HP-41 FAOS ROM

Algebraic Operation for the HP-41

Introduction.

This module includes a set of three FOCAL programs implementing Algebraic Operation on the HP-41. The programs are independent from one another and can be used indistinctly.

In addition to the three main AOS programs the module also includes utility programs for "Function Interpretation" using the ALPHA register. In many respects this approach has been superseded by the *Formula Evaluation Module*, but it's still interesting to document the different solutions contributed by the user community along the years.

Another group of routines are about using X-Memory registers in a more flexible way – similar to standard data registers in main memory, and even as extended stack for complete ALPHA strings. Here too these Focal routines have been superseded by the MCODE implementation in the *X-Mem Twin module*.

Finally, and somehow in an opposite kind of way, the module also includes Valentin Albillo's STKN program that simulates a N-Level RPN stack.

ROM Contents

The table below summarizes the main programs by category.

Program(s)	Category	Author
AOSXM	AOS	*Greg McClure*
AOS57	AOS	*Thomas Klem*
AOS67	AOS	*Jim Horn*
FLREG, FLSTO, FLRCL…	XM Registers	*Peter Reiter*
ASINIT, APOP, APUSH	XM ALPHA Stack	*Godwin Stewart*
SIZA, ASTOA, A<>	XM ALPHA Stack	*Tyann*
AFI, AFI+	Formula	*Erik Christensen*
FRMLA	Formula	*Stefan Fegert*
STKN	RPN Stack	*Valentín Albillo*

All authors listed should be credited for the programs. The sources include other ROMS like the GJM ROM, HP67FUN, and diverse user forums (MoHP and SwissMicros) and publications like the PPC Journal. Refer to the individual program description for more details.

AOS Simulator using X-Mem ; by Greg McClure

Written by Greg McClure, this FOCAL program was first released in the GJM ROM and it opens the set of AOS implementations in this module.

The AOSXM (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 "AOSXM"	Reset AOSXM	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).

B.6 Program Listing

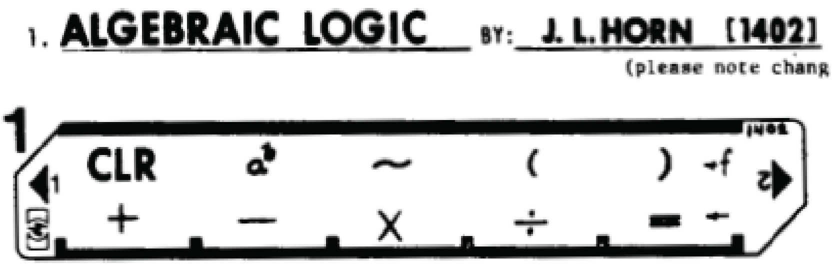
```
01 LBL "AOSXM"
02 RAD
03 SF 27
04 "AOS"
05 SF 25
06 FLSIZE
07 FS?C 25
08 GTO 00
09 "SIZE?"
10 PROMPT
11 "AOS"
12 CRFLD
13 GTO 01
14 *LBL 00
15 RCLFLAG
16 FIX 0
17 X<>Y
18 "NEW SZ <"
19 ARCL X
20 "`>?"
21 X<>Y
22 STOFLAG
23 RDN
24 CF 22
25 PROMPT
26 FC? 22
27 GTO 01
28 CHS
29 RESZFL
30 *LBL 01
31 CLST
32 CLA
33 SEEKPT
34 X<>F
35 X<> L
36 +
37 XEQ F
38 XEQ G
39 GTO 12
40 *LBL 14
41 FS?C 04
42 Y^X
43 RTN
44 *LBL 13
45 XEQ 14
46 FS?C 03
47 /
48 FS?C 02
49 *
50 RTN
51 *LBL 12
52 XEQ 13
53 FS?C 01
54 -
55 FS?C 00
56 +
57 RTN
58 GTO 09
59 *LBL A           ' Addition
60 XEQ 12
61 SF 00
62 RTN
63 GTO 09
64 *LBL B           ' Subtraction
65 XEQ 12
66 SF 01
67 RTN
68 GTO 09
69 *LBL C           ' Multiplication
70 XEQ 13
71 SF 02
72 RTN
73 GTO 09
74 *LBL D           ' Division
75 XEQ 13
76 SF 03
77 RTN
78 GTO 09
79 *LBL E           'Power
80 XEQ 14
81 SF 04
82 RTN
83 *LBL J           'R/S
84 *LBL 09          'Core routine
85 XEQ G
86 FC? 05
87 RTN
88 GTO 09
89 *LBL 11
90 SAVEX
91 CLX
92 +
93 RTN
94 *LBL F           '"Close Parenthesis
95 SF 05
```

```
96   ENTER^
97   RDN
98   FS? 04
99   XEQ 11
100  FS? 03
101  XEQ 11
102  FS? 02
103  XEQ 11
104  FS? 01
105  XEQ 11
106  FS? 00
107  XEQ 11
108  CLX
109  X<>F
110  XEQ 11
111  R^
112  RTN
113  GTO 09
114 *LBL 10
115  STO M
116  CLX
117  RCLPT
118  DSE X
119  ""
120  SEEKPT
121  X<> M
122  GETX
123  X<> M
124  SEEKPT
125  CLX
126  X<> M
127  RTN
128 *LBL G               'Open Parenthesis
129  XEQ 12
130  RCLPT
131  X=0?
132  GTO 00
133  RDN
134  XEQ 10
135 *LBL 00
136  X<>F
137  RDN
138  ENTER^
139  ENTER^
140  ENTER^
141  FS? 00
142  XEQ 10
143  FS? 01
144  XEQ 10
145  FS? 02
146  XEQ 10
147  FS? 03
148  XEQ 10
149  FS? 04
150  XEQ 10
151  R^
152  RTN
153  GTO J
154  END
```

