SELECT/CASE structure inside a FOR..NEXT loop) doesn't require any special consideration either because the NEXT instruction also resets the selvar# in the buffer header, thus no conflict can occur.

The table below shows the indexes needed for the non-merged instructions described above.

Argument	Shown as:
100	00
101	01
102	A
103	B
104	C
105	D
106	E
107	F
108	G
109	H
110	I
111	J

Argument	Shown as:
112	T
113	Z
114	Y
115	X
116	L
117	M
118	N
119	O
120	P
121	Q
122	|-
123	a

Argument	Shown as:
124	b
125	c
126	d
127	e
128	IND 00
129	IND 01
130	IND 02
131	IND 03
132	IND 04
133	IND 05
134	IND 06
135	IND 07

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.

- **ST>B7** and **B7>ST** copy the stack registers {X, Y, Z, T, L} to the stack buffer registers (a, b, c, d, e, F} and back respectively. Very useful for variable assignment *en masse*.

- **LOOPT** is an auxiliary function to recall the loop pointer in a FOR..NEXT structure, a.k.a. the content of the selected register. It does the same as **SRCL** in the WARP_Core module.

- 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**, **DRTN2**, **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 is1**, 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|xxxxxxxxxx|CCC", where CCC is the three-digit code in the internal table.

2. If the **sign digit is2**, 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|xxxxxxxxxx|xSS", where SS is the saved ST register contents.

3. If the **sign digit is0or 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

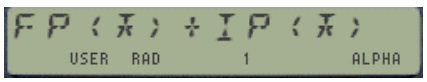
In summary, buffer #6 is used during the execution of the EVAL$ functions in a dynamic manner, populating first its registers with both the arguments and the operations (coding the ALPHA characters as described above), and decoding the registers later to calculate the value of the expression written 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	b10	....	....
	b9	<other>	<available?
	b8	<other>	<available>
	b7	Hex code	Operator-3
	b6	BCD value	4th argument
	b5	Hex code	Operator-2
	b4	BCD value	3rd argument
	b3	Hex code	Operator-1
	b2	BCD value	2nd argument
	b1	BCD value	1st argument
	b0	admin	Header

(*) In the picture abovethe buffer is shown with 7 "active" registers but we know it can hold up to 16 registers with the information on the pending operations and parameters. This size allows for as many intermediate operations as needed to support a 24-char length formula, regardless of how intricate the formula is.

Example: Buffer contents for the formula adding the integer and fractional parts of pi.



b3 @0DC: "π" 0|3141592653|000
b2@0DB: F7 2|0000000000|080 } => IP(π) result to b1
b1 @0DA:"IP" 1|0000000000|150

b4 @0DD: "π" 0|3141592653|000
b3 @0DC: F7,F0 2|0000000000|081 } => FP(π) result to b2
b2 #0DB: "FP" 1|0000000000|250
b1 @0DA: IP(π) 0|3000000000|000

Appendix 3.- Internal EVAL$ Execution flow.

The top-level execution flow to interpret and evaluate a formula string is described below.

1. The first phase is to read the string in ALPHA *from left to right*, checking for possible syntax errors, and pushing all valid elementsinto buffer #6. These can be either actual BCD values or the function codes. This phase is flagged with CPU flag F7 clear. The formula reading routine loops thru all characters and the last character is flagged by CPU flag F5 set. There are disambiguation routines for those multi-character function codes as well (like AS, AC, AT, ABS, AHS, AHC, AHT, etc.)

2. The second phase is the interpretation of the contents of buffer #6, in a LIFO fashion – thus corresponding to an evaluation of the formula in ALHA, this time*from right to left*. This phase is flagged with CPU flag F7 set.

In the interpretation/evaluation phase we distinguish three major categories:

- Arithmetic operations, such as addition, subtraction, multiplication, division, Module, and percentage. Note that these are dual-number operations.
- Single number Functions, all defined in [FNCTBL] and
- Stack & Buffer variables– all defined in [CASTBL]- and Numeric constants, build from ALPHA by [NUMRIC]

The dual-number cases are controlled by CPU flags F0 – F6, according to the tables below.

Routine	Function	F0	F1	F2	F3	F4	F6
[ADDSUB]	Addition	1	0	-	-	-	-
	Subtraction	0	1	-	-	-	-
	Percent	1	1	-	-	-	-
[MLTDIV]	Multiplication	-	-	1	0	-	-
	Division	-	-	0	1	-	-
	Modulus	-	-	1	1	-	-
[DOYTOX]	Power	-	-	-	-	1	-
[INVERS]	Sign Change	-	-	-	-	-	1

What we tried to do with the EVALx series of functions was to insure that only the intended stack register (and LastX) was modified with the results of the formula in Alpha. Hence, EVAL$ touches X and L only, EVALY touches Y and L only, EVALZ touches Z and L only, EVALT touches T and L only, and EVALL touches only L (with EVALL having no LastX capability, of course).

We get away with that because all the partial calculations are pushed/popped from a special buffer (#6) as well as other important formula info. So this yields a lot of power to the user, allowing them to use the stack the way they want to. This action only occurs at the very end of the formula evaluation phase, and that means you have 5 "variables" to use in the stack (X, Y, Z, T, and L) and 6 "parameters" to use in buffer 7 (a, b, c, d, e, and F) before the stack is modified by the final phase of the desired EVALx function.

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

EVLΣ+	Enhanced Sum Eval	String in ALPHA	*Martin-McClure*
EVLP+	Enhanced Product Eval	String in ALPHA	*Martin-McClure*
"GMXM"	Makes GAMMA Script	none	*Martin-McClure*
^01	Puts Chars in R00-R01	Strings in ALPHA	*Martin-McClure*
+REC	Advances one Record	FileName in ALPHA	*Martin-McClure*
"FCT#"	Factorial w/ DO.WHILE	Argument in X	*Ángel Martin*
"FIB#"	Fibonacci Number	Argument in X	*Ángel Martin*
"ULAM#"	Collatz'Conjecture	Argument in X	*Ángel Martin*

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$" =>4.383877 1 E20
77, ENTER^, 27, XEQ "NPK$" =>4.773546 6 E48

5, XEQ "WL$" => 1.326724665

3, XEQ "PSI$" =>0.922784334

75, XEQ "LNG$" =>247.572914 1

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 =>2 1
RDN =>7

1.4, XEQ "CI$" =>0.462006566
1.4, XEQ "SI$" => 1.256226762

24, ENTER^, 6 XEQ "AGM$" => 13.458 17 148

0.5, XEQ "KK$" => 1.854074677

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

1	LBL "NDF"	
2	"Y*Q(2*\p)*E(R((X"	
3	>"-Z)/Y)/2)"	
4	EVAL$	
5	1/X	
6	END	

1	LBL "LINE$"	
2	"(T-Y)/(Z-X)"	
3	EVAL$	
4	"Y-X*L"	
5	EVALY	
6	"Y="	
7	ARCL X(3)	
8	/-"*X"	
9	X<>Y	
10	X<0?	
11	X=0?	
12	>"+"	
13	X<>Y	
14	ARCL Y(2)	
15	AVIEW	*shows line*
16	END	

1	LBL "NCK$"	
2	LBL 00	
3	XROM "NPK$"	
4	"X/F(L)"	
5	EVAL$	
6	RTN	
7	GTO 00	
8	LBL "NPK$"	
9	LBL 02	
10	"P("	
11	RCL Y(2)	
12	RDN	
13	-	
14	ISG X(3)	leaves k in L
15	NOP	
16	AINT	
17	|-";"	
18	R^	
19	AINT	
20	|-";X)"	
21	XROM "EVALP"	
22	RTN	
23	GTO 02	
24	END	

1	LBL "HRMX"	
2	LBL 01	
3	"Σ(1;"	
4	AINT	
5	>";1/X)"	
6	XROM "EVALΣ"	
7	RTN	
8	GTO 01	
9	LBL "ERDOS"	
10	LBL 02	
11	"Σ(1;I;1/(2^X-1)"	
12	>")"	
13	XROM "EVALΣ"	
14	RTN	
15	GTO 02	
16	LBL "e^X"	
17	LBL 03	
18	"Σ(0;I;Y^X/F(X))"	
19	XROM "EVALΣ"	
20	RTN	
21	GTO 03	
22	END	

1	LBL "JNX"	
2	LET=	
3	1	
4	RDN	
5	LET=	
6	2	
7	"C(b*X-a*S(X)) "	
8	*E1*	
9	STO 13	
10	0	
11	PI	
12	XROM "IT$"	
13	PI	
14	/	
15	END	

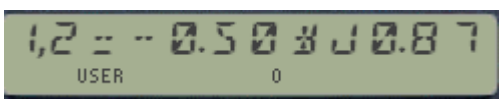
1	LBL "HRON$"	
2	"(X+Y+Z)/2"	
3	EVALT	
4	"Q(T*(T-X)*(T-Y)"	
5	>"*(T-Z))"	
6	EVALT	
7	R^	
8	END	

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

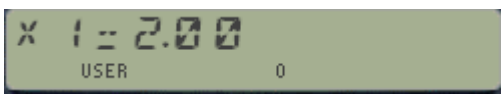
1	LBL "QRT$			1	LBL "P4$"	
2	LBL 00	new start		2	4	
3	LET= c			3	LBL 00	
4	3			4	"a"	
5	RDN			5	ARCLI	
6	LET= b			6	"|-=?"	
7	2			7	PROMPT	
8	RDN			8	STO IND Y	
9	LET= a			9	RDN	
10	1			10	DSE X	
11	"b^2-4*a*c<0"	discriminant		11	GTO 00	
12	XROM "EVAL?"	test for complex roots		12	"a(0)=?"	
13	"Q(ABS(b^2-4*a*c"			13	PROMPT	
14	|-"))/2/a"	avoids DATA ERROR		14	LET=	
15	EVAL$			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					

Examples.

Roots of x^2 + x + 1 =0



Roots of x^2 – 3x + 2 = 0



;

Routine listings for AGM$, KK$, CL$, FL$,FHB$ and SR / RS.