Note that the version in the module includes global labels for the main arithmetic operations, LBL "+", LBL "-", LBL "*", LBL "/", and LBL "^".

AOS Program (LBL "AOS")

History: This is based on the original AOS (Algebraic Operating System) program written by Jim Horn appearing in 65 Notes V4N10P25 for the write-up and on page 35 for the program itself. It incorporates an error fix Jim noted in PPC Notes V5N3P5. One of the adjustments made was to add the alpha labels allowing each function so these labels can be assigned to the associated keys in USER mode. This allows the calculator to function as if it were in AOS mode while in USER mode. The picture below is from V4N10P25 of 65 Notes.



Object: Allow the calculator to mimic functioning as an AOS-style machine rather than RPN. The author, Jim Horn, has graciously given these comments about the program:

> "The one or two digit codes given to each key in the program listing consist of a units digit that gives the operator hierarchy and the rest that gives a unique ID to each operator that can later be executed via an indirect XEQ. For instance, the A key is shown as the addition key, so LBL A and LBL "+" handle that by putting 61 in the X register and jumping ahead to the main handler (LBL 00, step 40). The 61 indicates that addition will happen by doing an XEQ 06 with a priority level of 1, putting it below multiplication and division (2), negation (3) or parenthesis and powers (always the highest, in this case, 4). As another example, the code entered for Y^X is 14, so execution will transfer to LBL 01 with a priority of 4.
>
> Flag 1 has the important role of noting when an implied multiply is needed and sees that it gets provided. Flag 2 indicates a unary (single operator) function. If not set, a function is treated as binary (two operator).
>
> There are two stacks set up in memory. R0 through R12 are the operator stack; R13 through R21 are the operand stack. The HP-67's remaining registers were used for the Last Operator, the Operator Stack Pointer, the Operand Stack Pointer, and the index register for all of the above.
>
> The algorithm was from a flow chart in a programming text from 1971. After I wrote the program and it was published in 65 Notes, I discovered that the book's flowchart and algorithm were incomplete and flawed. Thus the NOP published several issues later."