1	LBL "AGM$"
2	LBL 00
3	"Q(X*Y)"
4	EVALY
5	"(X+L)/2"
6	EVAL$
7	RND
8	X<>Y
9	RND
10	X=Y?
11	RTN
12	*GTO 00*
13	LBL "KK$"
14	LBL 01
15	"Q(1-X)"
16	EVAL$
17	E
18	XROM "AGM"
19	"π/2/X"
20	EVAL$
21	RTN
22	*GTO 01*
23	END

1	LBL "FHB$"
2	"Σ(1;"
3	X<>Y
4	AINT
5	|-";"
6	|-"X^Y)"
7	X<.Y
8	XROM "EVAL$"
9	END

1	LBL "R$S"
2	"Q(X^2+Y^2)"
3	EVALT
4	"AT(T/Z)"
5	EVALY
6	"AT(L/X)"
7	EVALZ
8	"Q(L^2+T^2)"
9	EVAL$
10	RTN
11	LBL "S$R"
12	"X*C(Y)"
13	EVALT
14	"X*S(Y)*S(Z)"
15	EVALY
16	"X*S(L)*C(Z)"
17	EVAL$
18	"T"
19	EVALZ
20	END

1	LBL "FL$"
2	CF 00
3	GTO 00
4	LBL "CL$"
5	SF 00
6	LBL 00
7	"X-(X&"
8	FS? 00
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 = IDN - 1,721,119

if Gregorian:
C = int {(N-0.2(/36,524.25]
N ' = N + C - int(C/4)

if Julian:
N' = N + 2

Y' = int[(N' -0.2) / 365.25] ->N" = N' - int(365.25 * Y']
M' = int[(N" - 0.5)/ 30.6] ->D = int [N" - 30.6 * M' + 0.5]

JDN = int{ int [[D + int(367 x) - int(x)] - 0.75 * int(x)] - 0.75 * int[int(x)/100) } + 1,721,115
where: X = Y' + (M-2.85) / 12

Routine listings for ARPLXY and JDN$ / CAL$.

1	LBL "ARPLXY"	
2	X<>Y	
3	POSA	
4	E	
5	+	
6	X=0?	
7	*GTO 00*	
8	AROT	
9	X<>Y	
10	SF#	
11	6	RADEL
12	XTOA	
13	X<>Y	
14	CHS	
15	AROT	
16	LBL 00	
17	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	

23	LBL "CAL$"	
24	*365.25*	
25	LET=	
26	5	*"e"*
27	CLX	
28	*30.6*	
29	LET=	
30	4	*"d"*
31	CLX	
32	*36524.25*	
33	LET=	
34	2	*"b"*
35	CLX	
36	*1721119*	
37	"Y-X+2"	
38	FS? 00	*Julian Cal?*
39	GTO 00	
40	"IP((Y-X-1/5)/b)"	
41	EVALT	
42	"Y-X+T-IP(T/4)"	
43	LBL 00	
44	EVAL$	
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	

Routine listings for SV$ and IT$.

An alternative form of the well-known routines is presented here!

1	LBL "SV$"	x0, x1 in Y,X	
2	STO$ 07	save integrand fnc,	
3	LBL 00		
4	EVALZ	f(x1)	
5	X<>Y		
6	EVALT	f(x0)	
7	"Y-Z*(Y-X)/(Z-T)"		
8	EVAL$	x2 = x1-f(x1).[x1-x0)/f(x1)-f(x0)]	
9	FS? 10		
10	VIEW X(3)		
11	RCL$ 07	recall integrand function	
12	X#Y?	is x1 = x'?	
13	GTO 00	no, do another loop	
14	END	yes, we're done.	

$$x_n = x_{n-1} - f(x_{n-1})\frac{x_{n-1} - x_{n-2}}{f(x_{n-1}) - f(x_{n-2})}$$

it can't get any better than this!

1	LBL ÏT$"	N, a, b, in stack	13	LBL 00	
2	STO$ 07	save formula in R07-R10	14	CLX	
3	ENTER^	b	15	E	
4	EVAL$	f(b)	16	ST- T(0)	decrement N
5	STO 11		17	XEQ 09	2,f((b+(b-a)/2n)
6	RDN	b	18	ST+ X(3)	4,f((b+(b-a)/2n)
7	"(X-Y)/Z/2"	(a-b)/2N	19	ST+ 11	add to sum
8	EVAL$		20	R^	
9	RCL$ 07		21	X=0?	
10	RCL Y(2)	a	22	GTO 01	
11	EVAL$	f(a)	23	RDN	
12	ST+ 11	f(a) +f(b)	24	XEQ 09	
			25	ST+ 11	add to sum
			26	GTO 00	
			27	LBL 09	
			28	RDN	(b-a)/2n
			29	ST+ Y(2)	
			30	RCL Y(2)	b+(b-a)/2n)
			31	EVAL$	f((b+(b-a)/2n)
			32	ST+ X(3)	2,f((b+(b-a)/2n)
			33	RTN	
			34	LBL 01	
			35	RCL 11	
			36	"X*T/3"	
			37	EVAL$	
			38	RCL$ 07	
			39	END	

$$\int_a^b f(x)\,dx \approx \frac{b-a}{6}\left[f(a) + 4f\left(\frac{a+b}{2}\right) + f(b)\right].$$

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.

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 **?SELCT**/**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 **EVLP+**). "**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 place 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

Notice: 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  Y1.0
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.

Binet's Formula for Fibonacci numbers.

Here's a way to obtain the n-th. Fibonacci number directly using Binet's formula, without iterations based on the previous values. Here phi is the golden ratio

```
01 LBL "BINET"
02 16 1803398 9
03 LET= 1 (α)
04 RDN
05   "(α^X-1/(-α^X))/"
06 "Q(5)"
07 EVAL"
08 END
```

$$F_n = \frac{\varphi^n - (-\varphi)^{-n}}{\sqrt{5}}$$

A few more scripting examples:

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

Variables vs. Integer indexes.

If you are using the new **EVLXM+** 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 Length.

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 **EVALXM**. 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.

EVALXM Program Listing.(Updated)

The first version of **EVALXM** used the SELCT/?CASE functions from WARP_Core functions to sieve through the different possible cases of the control characters in the X-Mem ASCII files. With the subsequent addition of the SELECT/CASE/ENDSLCT structure it was possible to replace them and so simplify and enhance the program, saving about 50 bytes in the process. The new version still requires the WARP_Core for functions **?X=** and **?X#**, but it's smaller and leaner thus well worth the change.

Bending the rules just a little bit.

As it was mentioned in the description of the SELECT/CASE/ENDSLCT structure, nested structures are only supported if the second level is placed within the CASELSE clause. This is the only way to always ensure that no conflicts arise in cases where the last level-1 SELECT clause is false – which will trigger a search for other SELECT instructions below it, finding those belonging to the level-2 structure.But *it's possible to have a second-level structure if its code isn't placed within the first one.* This is accomplished using a GOTO approach to move the code off the body of the Level-1. This is exactly what we've done in this case - see the listing in next page. When/If the execution reaches the ENDSLCT instruction in the level-2 structure it clears the pointers in the buffer header and deactivates the structure, thus we need to provide a little extra aid to tell it where to go next, which is back to the mail loop of course. To pick up the next record form the ASCII file and repeat the process.

Program remarks.

The program does a sequential interpretation of the ASCII file records until all are done. The first two characters of each record control the action, and the third one provides additional information if needed.

The program uses {R00 – R04} which therefore cannot be used in any of the script formulas. R02, R03, and R04 are scratch. R00 and R01 hold the first two characters of each ASCII record, which are used as control characters for the scripted action – see the list below as a refresher:

1. 'V'_ space after "Variable" is required
2. 'S'**=** equal sign after "StackVar" is required, does EVAL#
3. 'L'**:** colon after the "Label" is required.
4. **G**_ space after the G is required, "GOTO" statement
5. 'V'**S** stores value at "Variable" into memory register "RegNumber"]
6. 'V'**R** stores value from register into location "Variable"
7. **??** conditional statement using EVAL?
8. **ΣΣ** for using summation function EVALΣ
9. **PP** for using product function EVALP

The two ancillary routines **"+REC**" and **"^01**" perform: (1) a record pointer increase and record to ALPHA, and (2) storage of the first two characters into data registers R00 and R01.

The other function playing a key role here is **TRIAGE**. Its mission is to store the value in R02 into the variableindicated in R00, identified by the ASCII value of its initial letter { X,Y,Z,T,L ;a,b,c,d,e,F }.

Without further ado, here's the program listing.

1	LBL "EVALXM"	ASCII File Evaluation
2	CF 21	non-stop AVIEW
3	STO 00	' SAVE X
4	CLX	disable stack lift
5	SEEKPTA	beginning of file
6	X<> 00	' RESTORE X
7	LBL H	' MAIN LOOP
8	SF 25	' TURN ON TO HANDLE END OF F
9	GETREC	get current record
10	FC?C 25	' END OF FILE CLEARS FLAG 25
11	GTO E	end of program
12	AVIEW	; SHOW RECORD
13	XROM "^01"	puts first 2-Chars in R00-R01
14	SELCT 01	second char (operator)
15	CASE (00)	blank record case
16	GTO E	
17	CASE 126	"Σ" for EVAL Σ
18	|-"Σ"	
19	STO 02	
20	CLX	disable stack lift
21	-1	
22	AROT	
23	X<> 02	
24	EVALΣ	perform the sum
25	CASE 80	"P"for EVALP
26	|-"P"	
27	STO 02	
28	CLX	
29	-1	
30	AROT	
31	X<> 02	
32	EVALP	perform the product
33	CASE 82	"R" FOR RECALL
34	STO 02	; SAVE X
35	CLX	
36	ANUM	GET REG#
37	STO 03	REG# to R03
38	CLX	
39	RCL IND 03	GET VALUE FROM REG#
40	X<> 02	; RESTORE X, VALUE TO R02
41	TRIAGE	SF# 8
42	CASE 58	":" for LABEL
43	CASE 61	"=" for EVAL
44	EVAL#	SF# 5
45	CASE 63	"?" for COMPARE
46	EVAL?	make comparison
47	SKIP	yes, next record
48	XROM "+REC"	advance record
49	CASE 83	"S" FOR STORE
50	STO 02	X to R02
51	CLX	
52	LASTX	
53	STO 04	L to R04
54	CLX	