Running the program:

1) XEQ ALPHA AOS ALPHA
2) Optional: If you are using an HP 41CX or have the Extended Functions module for the HP 41 series, key in and use the auxiliary AOSKY program included after the AOS program listing to assign the labels in this program file to the "natural" keys to use the calculator more "normally." Or assign them manually.
   a. This program assigns the global labels in the AOS program file as follows:
      - LBL "+" to the add key, LBL "–" to the subtract key, LBL "*" to the multiply key, and LBL "/" to the divide key.
      - LBL "=" to the ENTER key
      - LBL "YX", the algebraic Y^X function, to the shifted location of Y^X
      - LBL "<", the label for open parenthesis, to the X<>Y key
      - LBL ">", the label for the close parenthesis, to the RDN (Roll Down) key, and
      - LBL "NEG" (Negate – not quite CHS) to the shifted location of the CHS key.
      - LBL "AOS" to the shift of the backarrow key to use as a "CLR" style function similar to LBL a.
   b. Once you are done using the AOS program, execute CLKEYS to return the keyboard to normal.
3) The AOS program can handle up to 13 pending operations, counting all open parentheses as an operation, and up to 8 pending operands. If you exceed these, you will generate an error.
4) Note: While the program is not running, any built-in HP calculator math function can be executed on the displayed data. If, however, you need to execute a function on a key that has been reassigned or that is mapped to a local label in USER mode, you will need to press the USER mode rocker switch at the top of the calculator, execute the desired function, and then press the USER mode switch again before continuing with a R/S.
5) In fact, you can do any manual calculations in RPN mode you like (probably out of USER mode to keep things simple) and then when you are ready to return to the AOS program, make sure USER mode is set and press R/S. This is possible because LBL 99 in the code shown below is the common exit point for all operations. If you look

at the code, pressing R/S when the code stops after LBL 99 will set flag 22 so that the result of a manually executed math function can be used for subsequent calculations in the program as you resume it.

6) Running the program is perhaps best illustrated by the examples below.

Example 1: Evaluate 1 + 2 x 3 ^{4}

	See	Press (Assumes key assignments)	Press (without key assignments)
1)		XEQ ALPHA AOS ALPHA	XEQ ALPHA AOS ALPHA
2)	0.00	1 +	1 A
3)	1.00	2 x	2 C
4)	2.00	3 shift Y^X	3 shift b
5)	3.00	4 ENTER	4 E
6)	163.00		

Example 2: Evaluate (1 + 2) x (3 – (-4 / (5 – 6))) =

	See	Press (Assumes key assignments)	Press (without key assignments)
1)		XEQ ALPHA AOS ALPHA	XEQ ALPHA AOS ALPHA
2)	0.00	X<>Y	shift d
3)	0.00	1 +	1 A
4)	1.00	2 RDN (Note: Flag 1 is set)	2 shift e
5)	3.00	x (Note: Flag 1 is cleared)	C
6)	3.00	X<>Y	shift d
7)	3.00	3 –	3 B
8)	3.00	X<>Y	shift d
9)	3.00	4 CHS /	4 CHS D
10)	4.00	X<>Y	shift d
11)	4.00	5 –	5 B
12)	5.00	6 ENTER	6 E
13)	21.00		

Example 2 notes: 1) The last closing parentheses are automatically supplied by pressing E (or ENTER).
2) To enter a negative number, simply press the CHS key. The working of the NEG function is demonstrated by example 3 below.

Example 3: Evaluate 5 – (9 x 3) using the NEG function without using parentheses.

	See	Press (Assumes key assignments)	Press (without key assignments)
1)		XEQ ALPHA AOS ALPHA	XEQ ALPHA AOS ALPHA
2)	0.00	9 x	9 C
3)	9.00	3 ENTER	3 E
4)	27.00	shift CHS	shift c (This is the NEG function)
5)	27.00	+	A
6)	–27.00	5 ENTER	5 E (The -27.00 reflects the NEG)
7)	–22.00		

Example 3 note: NEG takes the previous result and makes it negative but does not display it until the next operation is performed.

Example 4: Evaluate 5 – (9 x 3) using the built-in CHS function.

	See	Press (Assumes key assignments)	Press (without key assignments)
1)		XEQ ALPHA AOS ALPHA	XEQ ALPHA AOS ALPHA
2)	0.00	9 x	9 C
3)	9.00	3 ENTER	3 E
4)	27.00	CHS R/S	CHS R/S
5)	–27.00	+	A
6)	–27.00	5 ENTER	5 E
7)	–22.00		

Example 4 note: After pressing CHS in step 4, the R/S key must be pressed so that flag 22, the numeric input flag, is set. This indicates to the program that an entry has been made allowing it to process the input properly.

Example 5: Evaluate $M = \sqrt{5\left[\left(\left\{\left[\left(1+0.2\left[\frac{350}{661.5}\right]^2\right)^{3.5}-1\right]\left[1-(6.875\times10^{-6})25500\right]^{-5.2656}\right\}+1\right)^{0.286}-1\right]}$

	See	Press (With key assignments)	Press (No key assignments)	Comments
1)		shift FIX 4	shift FIX 4	Show 4 decimals
2)		XEQ ALPHA AOS ALPHA	XEQ ALPHA AOS ALPHA	
3)	0.0000	5 x	5 C	5 x
4)	5.0000	X<>Y	shift d	(
5)	5.0000	X<>Y	shift d	(
6)	5.0000	X<>Y	shift d	(
7)	5.0000	X<>Y	shift d	(
8)	5.0000	X<>Y	shift d	(
9)	5.0000	1 +	1 A	1 +
10)	1.0000	0.2 x	0.2 C	0.2 x
11)	0.2000	X<>Y	shift d	(
12)	0.2000	350 /	350 D	350 /
13)	350.0000	661.5 RDN	661.5 shift e	661.5)
14)	0.5291	shift Y^X	shift b	Y^X
15)	0.5291	2 RDN	2 shift e	2)
16)	1.0560	shift Y^X	shift b	Y^X
17)	1.0560	3.5 –	3.5 B	3.5 –
18)	1.2101	1 RDN	1 shift e	1)
19)	0.2101	x	C	x
20)	0.2101	X<>Y	shift d	(
21)	0.2101	1 –	1 B	1 –
22)	1.0000	6.875 EEX 6 CHS x	6.875 EEX 6 CHS C	.000006875 x
23)	6.8750 –06	25500 RDN	25500 shift e	25500)
24)	0.8247	shift Y^X	shift b	Y^X
25)	0.8247	5.2656 CHS RDN	5.2656 CHS shift e	–5.2656)
26)	0.5796	+	A	+
27)	0.5796	1 RDN	1 shift e	1)
28)	1.5796	shift Y^X	shift b	Y^X
29)	1.5796	0.286 –	0.286 B	0.286 –
30)	1.1397	1 RDN	1 shift e	1)
31)	0.1397	=	E	=
32)	0.6984	USER SQRT USER	USER SQRT USER	SQRT
33)	0.8357			(Mach 0.8357)

Example 5 note: This is the famous "Mach Number" formula from many past RPN vs. AOS illustrations. With this AOS program, you can now choose either way to approach this problem. In AOS, you will need to do the square root last so begin with the 5 x portion of the formula. To execute the square root, notice how USER mode must be turned off, the square root key pressed, and then USER turned back on. For the power of 2 in the formula, you can turn USER off and execute the X^2 function and turn USER back on, or you can simply use shift Y^X and then 2.

Example 6: Evaluate $(10 / 2)^2 + 1$ using the built-in X^2 function.