55	ANUM	
56	STO 03	RG# to R03
57	CLA	
58	X<> 00	chr#(0)
59	XTOA	
60	CLX	
61	RCL 02	restore X
62	EVAL$	evaluate expression
63	STO IND 03	store in RG#
64	X<> 04	recall L
65	STO L (4)	restore L
66	X<> 02	restore X
67	CASE 32	
68	GTO 32	get it out of the way
69	CASELSE	any other char$
70	SYNERR	show "SYNTAX ERR"
71	ENDSLCT	end of Level-1
72	GTO H (109)	next record
73	LBL 32	
74	STO 02	
75	CLX	
76	ANUM	get REG#
77	X<> 02	save in R02, X back
78	SELECT 0	Level-2 begins
79	CASE 71	"G" for GOTO
80	STO 02	; SAVE X
81	CLX	disable stack lift
82	ATOX	get chr# value
83	X<> 02	save it in R02
84	STO 03	; SAVE X IN R03
85	CLX	
86	SEEKPTA	beginning of file
87	LBL 13	
88	CLD	clear LCD
89	XROM "+REC"	advance record
90	XROM "^01"	puts first Chars in R00-R01
91	CLX	
92	58	":" fot LABEL
93	?X# 01	
94	GTO 13	
95	X<> 02	get chr# value
96	?X= 00	
97	GTO 00	
98	X<> 02	save chr# value in R02
99	GTO 13	
100	LBL 00	
101	X<> 03	; RESTORE X
102	CASELSE	
103	TRIAGE	SF# 8
104	ENDSLCT	
105	GTO H (109)	next record
106	LBL E	
107	CLD	
108	WORKFL	SF# 9
109	END	

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)                ; product term
01 XS9                        ; result to R09
02 X=L                        ; argument back to X
03 a 5.5
04 X=(X+a)^(X+1/2)/E(X+a)     ; exponential term
05 XS10                       ; stored in R10
06 a 75122.63315              ; load coefficients
07 b 80916.62789              ; in buffer regs
08 c 36308.29514
09 d 8687.245297
10 e 1168.926495
11 Z 83.86760434
12 Y 2.5066282
13 X=L                        ; argument back to X
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                       ; recall exponential result
19 X=X*Y
20 YR9                        ; recall product result
21 X=X/Y
```

Example: Create the GAMMA ASCII file in X-Mem using **GMXM**, then calculate $\Gamma(1)$ via EVALXM.

XEQ "GMXM", 1, **EVALXM** => 1.000000000

Below you can see the code for the **EVALXM** routine. Note the repeated use of functions **?SELCT**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.

Script Reading program comparison summary

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

Program	*Condition Routine*	*Summing Routine*	*Multiplying Routine*	*Other Routines*
EVALXM	**EVAL?**	**EVALΣ**	**EVALP**	n/a
EVLXM+	**EVAL?**	XROM "EVLΣ+"	XROM "EVLP+"	XROM "IT$" XROM "SV$"

Note that besides being capable of using integral and solve commands directly in the scripts**, the enhanced 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 the enhanced versions **EVLΣ+** and **EVLP+** also *allow using expression formulas for the indexes*, i.e. the actual index value is calculated during program execution.

Appendix 0. 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_VF	EVAl_VF
Upper Page	EVAL_APPS	**EVAL_EQNS**

Appendix. Enhanced 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       ; no-stop
03  STO 00      ; x to R00
04  CLX
05  SEEKPTA     ; top of file
06  X<> 00      ; x is back
07  LBL H       ;new record
08  SF 25
09  GETREC
10  FC?C 25     ; last one?
11  GTO  E      ; yes, exit
12  AVIEW       ; no, show
13  XROM "^01"
14  SELCT 1
16  ?CASE 126 ;evalΣ
18  GTO 10
19  ?CASE 80    ; evalP
21  GTO 12
22  ?CASE 61    ; = assign
24  GTO 20
25  ?CASE32     ; space
27  GTO 21
28  ?CASE 70    ; Forwd
30  GTO 21
31  ?CASE 66    ; Backd
33  GTO 21
34  ?CASE 79    ; Do
36  GTO H
37  ?CASE 58    ; : label
39  GTO H
40  ?CASE 63    ; eval?
42  GTO 23
43  ?CASE 82    ; Recall
45  GTO 24
46  ?CASE 83    ; Store
48  GTO 25
49  ?CASE 88    ; neXt
51  GTO 27
52  ?CASE 84    ; inTeg
54  GTO 28
55  ?CASE 86    ; solVe
57  GTO 29
58  ?CASE  0
59  GTO E       ; Exit
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 "EVLP+"
78  GTO H
79  LBL 20      ; : assign
80  SF#05  (EVAL#)
82  GTO H
83  LBL 21      ; Frwd. GTO
84  SELCT 0
85  ?CASE 71    ; Goto
87  GTO 10
88  ?CASE 87    ; While
90  GTO 17
91  ?CASE 70    ; For
93  GTO 26
94  STO 02
95  CLX
96  ANUM
97  X<> 02
98  SF# 8 (TRIAGE)
100 GTO H
101 LBL 10      ; Goto
102 CLD
103 ST>RG 03
105 ATOX
106 STO 02
107 SELCT 0     ; again?
108 E
109 ?CASE 70    ; For
111 GTO 11
112 ?CASE  66       ; Bck.
GTO
114 GTO 15
115 CLX
116 SEEKPT
117 GTO 11
118 LBL 15 ; Bck. GTO
119 RCLPT
120 INT
121 X=0?
122 SYNERR
123 DSE X
124 NOP
125 SEEKPT
126 XROM "+REC"
127 XROM "^01"
128 SELCT 1
130 ?CASE 58        ; :
132 GTO 00
133 GTO 15
134 LBL 00  ; : assign
135 RCL 00
136 ?X# 02
139 GTO 15
141 RG>ST 03
143 GTO H
144 LBL 11
145 XROM "+REC"
146 XROM "^01"
147 SELCT 1
149 ?CASE 58        ; :
151 GTO 00
152 GTO 11
153 LBL 00
154 RCL 00
155 ?X#0 2
158 GTO 11
160 RG>ST 03
162 GTO H
163 LBL 17  ;
164 EVAL?
165 GTO 04
166 GTO H
166 LBL 04
```

```
167     CLD
168     ST>RG 02
170     LBL 22
171     RCLPT
172     INT
173     X=0?
174     SYNERR
175     DSE X
176     NOP
177     SEEKPT
178     XROM “+REC”
179     XROM “^01”
180     SELCT   ; again?
181     ?CASE 68        ; D
183     GTO 00
184     GTO 22
185     LBL 00
186     RG>ST02
188     GTO H
189     LBL 23  ;
190     EVAL?
191     GTO H
193     XROM “+REC”
194     GTO H
195     LBL 24  ;
196     ST>RG 04
198     ANUM
199     STO 03
200     RCL IND 03
201     STO 02
202     RG>ST04
204     SF# 08 (TRIAGE)
206     GTO H
207     LBL 25  ;
208     ST>RG 04
210     ANUM
211     STO 03
212     CLA
213     RCL 00
214     XTOA
215     RCL 07
216     EVAL$
217     STO IND 03
218     RG>ST 04
220     GTO H
221     LBL 26
222     ST>RG 03
224     ATOX
225     STO 00
226     ATOX
227     RG>ST03
229     1ST
230     RG>ST03
232     EVAL$
233     STO 02
234     X<> 06
235     SF# 8 (TRIAGE)
237     GTO H
238     LBL 27
239     CLD
240     ST>RG 02
242     RCLPT
243     STO 11
244     CLX
245     LBL 30
246     RCLPT
248     INT
249     X=0?
250     SYNERR
251     DSE X
252     NOP
253     SEEKPT
254     XROM “+REC”
255     XROM “^01”
256     SELCT
257     ?CASE 70        ; For
259     GTO 00
260     GTO 30
261     LBL 00
262     STO$ 07
264     ATOX
265     STO 00
266     CLA
267     XTOA
268     " ├="
269     XTOA
270     " ├+1"
271     RG>ST 02
273     SF#05 (EVAL#)
275     ST>RG 02
277     RCL$ 07
279     CF 00
280     2ND
281     RCL 00
282     XTOA
283     " ├<="
284     -3
285     AROT
286     RG>ST 2
288     EVAL?
289     GTO H
291     X<> 11
292     INT
293     SEEKPT
294     XROM “+REC”
295     X<> 11
296     GTO H
297     LBL 29  ; solve
298     SF 01
299     GTO 01
300     LBL 28  ; integrate
301     CF 01
302     LBL 01
303     ST>RG 02
305     SF# 07  (RADEL)
307     STO$ 07
309     1ST
310     RG>ST 02
312     EVAL$
313     FC? 01
314     STO 03
315     FS? 01
316     STO 04
317     RCL$ 07
319     SF 00
320     2ND
321     RG>ST 02
323     EVAL$
324     FC? 01
325     STO 04
326     FS? 01
327     STO 05
328     RCL$ 07
330     FS? 01
331     CF 00
332     3RD
333     FS? 01
334     GTO 00
335     EVAL$
336     STO 05
337     RCL$ 07
339     4TH
340     LBL 00
341     RG>ST 02
343     STO 06
344     FC? 01
345     XROM "IT$"
346     FS? 01
347     XROM "SV$"
348     STO 05
```

```
349   RG>ST02
351   GTO H
```

```
368   LBL E   ; EXIT
369   SF# 9  (WRKFILE)
```

```
371   CLD
372   END
```

```
1     LBL "1ST"
2     LBL 01
3     59
4     POSA
5     LEFT$
6     RTN
7     LBL "2ND"
8     XEQ 07
9     FS? 00
10    GTO 01
```

```
11    RTN
12    LBL "3RD"
13    XEQ 07
14    XEQ 07
15    FS? 00
16    GTO 01
17    RTN
18    LBL "4TH"
19    XEQ 07
20    XEQ 07
```

```
21    LBL 07
22    ALENG
23    59
24    POSA
25    -
26    E
27    -
28    RIGHT$
29    END
```

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

- "+REC" simply gets the next record content to ALPHA, and shows an error if somehow the end of file is reached (not meant to at this point).

- "^01" gets the char$ value for the two first characters into data registers R00 and R01 respectively. The characters are removed from the ALPA string.

```
1     LBL "^01"     puts first Two Chars in R00-R01
2     STO 00
3     CLX           disable stack lift
4     ATOX          left char to X
5     X<> 00
6     STO 01
7     CLX           disable stack lift
8     ATOX          left char to X
9     X<> 01
10    RTN

11    LBL "+REC"
12    SF 25
13    GETREC
14    FC?C 25       end of File?
15    SYNERR        show "SYNTAX ERR"
16    END
```

EVALXM uses {R00-R10} for scratch, thus they cannot be used in the script.

Like it was the case with **EVALXM**, one of the major hurdles for the scripting programs is the fact that there's no available registers for scratch calculations: indeed, the stack, buffer registers and data registers can all and any of them be part of the expressions used in the formulas, thus they're not freely available for the program's internal usage.

This is overcome by using functions such as **?CASE**, **?X=**, etc. that feature in-place argument capability, i.e. there's no need to bring the arguments to the stack to operate on them. The only drawback is the requirement of the WARP_Core module – where those functions reside.

Modified EVLXM+ using SELECT-CASE Structures

Like what we did for the standard EVALXM, applying the new conditional structures to the task has saved about 100 bytes and made the program easier to follow and maintain.

In this conversion we have taken a few liberties (a.k.a. bent the rules a bit more), as follows.

- Taking several clauses outside of the body of the main structure to avoid interferences with sub-level clauses.
- Using GTO statements between structure clauses, which would make structure programming purists frown but works like a charm.
- Finally, we also use the **SELCT**/ **?CASE** functions from the **WARP_Core** within some clauses: they are compatible if we keep in mind their mutual impact on the selected variable declaration, and that they don't alter the skip-to ENDSLCT address saved in the buffer header by **SELECT,** nor they deactivate the structure flag either.

All these tricks and techniques may appear a bit confusing perhaps, specially at first, but all combined it adds the ultimate flexibility for more efficient and smaller code.Below you can see the result.

```
01  *LBL "EVLXM+"
02  CF 21
03  STO 00
04  CLX
05  SEEKPTA
06  X<> 00
07  *LBL H
08  SF 25
09  GETREC
10  FC?C 25
11  GTO E
12  AVIEW
13  ^01
14  SELECT 1
16  CASE 0
17  GTO E
18  CASE 126
20  >"Σ"
21  STO 02
22  CLX
23  -1
24  AROT
25  X<> 02
26  XROM "EVLΣ+"
27  CASE 80
29  >"P"
30  STO 02
31  CLX
32  -1
33  AROT
34  X<> 02
35  XROM "EVLP+"
36  CASE 61
38  EVAL#  (SF# 5)
40  CASE 32
42  GTO 21 ; -> moved
43  CASE 70
45  GTO 21
46  CASE 66
48  GTO 21
49  CASE 58
51  CASE 79
53  CASE 63
55  EVAL?
56  GTO H
57  XROM "+REC"
58  CASE 82
60  ST>RG 04
62  ANUM
63  STO 03
64  RCL IND 03
65  STO 02
66  RG>ST 04
68  TRIAGE  (SF# 8)
70  CASE 83
72  ST>RG 04
74  ANUM
75  STO 03
76  CLA
77  RCL 00
78  XTOA
79  RCL 07
80  EVAL$
81  STO IND 03
82  RG>ST 04
84  CASE 88
86  GTO 27 ; -> moved
87  CASE 86
89  SF 01
90  GTO 01
91  CASE 84
93  CF 01
94  *LBL 01
95  ST>RG 02
97  RADEL  (SF#7)
99  STO$ 07
101 XROM "1ST"
102 RG>ST 02
104 EVAL$
105 FC? 01
106 STO 03
107 FS? 01
108 STO 04
109 RCL$ 07
111 SF 00
112 XROM "2ND"
113 RG>ST 02
115 EVAL$
116 FC? 01
117 STO 04
118 FS? 01
119 STO 05
120 RCL$ 07
122 FS? 01
123 CF 00
124 XROM "3RD"
125 FS? 01
126 GTO 00
127 EVAL$
128 STO 05
```

```
129   RCL$ 07
131   XROM "4TH"
132   *LBL 00
133   RG>ST 02
135   STO 06
136   FC? 01
137   XROM "IT$"
138   FS? 01
139   XROM "SV$"
140   STO 05
141   RG>ST 02
143   CASELSE
144   SYNERR
145   ENDSLCT
146   GTO H
147   *LBL 21 ; moved out
148   SELECT 0
149   CASE70
151   ST>RG 03
153   ATOX
154   STO 00
155   ATOX
156   RG>ST 03
158   XROM "1ST"
159   RG>ST 03
161   EVAL$
162   STO 02
163   X<> 06
164   TRIAGE (SF# 8)
166   CASE 71
168   GTO 10
169   CASE 87
171   GTO 17 ; -> moved
172   CASELSE
173   STO 02
174   CLX
175   ANUM
176   X<> 02
177   TRIAGE (SF#8)
179   ENDSLCT
180   GTO H
181   *LBL 17
182   EVAL?
183   GTO 04
184   GTO H
185   *LBL 04
186   CLD
187   ST>RG 02
189   *LBL 22
190   XEQ 99
191   SELECT 0
192   CASE 68
194   RG>ST02
196   CASELSE
197   GTO 22
198   ENDSLCT
199   GTO H
200   *LBL 10
201   CLD
202   ST>RG 03
204   ATOX
205   STO 02
206   SELECT 1
208   CASE 70
210   *LBL 11
211   XROM "+REC"
212   XROM "^01"
213   SLCT 1
215   ?CASE 58
217   GTO 00
218   GTO 11
219   *LBL 00
220   RCL 00
221   ?X# 02
223   GTO 11
224   RG>ST 03
226   CASE 66
228   *LBL 15
229   XEQ 99
230   SLCT 1
232   ?CASE 58
234   GTO 00
235   GTO 15
236   *LBL 00
237   RCL 00
238   ?X# 02
240   GTO 15
241   RG>ST 03
243   CASELSE
244   CLX
245   SEEKPT
246   GTO 11
247   ENDSLCT
248   GTO H
249   *LBL 27 ; moved
250   CLD
251   ST>RG 02
253   RCLPT
254   STO 11
255   CLX
256   *LBL 30
257   XEQ 99
258   SELECT 0
259   CASE 70
261   STO$ 07
263   ATOX
264   STO 00
265   CLA
266   XTOA
267   >"="
268   XTOA
269   >"+1"
270   RG>ST 02
272   EVAL# (SF#5)
274   ST>RG 02
276   RCL$ 07
278   CF 00
279   XROM "2ND"
280   RCL 00
281   XTOA
282   >"="
283   -3
284   AROT
285   RG>ST 02
287   EVAL?
288   GTO H
289   X<> 11
290   INT
291   SEEKPT
292   XROM "+REC"
293   X<> 11
294   CASELSE
295   GTO 30
296   ENDSLCT
297   GTO H
298   *LBL 99
299   RCLPT
300   INT
301   X=0?
302   SYNERR
303   DSE X
304   ADV
305   SEEKPT
306   XROM "+REC"
307   XROM "^01"
308   RTN
309   *LBL E
310   WORKFL (SF# 09)
312   CLD
313   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 Martin*
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*	*LeftJ, Show and Halt*
	A8EA	*0FE*	*->3F7C*	*[APEREX]*

At this point let's refresh our understanding of the RTN stack registers. Remember that because of the RAM/ROM issue *PCNT's format is not the same as the other six RTN addresses !*

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 Martin*
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"*	***Pops first RTN address***
Header	ACB2	*03E*	*">"*	
Header	ACB3	*051*	*"Q"*	
Header	ACB4	*058*	*"X"*	*Hakan Thörgren*
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"*	***Kills the 2nd. RTN adr***
Header	A335	*054*	*"T"*	
Header	A336	*052*	*"R"*	
Header	A337	*04B*	*"K"*	*Ángel Martin*
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	0A4	->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
annot 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	PNC 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 addr 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 three loop pointers (step size, beginning and end). The variable is the SELCT'edregister, and the pointers are combined in bbb.eee:ss 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 SELCT'ed registerand (2) pushing its own location address in the RTN stack (for the **NEXT** statement consumption later on), done in the [GTO2ADR] routine. This second task is identical to [DO]'s mission, thus the execution is transferred to the same point for that.

Header	A8B9	092	"R"	Saves X in SLCT and does PC>RTN
Header	A8BA	00F	"O"	bbb.eee:ss
Header	A8BB	006	"F"	Ángel Martin