	See	Press (Assumes key assignments)	Press (without key assignments)
1)		XEQ ALPHA AOS ALPHA	XEQ ALPHA AOS ALPHA
2)	0.00	X<>Y 10 /	shift d 10 D
3)	10.00	2 RDN	2 shift e
4)	5.00	USER X^2 USER	USER X^2 USER
5)	25.00	+	A
6)	5.00		

Note: This is incorrect at this point. Because the access to the built-in X^2 function did not occur with flag 22 set, the AOS program does not detect it properly. Flag 22 was set upon the entry of the "2" in line 3 above, but pressing RDN or executing shift e (to close the parenthesis) does not preserve flag 22. Flag 22 is how the AOS program determines if a number displayed has changed.
The way to ensure it functions correctly is shown in example 7 below.

Example 7: Evaluate (10 / 2)^2 + 1 using the built-in X^2 function.

	See	Press (Assumes key assignments)	Press (without key assignments)
1)		XEQ ALPHA AOS ALPHA	XEQ ALPHA AOS ALPHA
2)	0.00	X<>Y 10 /	shift d 10 D
3)	10.00	2 RDN	2 shift e
4)	5.00	USER X^2 USER	USER X^2 USER
5)	25.00	R/S	R/S
6)	25.00	+	A
7)	25.00	1 ENTER	1 E
8)	6.00		

Note: This is correct. Pressing R/S enables the AOS program to detect the used computed square of 5 and use it for further computations.

Therefore, it is probably always safer to press R/S after making any calculations outside of the program itself.

Specifics:

1) Program is 350 bytes long.
2) Program is XROM 23,14 for the AOS label. XROMs 23,14 through 23,23 are used in this program file.
3) Uses registers 00 – 24. SIZE 025 required.
   - 00 – 12: Operator value
   - 13 – 21: Pending operator stack
   - 22 – Last operator
   - 23 – Stack pointer for values
   - 24 – Stack pointer for operations
4) Labels used:
   - 00 – 09: used.
   - 99 – Common exit point for all operations
   - A and "+" – Addition
   - B and "-" – Subtraction
   - C and "*" – Multiplication
   - D and "/" – Division
   - E and "=" – Equals key
   - a and "AOS" – Clear AOS calculator
   - b and "YX" – Y^X function
   - c and "NEG" – Negate function
   - d and "<" – Open parenthesis
   - e and ">" – Close parenthesis. Note: At the end of a calculation, press E or = key assignment instead.
5) Flags used:
   - Flag 1 has the important role of noting when an implied multiply is needed and sees that it gets provided.
   - Flag 2 indicates a unary (single operator) function. If not set, a function is treated as binary (two operator).
   - Flag 22 is used to detect user numeric input.
   - Flag 27 is set.
6) Display mode: Display mode is not changed. The existing display mode is retained.

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 -1
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 10
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 1
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 1
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
165 RTN
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
```

Auxiliary Program Listing:

```
01 LBL "AOSKY"
02 "AOS"
03 -44
04 PASN
05 "-"
06 51
07 PASN
08 "+"
09 61
10 PASN
11 "*"
12 71
13 PASN
14 "/"
15 81
16 PASN
17 "="
18 41
19 PASN
20 "YX"
21 -12
22 PASN
23 "<"
24 21
25 PASN
26 ">"
27 22
28 PASN
29 "NEG"
30 -42
31 PASN
32 CLST
33 END
```

Algebraic Operation System (AOS) ; by Thomas Klem

https://www.hpmuseum.org/forum/thread-18271.html

Description

This program allows you to use the *Algebraic Operation System* (AOS) similar to how old *Texas Instruments* calculators work.

The *shunting yard algorithm* is used with a data and an operator stack.
Their stack size is configurable and is only limited by the amount of memory available.

Functions

The functions just operate on the **X** register in postfix notation.
This is how the *TI-57* and other older calculators from *Texas Instruments* work.

For example, to calculate √(3^2+4^2) use:

```
3 x^2 + 4 x^2 = √x
```

Alternatively we can use:

```
( 3 x^2 + 4 x^2 ) √x
```

However, this requires one more keystroke.

Apparently we use a mixture of infix notation for arithmetic operations and postfix notation for functions.

Change Sign

It behaves similarly to an ordinary function.

E.g. an expression like −3^4 has to be keyed in like:

```
3 y^x 4 = +/-
```

Or alternatively:

```
( 3 y^x 4 ) +/-
```

Intermediate Results

The intermediate results of a calculation are viewed and may also be printed.

Example

$$\frac{1\times2+3\times4+5\times6+7\times8}{4}$$

```
( 1 * 2 + 3 * 4 + 5 * 6 + 7 * 8 ) / 4 =
```

```
ST X=       2
ST X=      12
ST X=      14
ST X=      30
ST X=      44
ST X=      56
ST X=     100
ST X=      25
```

Implicit Data Entry

The current value in the **X** register is used as data entry.

This allows to reuse the first entry:

```
3 + =
```

This results in 3+3 = 6.

```
3 * * =
```

This results in 3^4 = 81.

The Monster Formula

The formula is from [A case against the x<>y key]:

$$1 - 2 \times 3^4 \div 5 + \sin\left(6 - \sqrt[3]{7^2}\right) \times 8! + \ln\left[\left(-9^{2^3} \times 45^{\frac{6}{7}}\right)^2\right]$$

Here's how it is entered with this program.

```
1 - 2 * 3 ^ 4 / 5 + ( 6 - 7 X^2 ^ 3 1/X ) SIN * 8 FACT + ( 9 ^
2 ^ 3 * 45 ^ ( 6 / 7 ) ) CHS X^2 LN =
```

1657.008948

Intermediate Results

```
ST X=               81
ST X=              162
ST X=             32.4
ST X=            -31.4
ST X=    3.65930571002
ST X=    2.34069428998
ST X=    1646.72764773
ST X=    1615.32764773
ST X=                8
ST X=         43046721
ST X= 8.57142857143E-1
ST X=    26.1239772883
ST X=    1124551561.74
ST X=    1657.00894809
```

Registers

This is a list of the registers after the calculation:

```
00:                5
01:               10
02:             -4.1
03:                0
04:                0
05:                0
06:    1615.32764773
07:         43046721
08:               45
09:                6
10:                0
```

Mark Hardman's solution

(05-10-2015 03:12 PM)**Mark Hardman Wrote:** _

Code:

```
x             y          z          t
1 [Enter]     1          -          -          -
3 [Enter]     3          1          -          -
4             4          3          1          -
y^x          81          1          -          -
2             2         81          1          -
x           162          1          -          -
5             5        162          1          -
/            32.4        1          -          -
-           -31.4        -          -          -
6 [Enter]     6        -31.4        -          -
7             7          6        -31.4        -
x^2          49          6        -31.4        -
3             3         49          6        -31.4
1/x           0.3333    49          6        -31.4
y^x           3.6593     6        -31.4      -31.4
-             2.3407   -31.4      -31.4      -31.4
sin           0.0408   -31.4      -31.4      -31.4
8             8          0.0408   -31.4      -31.4
x!        40320          0.0408   -31.4      -31.4
x           1646.7276  -31.4      -31.4      -31.4
+           1615.3276  -31.4      -31.4      -31.4
45 [Enter]   45       1615.3276   -31.4      -31.4
6  [Enter]    6         45       1615.3276   -31.4
7             7          6         45       1615.3276
/             0.8571    45       1615.3276  1615.3276


y^x          26.1240  1615.3276  1615.3276  1615.3276
2 [Enter]     2         26.1240  1615.3276  1615.3276
```