FOR	A8BC	248	SETF 9	F8 is used by BCDBIN w/ IND SELCT
	A8BD	0F8	READ 3(X)	get control word
	A8BE	070	N=C ALL	bbb.eee:ss
	A8BF	379	PORT DEP:	Selects SLCT register, and puts
le with	A8C0	03C	XQ	header addr in Q, content in M
!!	A8C1	185	->A985	[SELSLC] - leaves SELCT selected
small advance?	A8C2	0B0	C=N ALL	no, recall control word
	A8C3	2F0	WRTDATA	put cnt'l word in SLCT reg
	A8C4	130	LDI S&X	
	A8C5	0A1	CON:	NEXT 2nd. byte
	A8C6	033	JNC +06	merge w/ DO code
Header	A8C7	08F	"O"	does PC>RTN
Header	A8C8	004	"D"	Ángel Martin
DO	A8C9	244	CLRF 9	F8 is used by BCDBIN w/ IND SELCT
	A8CA	130	LDI S&X	
	A8CB	09C	CON:	WHILE 2nd. byte
and	A8CC	39C	PT= 0	
!	A8CD	058	G=C @PT,+	
)	A8CE	379	PORT DEP:	Search for matching WHILE
	A8CF	03C	XQ	doesn't change the PC
	A8D0	08E	->A88E	[?END0]
NOTFND	A8D1	0E3	JNC +28d	Show "NO_BOUND" msg
FOUND	A8D2	141	?NC XQ	get current PC address
	A8D3	0A4	->2950	[GETPC] - points at byte * after* D
at [DO]	A8D4	379	PORT DEP:	Push DO/FOR adr+2 in RTN#1
h iteration	A8D5	03C	XQ	for loop return
r in RTN stack	A8D6	0D9	->A8D9	[GO2ADR]
	A8D7	3C1	?NC GO	
	A8D8	002	->00F0	[NFRPU]

Note that the address saved in the FOCAL RTN stack is for the instruction following FOR, in other words FOR is only executed once. This is a vital design point, that frees up the X-register within the loop.

Also done by FOR is the search for a NEXT instruction downstream to ensure the loop integrity.

The subroutine [SELSLC] shown below is also used by **NEXT** to read the current control word kkk.eee:ss, and check the existence of the variable register. It leaves it selected on exit.

SELSLC	A985	369	?NC XQ	Check buffer id#7 - > header in C
	A986	124	->49DA	[CHKBF#7] - returns addr in A.X
ELECT" token	A987	0A6	A<>C S&X	
CT update	A988	270	RAMSLCT	
	A989	038	READATA	get header content
	A98A	03C	RCR 3	yes, shift group
	A98B	266	C=C-1 S&X	remove the padding
	A98C	01B	JNC +03	if zero, replace with default
	A98D	130	LDI S&X	
	A98E	073	X register	defaults to X if no info
	A98F	106	A=C S&X	put in A.X for vetting
	A990	321	?NC XQ	Check Existence - IND, STK
	A991	138	->4EC8	[EXISTS3] - adr in A.X
	A992	0A6	A<>C S&X	
	A993	270	RAMSLCT	select SELCT'ed register
	A994	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 line below 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 the line below **FOR** is made by copying the first return address from RTN1 into the program pointer PC. Easy does it, no frills, no added complexity. See code segment from 0xA97F to 0xA984 in next page. Interestingly this code stream ends with a call to [NFRPU], needed to tell the OS where to go when the MCODE RTN stack is empty – which occurs when the selected register is an INDirect type or a buffer register.

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	A951	*094*	*"T"*	***Increase Cnt'l word and decide***
Header	A952	*018*	*"X"*	*whether to return to FOR*
Header	A953	*005*	*"E"*	
Header	A954	*00E*	*"N"*	*Ángel Martin*
NEXT	**A955**	*379*	*PORT DEP:*	*Selects SLCT register, and puts*
	A956	*03C*	*XQ*	*header addr in Q, content in M*
	A957	*185*	*->A985*	*[SELSLC] - leaves SELCT selected*
	A958	038	READATA	*bbb.eee:ss*
	A959	*361*	*?NC XQ*	*(includes SETDEC)*
	A95A	*050*	*->14D8*	*[CHK_NO_S]*
	A95B	*11D*	*?NC XQ*	*"0|EEESS00000|000" in A*
	A95C	*064*	*->1947*	*[SINFR]*
	A95D	3FA	LSHFA M	"0|EESS000000|000"
	A95E	3FA	LSHFA M	"0|ESS0000000|000"
	A95F	3FA	LSHFA M	"0|SS00000000|000"
	A960	35C	PT= 12	
	A961	342	?A#0 @PT	
	A962	***037***	***JC +06***	
	A963	3FA	LSHFA M	"0|S000000000|000"
	A964	342	?A#0 @PT	*step#=0?*
	A965	***027***	***JC +04***	*no, skip*
	A966	162	A=A+1 @PT	*yes, add one*
	A967	*013*	*JNC +02*	
STEP>9	A968	166	A=A+1 S&X	
STEP#	A969	038	READATA	kkk,eee:ss ; kkk>=bbb
	A96A	*01D*	*?NC XQ*	
	A96B	*060*	*->1807*	*[AD2_10]*
	A96C	070	N=C ALL	
	A96D	*035*	*?NC XQ*	*write result to register*
	A96E	*124*	*->490D*	*[WRTSEL] - selects Chip0*
	A96F	0B0	C=N ALL	*get final sum*
	A970	*379*	*?NC XQ*	*puts kkk in C,A; eee in N*
	A971	*100*	*->40DE*	*[BRKUP4]+1*
	A972	0F0	C<>N ALL	*kkk in N; eee in C*
	A973	10E	A=C ALL	eee
	A974	0B0	C=N ALL	*kkk*
	A975	2BE	C=-C-1 MS	*-kkk*
	A976	000	NOP	*let carry settle*
	A977	*01D*	*?NC XQ*	*eee-kkk*
	A978	*060*	*->1807*	*[AD2-10]*
	A979	260	SETHEX	
	A97A	2FE	?C#0 MS	*is kkk>eee ?*
	A97B	*023*	*JNC +04*	*no, copy RTN2 into RTN1*
REPEAT	A97C	*3AD*	*PORT DEP:*	*yes, kill the [FOR] RTN addr*
	A97D	*08C*	***GO***	*and merrily go on...*
	A97E	*0B5*	*->ACB5*	*[XQ>GO]*
LOOP#	A97F	01C	PT= 3	*write RTN1 into the PC*
	A980	338	READ 12(b)	*get lower RTN stack*
	A981	07C	RCR 4	*put RTN1 in C<3:0>*
	A982	10A	A=C PT<-	*copy to A<3:0>*
	A983	*049*	*?NC GO*	*go back to line below FOR*
	A984	*096*	*->2512*	*[XGNN10] =- [PUTPC] + [NFRPU]*

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.

SELECT-CASE Structures

Not to be underdone, let's tackle the last program flow control to end this section. The WARP Core includes a precursor of this structure, namely functions **SELCT**and **?CASE**. They do a wonderful job by themselves as it can be seen in the **EVALXM** programs, but frankly they're not the real McCoy.

For the complete SELECT-CASE implementation we need to include four new functions that, working collaboratively, can handle all instances arising in this type of structure. They are the four clauses of the structure:

1. **SELECT _ _** demarcates the beginning and initializes variables, pointers and flags. It also searches for ENDSLECT to check for structural integrity – done by our trusty [?END0] subroutine again. The selected data register is in the program line below as a non-merged line, and it is retrieved by the [GETRG#] subroutine.

2. **CASE _ _ _** runs the code segment if the value of the selected variable matches the input parameter, otherwise it passes the baton to the following CASE instruction, if there's any below in the structure. On paired matches it also clears the "active" flag so the subsequent CASEs will be ignored.

3. **CASELSE** is a special case without numeric value to match – so anything goes. It's the "last chance" clause after all other options have been checked without a match.

4. **ENDSLCT** terminates the structure by clearing variables and flags.

SELECT	selects REG, sets flag and search for ENDSLCT	
CASE	if flag is clear: move PC to ENDSLCT	
	if flag is set, check value:	
		if value is true, clear it and run instructions below
		if value is false, move to next CASE or ENDSLCT
CASELSE	clear flag and run stuff downstream	
ENDSLCT	clear flag and beep	

The header buffer holds the following data:

"77|SZ|ADDR|REG|F00" where:

- ADDR is the address of the ENDSLCT instruction
- REG the number of the selected data register (can be INDirect)
- When C[XS] ="F"it denotes an active structure, otherwise C[XS]=0

Note: Importantly enough, the FOCAL RTN stack (status registers a(11) and b(12)) is not used to store any address – so it's available for all other flow control groups to use without interferences.

Here's the MCODE for SELECT, showing how the previous considerations are done:

Header	AA52	094	"T"	
Header	AA53	003	"C"	Inputs SELCT'ed REG#
Header	AA54	005	"E"	and activates SELCASE flag
Header	AA55	00C	"L"	Warning: no nested ability!
Header	AA56	205	"E"	
Header	AA57	213	"S"	Ángel Martin
SELECT _ _	AA58	04C	?FSET 4	SST'ing a program
	AA59	01F	JC +03	yes, divert
{ addresses	AA60	2CC	?FSET 13	RUN'ing a program?
	AA5B	01B	JNC +03	nom skip
	AA5C	179	?NC XQ	Get Parameter from NextLine
	AA5D	10C	->435E	[GETRG#]
CON'T	AA5E	0A6	A<>C S&X	input parameter to C.X
	AA5F	226	C=C+1 S&X	pad it
	AA60	070	N=C ALL	save it in N.X for later
	AA61	39C	PT= 0	
	AA62	130	LDI S&X	
	AA63	0A5	CON:	ENDSLCT 2nd. byte
	AA64	058	G=C @PT,+	
	AA65	379	PORT DEP:	Search for matching ENDSLCT
	AA66	03C	XQ	doesn't change the PC
	AA67	08E	->A88E	[?END0] - result in B<3:0>
NOTFND	AA68	09B	JNC +19d	Show "NO_BOUND" msg
FOUND	AA69	369	?NC XQ	Check buffer id#7 - > header in C
	AA6A	124	->49DA	[CHKBF#7] - returns addr in A.X
	AA6B	0A6	A<>C S&X	
	AA6C	270	RAMSLCT	select buf header
	AA6D	038	READATA	get header content
	AA6E	056	C=0 XS	
	AA6F	276	C=C-1 XS	flags active SELCT
	AA70	17C	RCR 6	rotate ADDR field to C<3:0>
	AA71	01C	PT= 3	
	AA72	0CA	C=B PT<-	copy ENDSLCT addr to C<3:0>
	AA73	1BC	RCR 11	rotate 3 digits to left
	AA74	10E	A=C ALL	move it to A
	AA75	0B0	C=N ALL	recover REG#
	AA76	106	A=C S&X	put it in A.X
	AA77	0AE	A<>C ALL	complete header to C
	AA78	1BC	RCR 11	"77|SZ|**ADDR**|REG|F00"
	AA79	035	?NC GO	write updated header
	AA7A	126	->490D	[WRTSEL] - selects Chip0
NOBOUND	AA7B	369	PORT DEP:	Show "NO_BOUND" msg
	AA7C	03C	GO	
	AA7D	0ED	->A8ED	[NOBOND]

Next comes CASE, a bit trickier in that it needs to have the logic to make a go/no-go call based on the parameter values. It also needs to deactivate the structure in cases of matched conditions, and search forother CASE or CASELSE statements when said flag is inactive.

Here's the first part, note how CASELSE piggybacks on the CASE code, as most of the work is done by both anyway. Here flag 9 is used to tell the cases apart.

NOBOUND	AA7B	369	PORT DEP:	Show "NO_BOUND" msg
	AA7C	03C	GO	
	AA7D	0ED	->A8ED	[NOBOND]
Header	AA7E	085	"E"	
Header	AA7F	013	"S"	**Any aoter value**
Header	AA80	00C	"L"	Anything goes !
Header	AA81	005	"E"	
Header	AA82	013	"S"	
Header	AA83	001	"A"	
Header	AA84	003	"C"	Ángel Martin
CASELSE	AA85	248	SETF 9	F8 is used by BCDBIN w/ IND SELCT
	AA86	073	JNC +14d	
Header	AA87	085	"E"	**Checks for match between**
Header	AA88	013	"S"	SELCTéd Rge content and
Header	AA89	101	"A"	parameter in prompt
Header	AA8A	103	"C"	Ángel Martin
CASE _ _ _	AA8B	244	CLRF 9	F8 is used by BCDBIN w/ IND SELCT
	AA8C	04C	?FSET 4	SST'ing a program
(addresses	AA8D	01F	JC +03	yes, divert
	AA8E	2CC	?FSET 13	RUN'ing a program?
	AA8F	01B	JNC +03	nom skip
	AA90	179	?NC XQ	Get Parameter from NextLine
	AA91	10C	->435E	[GETRG#]
CON'T	AA92	0A6	A<>C S&X	input parameter to C.X
	AA93	070	N=C ALL	save CASE VAL# in N.X for later
CASELS#	AA94	369	?NC XQ	Check buffer id#7 - > header in C
	AA95	124	->49DA	[CHKBF#7] - returns addr in A.X
	AA96	0A6	A<>C S&X	buffer address
	AA97	268	WRIT 9(Q)	saved in Q.X for later
	AA98	270	RAMSLCT	get header content
	AA99	038	READATA	77|SZ|ADDR|REG|F00
	AA9A	17C	RCR 6	rotate ADDR to C<3:0>
	AA9B	01C	PT= 3	"REG|F00|77|SZ|ADDR"
	AA9C	2EA	?C#0 PT<-	is ENDSLCT addr zero?
	AA9D	2F3	JNC -34d	yes, Show "NO_BOUND" msg
	AA9E	0DC	PT= 10	will check flag
	AA9F	2E2	?C#0 @PT	is SELCT active?
	AAA0	03F	JC +07	yes, skip over
PCTOEND	AAA1	01C	PT= 3	no. already done in another CASE
	AAA2	10A	A=C PT<-	move PC to ENDSLCT
PCTONEXT	AAA3	31D	?NC XQ	backtrack one byte
	AAA4	0A4	->29C7	[DECAD]
	AAA5	0B1	?NC GO	[DECAD] plus [PUTPC]
	AAA6	08E	->232C	[PUTPCD]
ACTIVE	AAA7	24C	?FSET 9	is this CASELESE?
	AAA8	360	?C RTN	yes, let go!
	AAA9	158	M=C ALL	safeguard REG adr in M<3:0>

CASE continues with the comparison between the register content and the CASE value, now saved in register M as a three-digit decimal number (therefore up to 999 max)… BCDBIN alert!!

If the values match the routine defuses all activity markers and updates the header register.
If they don't then it's time to call [END0?] to search for more CASE/CASELSE instructions

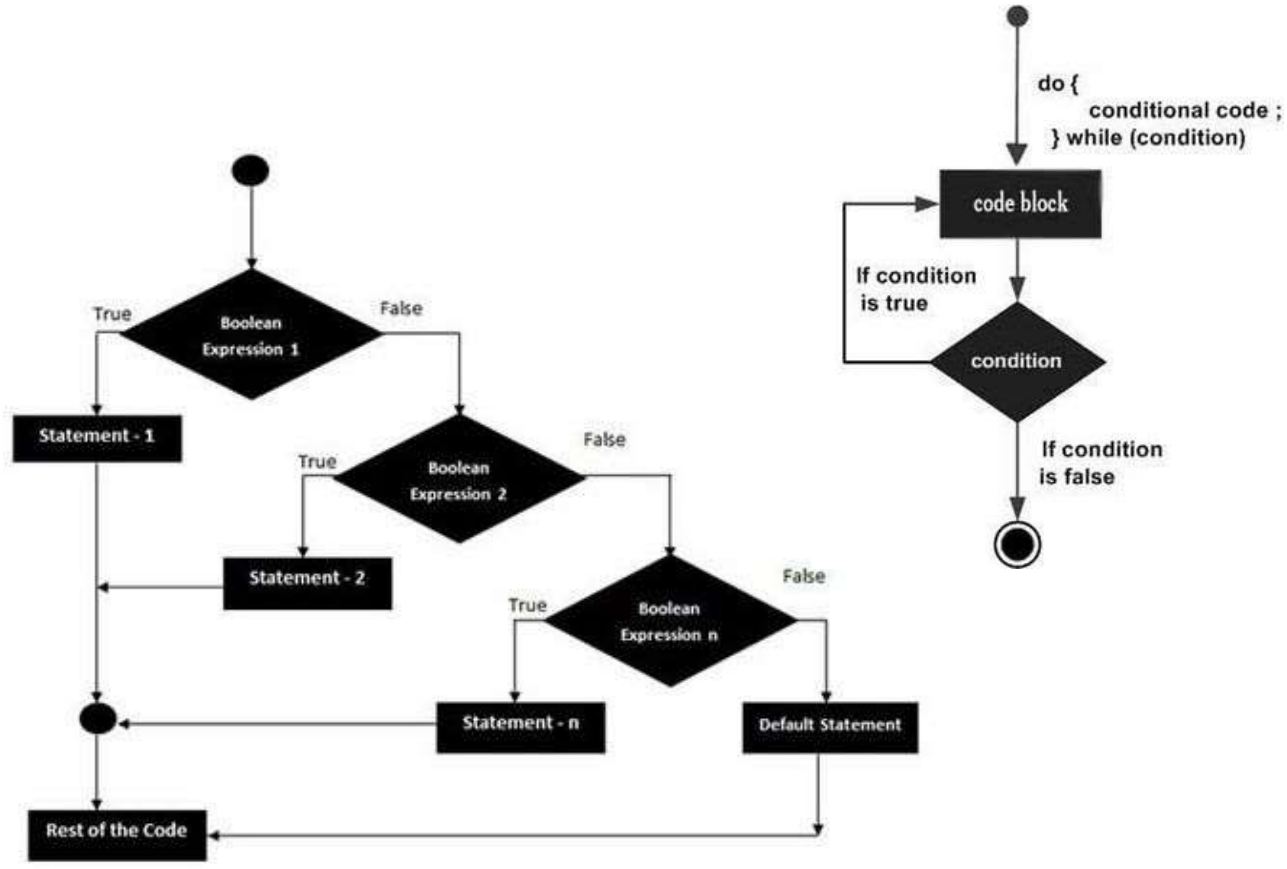
	AAAA	1BC	RCR 11	*put SLCT'ed Reg# in C.X*
	AAAB	266	C=C-1 S&X	*remove padding*
	AAAC	106	A=C S&X	
	AAAD	026	B=0 S&X	single register
	AAAE	*219*	*?NC XQ*	*check for valid addr*
	AAAF	*10C*	*->4386*	*[EXISTS] - selects Chip0*
	AAB0	0A6	A<>C S&X	
	AAB1	270	RAMSLCT	*select SELCT register*
	AAB2	038	READATA	*get register contents*
maximum is 999	AAB3	*38D*	*?NC XQ*	*convert C to Hex in C[S&X]*
	AAB4	*008*	*->02E3*	*[BCDBIN] - uses F8; selects Chip0*
	AAB5	10E	A=C ALL	*put in A for compares*
	AAB6	0B0	C=N ALL	get CASE VAL#
	AAB7	36E	?A#C ALL	value matches var# ?
	AAB8	***037***	***JC +06***	no, try next CASE or goto ENDSLCT
MATCH	AAB9	278	READ 9(Q)	*yes, deactivate SELCASE flag*
	AABA	270	RAMSLCT	*in case there are other CASEs below*
	AABB	038	READATA	read buffer header
	AABC	046	C=0 S&X	clear active flags
	AABD	***11B***	***JNC +35d***	*SAVE AND EXIT*
NOMATCH	AABE	**388**	**SETF 0**	*flags ENDSLCT found!*
	AABF	39C	PT= 0	
	AAC0	130	LDI S&X	
	AAC1	***0A3***	CON:	***CASE 2nd. byte***
	AAC2	058	G=C @PT,+	
	AAC3	*379*	*PORT DEP:*	*Search for other CASEs*
	AAC4	*03C*	*XQ*	*second pass!*
	AAC5	*08F*	*->A88F*	*[?END]*
NOTFND	AAC6	*01B*	*JNC +03*	*TRY CASELSE as well*
FOUND	AAC7	06A	A<>B PT<-	*put address in A<3:0>*
	AAC8	*2DB*	*JNC -37d*	*move the PC to the next CASE*