```
3              3          2         26.1240   1615.3276
y^x            8         26.1240  1615.3276  1615.3276
9 [Chs]       -9          8         26.1240   1615.3276
x<>y           8         -9         26.1240   1615.3276
y^x        4.3047e07     26.1240  1615.3276  1615.3276
x          1.1246e09   1615.3276  1615.3276  1615.3276
x^2        1.2646e18   1615.3276  1615.3276  1615.3276
ln          41.6813    1615.3276  1615.3276  1615.3276
+         1657.0089    1615.3276  1615.3276  1615.3276
```

TI-57

These are the key strokes for the *TI-57*:

1 - 2 × 3 y^x 4 ÷ 5 + (6 - 7 x^2 INV y^x 3) 2nd sin * 40320 + (9 y^x (2 y^x 3) × 45 ^ (6 ÷ 7)) +/- x^2 lnx =

We get the same result: **1657.0089**

or this I used the TI-57 Programmable Calculator.
However I had to cheat a little: since the factorial function is missing I just replaced 8!8! with 4032040320.

Also since the y^x operation apparently is not right associative I used another pair of parenthesis to calculate: (2^3)^2

Key Assignments

Of course you are free to choose differently buy I recommend the following key assignments:

Label	Key	Code
+	+	61
-	-	51
*	*	71
/	/	81
↑	Y↑X	-12
=	ENTER↑	41
<	X<>Y	21
>	R↓	22
AOS	CLx/A	-44

Program

This is the program for the *HP-41C*:

```
01▸LBL "AOS57"
02 CLRG
03 3
04 STO 00
05 10
06 STO 01
07 CLST
08 RTN
09▸LBL "+"
10 -1.2
11 GTO 00
12▸LBL "-"
13 -2.2
14 GTO 00
15▸LBL "*"
16 -3.1
17 GTO 00
18▸LBL "/"
19 -4.1
20 GTO 00
21▸LBL "^"
22 5.1
23▸LBL 00
24 STO 02
25 FRC
26 X>0?
27 GTO 11
28▸LBL 09
29 RCL IND 01
30 X=0?
31 GTO 10
32 FRC
33 X<Y?
34 GTO 10
35 X<> Z
36 LASTX
37 XEQ 06
38 R^
39 GTO 09
40▸LBL 10
41 RDN
42▸LBL 11
43 RDN
44 RCL 02
45 XEQ 07
46 ISG 00
47 RTN
48 STO IND 00
49 RTN
50▸LBL "<"
51 0
52▸LBL 07
53 ISG 01
54 RTN
55 STO IND 01
56 RDN
57 RTN
58▸LBL ">"
59 XEQ 08
60 DSE 01
61 RTN
62▸LBL "="
63▸LBL 08
64 RCL IND 01
65 X=0?
66 GTO 13
67 XEQ 06
68 GTO 08
69▸LBL 13
70 RDN
71 RTN
72▸LBL 06
73 DSE 01
74 X<>Y
75 RCL IND 00
76 DSE 00
77 X<>Y
78 XEQ IND Z
79 VIEW X
80 RTN
81▸LBL 01
82 +
83 RTN
84▸LBL 02
85 -
86 RTN
87▸LBL 03
88 *
89 RTN
90▸LBL 04
91 /
92 RTN
93▸LBL 05
94 Y^X
95 END
```

Registers

The program needs 3 register to control the data and the operator stack:

Register	Comment
00	top of data stack
01	top of operator stack
02	current operator

Synthetic Programming

We could use the alpha registers **M**, **N** and **O** instead of register **00**-**02**.
With this the data stack could be started at register **00**.
For now I'm leaving that as an exercise for the dear reader.

Operators

The decimal part of the code is used as precedence.
A negative code means left associativity.

The code for the left parenthesis **(** is **0**.
Thus we already have an implicit open parenthesis.
This makes handling the right parenthesis **)** and **=** similar.

Operator	Label	Code	Precedence	Associativity
(		0		
+	01	-1.2	-0.2	left
-	02	-2.2	-0.2	left
*	03	-3.1	-0.1	left
/	04	-4.1	-0.1	left
^	05	5.1	0.1	right

Code Walkthrough

Initialisation

The registers and the stack is cleared with:

XEQ AOS

Here you can configure the start of the data and the operator stack.
Be warned that there are no checks in the program.
Thus the data stack could grow into the operator stack and vice versa.
It's up to you to select reasonable values.