TRYELSE	AAC9	39C	PT= 0	
	AACA	130	LDI S&X	
	AACB	***0A4***	CON:	***CASELSE 2nd. byte***
	AACC	058	G=C @PT,+	
	AACD	*379*	*PORT DEP:*	*Search for other CASELSE*
	AACE	*03C*	*XQ*	*second pass!*
	AACF	*08F*	*->A88F*	*[?END]*
NOTFND	AAD0	*28B*	*JNC -47d*	*move PC to ENDSLCT*
FOUND	AAD1	*3B3*	*JNC -10d*	*move the PC to the next CASE*
SAVEXT	AAE0	*035*	*?NC GO*	*write updated header*
	AAE1	*126*	*->490D*	*[WRTSEL] - selects Chip0*

The PC gets moved either to the next CASE/CASELSE (if either one is found), or to ENDSLCT if we're at the end of the road. Note the two calls to [DECAD] prior to putting the PC value – this is due to the way [END0?] works, returning the address of the instruction *following* the sought-for instruction.

Finally, the ENDSLCT code is just a nominal deactivation of the active flag and clearing of the selected reg off the buffer header:

Header	AAD2	*094*	*"T"*		
Header	AAD3	*003*	*"C"*		***Deactivates SELCASE***
Header	AAD4	*00C*	*"L"*		*and clears all pointers*
Header	AAD5	*013*	*"S"*		
Header	AAD6	*004*	*"D"*		
Header	AAD7	*00E*	*"N"*		
Header	AAD8	*005*	*"E"*		*Ángel Martin*
ENDSLCT	**AAD9**	*369*	*?NC XQ*		*Check buffer id#7 - > header in C*
	AADA	*124*	*->49DA*		*[CHKBF#7] - returns addr in A.X*
	AADB	0A6	A<>C S&X		
	AADC	270	RAMSLCT		*select buf header*
	AADD	038	READATA		*get header content*
	AADE	25C	PT= 9		*deactivate SELCT flag*
	AADF	04A	C=0 PT<-		*and clear ADDR & REG#*
SAVEXT	AAE0	*035*	*?NC GO*		*write updated header*
	AAE1	*126*	*->490D*		*[WRTSEL] - selects Chip0*

That's all folks, hope you enjoyed this excursion throughout the "FOCAL+ extensions" – if nothing else surely particularly unusual applications on the HP-41 platform.



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 +	**0**1234	1.0000
2	G	2), + performed	01234	3.0000
	C	*	01**2**34	3.0000
	F	(, * with value saved	01234	3.0000
3	D	3 /	012**3**4	3.0000
4	G	4),/ performed, * with value recalled	01**2**34	0.7500
	A	+, * performed	**0**1234	2.2500
	F	(	01234	2.2500
5	E	5 ^	0123**4**	5.0000
	F	(, ^ with value saved	01234	5.0000
1	D	1 /	012**3**4	1.0000
2	G	2), / performed, ^ with value recalled	0123**4**	0.5000
	G	), ^ performed, + with value recalled	**0**1234	2.2361
	J or R/S	= final + performed	01234	4.4861

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

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

AOS Program Listing.

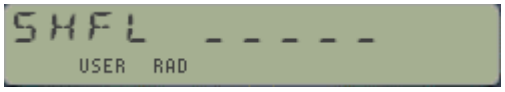
```
01 LBL "AOS"
02 SF 27
03 LBL a
04 CF 01
05 CF 02
06 CF 22
07 12
08 STO 23
09 -E
10 STO 24
11 CLX
12 RTN
13 LBL "+"
14 LBL A
15 61
16 GTO 00
17 LBL "-"
18 LBL B
19 51
20 GTO 00
21 LBL "*"
22 LBL C
23 42
24 GTO 00
25 LBL "/"
26 LBL D
27 32
28 GTO 00
29 LBL "YX"
30 LBL b
31 14
32 GTO 00
33 LBL "NEG"
34 LBL c
35 23
36 GTO 00
37 LBL "<"
38 LBL d
39 5
40 LBL 00
41 E1
42 /
43 STO 22
44 INT
45 X#0?
46 GTO 00
47 FS? 01
48 XEQ 03
49 LBL 00
50 RDN
51 FS?C 22
52 XEQ 02
53 RCL 22
54 INT
55 X=0?
56 GTO 00
57 LBL 07
58 RCL 24
59 X<0?
60 GTO 00
61 RCL IND 24
62 FRC
63 RCL 22
64 FRC
65 X>Y?
66 GTO 00
67 RCL IND 24
68 INT
69 X=0?
70 GTO 00
71 XEQ 01
72 GTO 07
73 LBL 00
74 ISG 24
75 ENTER^
76 RCL 24
77 13
78 X<=Y?
79 ASIN
80 RCL 22
81 STO IND 24
82 RCL IND 23
83 CF 01
84 GTO 99
85 LBL ">"
86 LBL e
87 E
88 STO 22
89 X<>Y
90 FS?C 22
91 XEQ 02
92 RCL 24
93 X<0?
94 SQRT
95 RCL IND 24
96 INT
97 X=0?
98 GTO 08
99 XEQ 01
100 GTO e
101 LBL 08
102 DSE 24
103 ENTER^
104 RCL IND 23
105 SF 01
106 GTO 99
107 LBL E
108 LBL "="
109 E
110 STO 22
111 X<>Y
112 FS?C 22
113 XEQ 02
114 RCL 24
115 X<0?
116 GTO 00
117 RCL IND 24
118 XEQ 01
119 GTO E
120 LBL 00
121 RCL IND 23
122 XEQ a
123 RDN
124 RDN
125 SF 22
126 GTO 99
127 LBL 02
128 ISG 23
129 ENTER^
130 21
131 RCL 23
132 -
133 X<0?
134 SQRT
135 RDN
136 STO IND 23
137 RCL 22
138 INT
139 X#0?
140 RTN
141 LBL 03
142 ISG 24
143 ENTER^
144 4.2
145 STO IND 24
146 RDN
147 RTN
148 LBL 01
149 RCL IND 23
150 DSE 23
151 RCL IND 23
152 X<>Y
153 XEQ IND Z
154 FS?C 02
155 ISG 23
156 ENTER^
157 RCL 23
158 13
159 -
160 X<0?
161 SQRT
162 X<>Y
163 STO IND 23
164 DSE 24
166 RTN
167 LBL 01
168 Y^X
169 RTN
170 LBL 02
171 CHS
172 LBL 00
173 SF 02
174 RTN
175 LBL 03
176 /
177 RTN
178 LBL 04
179 *
180 RTN
181 LBL 05
182 CHS
183 LBL 06
184 +
185 LBL 99
186 RTN
187 SF 22
188 END
```

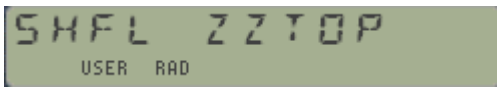
Appendix.-Stack Shuffling

This function has been moved to the WARP_Core module, where it has been enhanced and improved with a companion R0R4 shuffling counterpart. Refer to the WARP_Core manual for details.

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.*

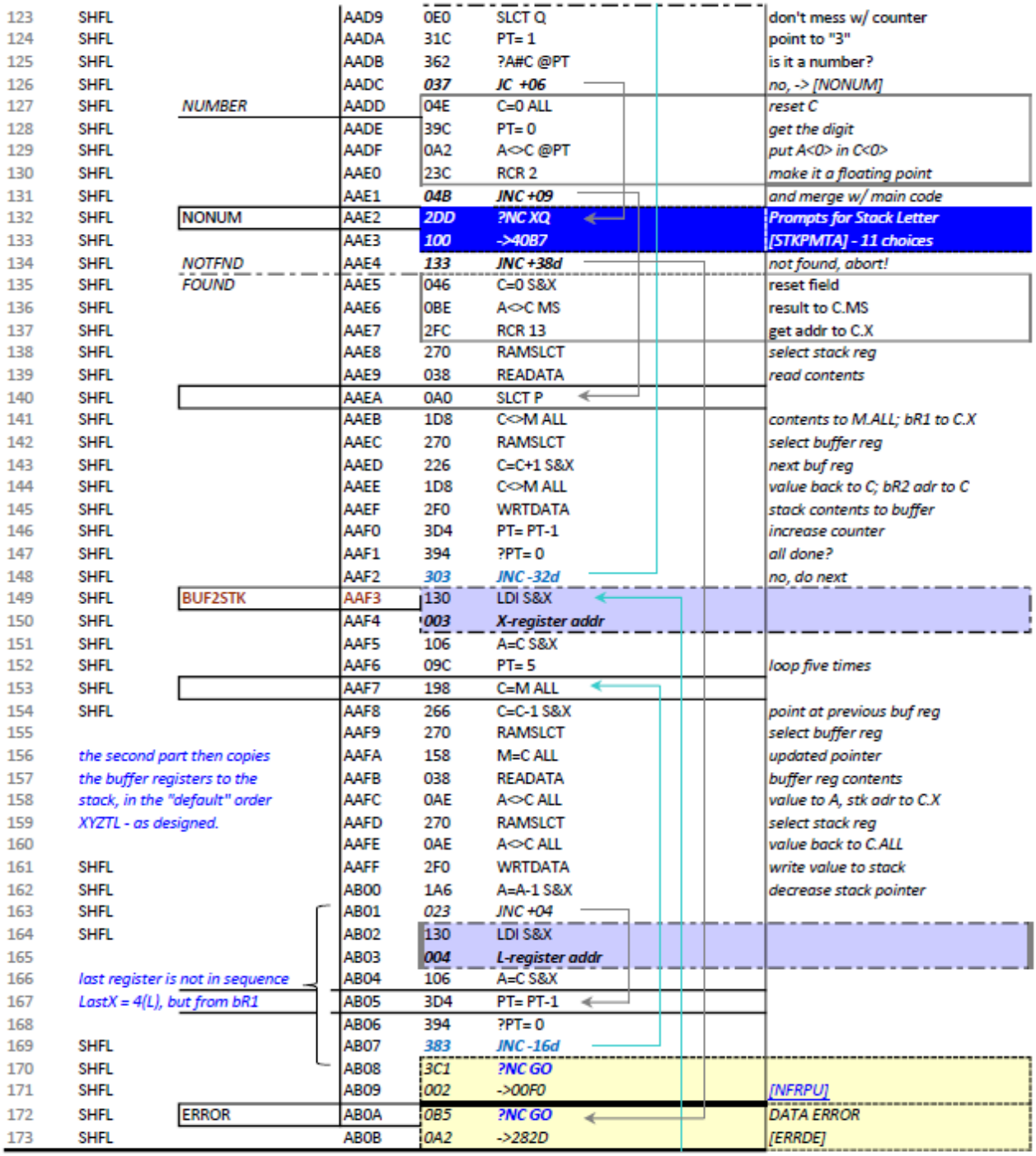
Note: You should be aware that **SHFL** uses the parameter buffer (id#=7) to hold a copy of the current stack registers after the shuffling. This could be useful to recall the previous values (basically an UNDO facility) but will conflict with your parameter assignments using **LET=** if you have made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

Note: for the sake of completion the original SHUFL MCODE is attached in the next few pages. As mentioned before, this function is now superseded with the more capable version included in the WARP Core module.

1	SHFL	STKBLD	AA5F	14C	?FSET 6	SHIFT key?
2	SHFL		AA60	17D	?C GO	yes, toggle SHIFT and RTN
3	SHFL		AA61	12F	->4B5F	[TGSHF4]
4	SHFL	NOSHFT	AA62	36D	?NC XQ	get ALPHA key code
5	SHFL		AA63	07C	->1FDB	[GTACOD]
6	SHFL		AA64	106	A=C S&X	master to compare to
7	SHFL		AA65	130	LDI S&X	
8	SHFL		AA66	03A	1 more than "9"	numeric digits
9	SHFL		AA67	362	?A#C @PT	is digit<>1 numeric mask?
10	SHFL		AA68	01F	JC +03	no, skip
11	SHFL		AA69	246	C=A-C S&X	is key>3A (3D and 3F)
12	SHFL		AA6A	027	JC +04	no, ->[NUMBER]
13	SHFL	NONUM	AA6B	2DD	?NC XQ	no, Prompt for Stack Letter
14	SHFL	NOTFND	AA6C	100	->40B7	[STKPMTA] - 11 choices
15	SHFL	FOUND	AA6D	3E0	RTN	
16	SHFL	NUMBER	AA6E	221	?NC XQ	ASCII char put to LCD
17	SHFL		AA6F	0B0	->2C88	[MASK]
18	SHFL		AA70	14D	?NC GO	Skip one line and RTN
19	SHFL		AA71	032	->0C53	[SKIP1]
20	SHFL	ADIOS	AA72	30D	?NC GO	abandon ship
21	SHFL		AA73	122	->48C3	[BAILOUT
22	SHFL	Header	AA74	08C	"L"	
23	SHFL	Header	AA75	006	"F"	Stack Shuffle
24	SHFL	Header	AA76	008	"H"	Supports Digit entry (!)
25	SHFL	Header	AA77	013	"S"	Ángel Martín
26	SHFL	SHFL	AA78	2CC	?FSET 13	RUN'ing a program?
27	SHFL		AA79	19F	JC +51d	yes, proceed
28	SHFL		AA7A	04C	?FSET 4	SST'ing a program?
29	SHFL		AA7B	18F	JC +49d	yes, proceed
30	SHFL	MANUAL	AA7C	158	M=C ALL	get function address
31	SHFL		AA7D	239	?NC XQ	Display Function Name
32	SHFL		AA7E	108	->428E	[PMTFNM]
33	SHFL		AA7F	013	JNC +02	LB_A440
34	SHFL	REPMT5	AA80	3B8	READ 14(d)	remove rightmost chr
35	SHFL	PRMPT5	AA81	130	LDI S&X	5-prompts
36	SHFL		AA82	01F	"_"	underscore chr
37	SHFL		AA83	3E8	WRIT 15(e)	
38	SHFL		AA84	3E8	WRIT 15(e)	
39	SHFL		AA85	12D	?NC XQ	5-prompts
40	SHFL		AA86	038	->0E4B	[NEXT3]
41	SHFL	BCKARW	AA87	35B	JNC -21d	[ABTSEQ]
42	SHFL	OTHER	AA88	28C	?FSET 7	Radix key pressed?
43	SHFL		AA89	04B	JNC +09	no, ignore
44	SHFL	SHFL0	AA8A	3BD	?NC XQ	Message Line
45	SHFL		AA8B	01C	->07EF	[MESSL]
46	SHFL		AA8C	018	"X"	
47	SHFL		AA8D	019	"Y"	
48			AA8E	01A	"Z"	"XYZTL"
49	quickest way to create a		AA8F	014	"T"	
50	"shadow stack" in the buffer		AA90	20C	"L"	
51			AA91	15B	JNC +43d	
52	SHFL		AA92	379	PORT DEP?	Select Stack Reg/ Capped!
53	SHFL		AA93	03C	XQ	and puts it in LCD
54	SHFL		AA94	25F	->AA5F	[STKBLD]
55	SHFL	NOSTK	AA95	363	JNC -20d	[PRMPT5]
56	SHFL	STACK	AA96	013	JNC +02	[PRMPT4]
57	SHFL	REPMT4	AA97	3B8	READ 14(d)	remove rightmost chr
58	SHFL	PRMPT4	AA98	130	LDI S&X	4-prompts
59	SHFL		AA99	01F	"_"	underscore chr
60	SHFL		AA9A	3E8	WRIT 15(e)	add the fourth one
61	SHFL		AA9B	12D	?NC XQ	4-prompts

62	SHFL		AA9C	038	->0E4B	[NEXT3]
63	SHFL	BCKARW	AA9D	31B	JNC -29d	[REPMT5]
64	SHFL	OTHER	AA9E	379	PORT DEP:	Select Stack Reg/ Capped!
65	SHFL		AA9F	03C	XQ	and puts it in LCD
66	SHFL		AAA0	25F	->AA5F	[STKBLD]
67	SHFL	NOSTK	AAA1	3B8	JNC -09	[PRMPT4]
68	SHFL	STACK	AAA2	013	JNC +02	[PRMPT3]
69	SHFL	REPMT3	AAA3	3B8	READ 14(d)	remove rightmost chr
70	SHFL	PRMPT3	AAA4	12D	?NC XQ	3-prompts
71	SHFL		AAA5	038	->0E4B	[NEXT3]
72	SHFL	BCKARW	AAA6	38B	JNC -15d	REPMT4
73	SHFL	OTHER	AAA7	379	PORT DEP:	Select Stack Reg/ Capped!
74	SHFL		AAA8	03C	XQ	and puts it in LCD
75	SHFL		AAA9	25F	->AA5F	[STKBLD]
76	SHFL	NOSTK	AAAA	3D3	JNC -06	PRMPT3
77	SHFL	STACK	AAAB	01B	JNC +03	PRMPT2
78	SHFL		AAAC	0BB	JNC +28d	
79	SHFL	REPMT2	AAAD	3B8	READ 14(d)	remove rightmost chr
80	SHFL	PRMPT2	AAAE	121	?NC XQ	2-prompts
81	SHFL		AAAF	038	->0E48	[NEXT2]
82	SHFL	BCKARW	AAB0	39B	JNC -13d	[REPMT3]
83	SHFL	OTHER	AAB1	379	PORT DEP:	Select Stack Reg/ Capped!
84	SHFL		AAB2	03C	XQ	and puts it in LCD
85	SHFL		AAB3	25F	->AA5F	[STKBLD]
86	SHFL		AAB4	3D3	JNC -06	[PRMPT2]
87	SHFL	PRMPT1	AAB5	115	?NC XQ	1-prompt
88	SHFL		AAB6	038	->0E45	[NEXT1]
89	SHFL	BCKARW	AAB7	3B3	JNC -10d	[REPMT2]
90	SHFL	OTHER	AAB8	379	PORT DEP:	Select Stack Reg/ Capped!
91	SHFL		AAB9	03C	XQ	and puts it in LCD
92	SHFL		AABA	25F	->AA5F	[STKBLD]
93	SHFL	NOSTK	AABB	3D3	JNC -06	[PRMPT1]
94	SHFL	STACK	AABC	3DD	?NC XQ	Left Justified format
95	SHFL		AABD	0AC	->2BF7	[LEFTJ]
96	SHFL		AABE	319	?NC XQ	last chance to cancel out
97	SHFL		AABF	038	->0EC6	[NULTST]
98	SHFL		AAC0	138	READ 4(L)	get rid of "SHFL: "
99	SHFL		AAC1	3F8	READ 15(e)	in two strokes!
100	SHFL	SHFL1	AAC2	169	?NC XQ	Enable RAM & Reset Seq
101	SHFL		AAC3	124	->495A	[EXIT4]
102	SHFL		AAC4	04E	C=0 ALL	
103	SHFL		AAC5	168	WRIT 5(M)	
104	SHFL		AAC6	3A9	?NC XQ	Copy Display to Alpha
105	SHFL		AAC7	108	->42EA	[NXTCHR]
106	SHFL	SST/RUN	AAC8	178	READ M(5)	get master string
107	SHFL		AAC9	2EE	?C#0 ALL	
108	SHFL		AACA	3A0	?NC RTN	
109			AACB	0EE	C<>B ALL	put in B.ALL
110	this first part copies the		AACC	369	?NC XQ	Check buffer id#7 - > header in C
111	stack registers to the buffer		AACD	124	->49DA	[CHKBF#7] - returns addr in A.X
112	in the order defined by the		AACE	0A6	A<>C S&X	header address
113	master string (prompt)		AACF	226	C=C+1 S&X	bR1 adr
114			AAD0	158	M=C ALL	save for later
115	SHFL		AAD1	09C	PT= 5	will loop five times
116	SHFL	STK2BUF	AAD2	0CE	C=B ALL	get master string
117	SHFL		AAD3	106	A=C S&X	put NIBBLE in A
118	SHFL		AAD4	016	A=0 XS	neuter the XS nibble
119	SHFL		AAD5	23C	RCR 2	rotate two
120	SHFL		AAD6	0EE	C<>B ALL	update master string
121	SHFL		AAD7	130	LDI S&X	
122	SHFL		AAD8	030	"0"	numeric mask



123	SHFL		AAD9	0E0	SLCT Q	don't mess w/ counter
124	SHFL		AADA	31C	PT= 1	point to "3"
125	SHFL		AADB	362	?A#C @PT	is it a number?
126	SHFL		AADC	037	JC +06	no, -> [NONUM]
127	SHFL	NUMBER	AADD	04E	C=0 ALL	reset C
128	SHFL		AADE	39C	PT= 0	get the digit
129	SHFL		AADF	0A2	A<>C @PT	put A<0> in C<0>
130	SHFL		AAE0	23C	RCR 2	make it a floating point
131	SHFL		AAE1	04B	JNC +09	and merge w/ main code
132	SHFL	NONUM	AAE2	2DD	?NC XQ	Prompts for Stack Letter
133	SHFL		AAE3	100	->40B7	[STKPMTA] - 11 choices
134	SHFL	NOTFND	AAE4	133	JNC +38d	not found, abort!
135	SHFL	FOUND	AAE5	046	C=0 S&X	reset field
136	SHFL		AAE6	0BE	A<>C MS	result to C.MS
137	SHFL		AAE7	2FC	RCR 13	get addr to C.X
138	SHFL		AAE8	270	RAMSLCT	select stack reg
139	SHFL		AAE9	038	READATA	read contents
140	SHFL		AAEA	0A0	SLCT P	
141	SHFL		AAEB	1D8	C<>M ALL	contents to M.ALL; bR1 to C.X
142	SHFL		AAEC	270	RAMSLCT	select buffer reg
143	SHFL		AAED	226	C=C+1 S&X	next buf reg
144	SHFL		AAEE	1D8	C<>M ALL	value back to C; bR2 adr to C
145	SHFL		AAEF	2F0	WRTDATA	stack contents to buffer
146	SHFL		AAF0	3D4	PT= PT-1	increase counter
147	SHFL		AAF1	394	?PT= 0	all done?
148	SHFL		AAF2	303	JNC -32d	no, do next
149	SHFL	BUF2STK	AAF3	130	LDI S&X	
150	SHFL		AAF4	003	X-register addr	
151	SHFL		AAF5	106	A=C S&X	
152	SHFL		AAF6	09C	PT= 5	loop five times
153	SHFL		AAF7	198	C=M ALL	
154	SHFL		AAF8	266	C=C-1 S&X	point at previous buf reg
155			AAF9	270	RAMSLCT	select buffer reg
156	the second part then copies		AAFA	158	M=C ALL	updated pointer
157	the buffer registers to the		AAFB	038	READATA	buffer reg contents
158	stack, in the "default" order		AAFC	0AE	A<>C ALL	value to A, stk adr to C.X
159	XYZTL - as designed.		AAFD	270	RAMSLCT	select stack reg
160			AAFE	0AE	A<>C ALL	value back to C.ALL
161	SHFL		AAFF	2F0	WRTDATA	write value to stack
162	SHFL		AB00	1A6	A=A-1 S&X	decrease stack pointer
163	SHFL		AB01	023	JNC +04	
164	SHFL		AB02	130	LDI S&X	
165			AB03	004	L-register addr	
166	last register is not in sequence		AB04	106	A=C S&X	
167	LastX = 4(L), but from bR1		AB05	3D4	PT= PT-1	
168			AB06	394	?PT= 0	
169	SHFL		AB07	383	JNC -16d	
170	SHFL		AB08	3C1	?NC GO	
171	SHFL		AB09	002	->00F0	[NFRPU]
172	SHFL	ERROR	AB0A	0B5	?NC GO	DATA ERROR
173	SHFL		AB0B	0A2	->282D	[ERRDE]