```
LBL "AOS"
CLRG
3                ; top of data stack
STO 00
10               ; top of operator stack
STO 01
CLST
RTN
```

Enter Operator

Each operator pushes a specific code onto the stack in which label, precedence and associativity is encoded.

```
LBL "+"
-1.2
GTO 00              ; new operator

LBL "-"
-2.2
GTO 00              ; new operator

LBL "*"
-3.1
GTO 00              ; new operator

LBL "/"
-4.1
GTO 00              ; new operator

LBL "^"
5.1

LBL 00              ; new operator
```

New operator

Each time we reach a new operator, we pop operators from the stack until we reach one that has lower precedence.
In the case of a right associative operator, we also stop if we reach an operator of the same precedence.

X	Y	Decision
-0.2	-0.2	pop
-0.1	-0.2	pop
0.1	-0.2	pop
-0.2	-0.1	no more
-0.1	-0.1	pop
0.1	-0.1	pop
-0.2	0.1	no more
-0.1	0.1	no more
0.1	0.1	no more

There's no lower precedence than **-0.2**, thus **+** and **-** always pop.
On the other hand, **^** never pops previous operators.
This leaves us with ***** and **/** which pop unless an operator on the stack has lower precedence like **+** or **-**.

Stack diagram: `( x op -- x' )`

```
LBL 00              ; add new operator
STO 02              ; save new operator
FRC                 ; precedence of new operator
X>0?                ; it is ^
GTO 11              ; no more pop
LBL 09              ; while higher precedence
RCL IND 01          ; top of stack operator
X=0?                ; is left parenthesis ?
GTO 10              ; no more pop
```

```
FRC             ; precedence of top of stack operator
X<Y?            ; has lower precedence ?
GTO 10          ; no more pop
X<> Z           ; x
LASTX           ; top of stack operator
XEQ 06          ; pop operator
R^              ; precedence of new operator
GTO 09          ; while higher precedence
LBL 10          ; no more pop
RDN             ; drop precedence of top of stack operator
LBL 11          ; no more pop
RDN             ; drop precedence of new operator
RCL 02          ; current operator
XEQ 07          ; push operator
ISG 00          ; push data
RTN             ; no op
STO IND 00      ; store data
RTN
```

Push Operator

The left parenthesis **(** is just pushed onto the operator stack.
The **RTN** command after **ISG** is used as a no-operation which is always skipped.

```
LBL "("
0
LBL 07          ; push operator
ISG 01          ; increment top operator
RTN             ; no op
STO IND 01      ; store operator
RDN             ; drop operator
RTN
```

Right Parentheses and Equals

```
while the operator at the top of the operator stack is not a left parenthesis:
pop the operator from the operator stack into the output queue
pop the left parenthesis from the operator stack and discard it
```

```
LBL ")"
XEQ 08
DSE 01         ; pop left parenthesis
RTN

LBL "="
LBL 08         ; while not (
RCL IND 01     ; top of operator stack
X=0?           ; is left parenthesis ?
```

```
GTO 13          ; pop (
XEQ 06          ; pop operator
GTO 08          ; while not (
LBL 13          ; pop (
RDN             ; drop operator
RTN
```

The **=** operator does not pop the implicit left parenthesis.
But otherwise it behaves like the right parenthesis and removes any leftover operators from the operator stack.

Pop Operator

Stack diagram: `( a x op -- a a op x' )`

```
LBL 06          ; pop operator
DSE 01          ; decrement top of operator stack
X<>Y            ; ( a op x )
RCL IND 00      ; y: top of data stack
DSE 00          ; pop data
X<>Y            ; ( a op y x )
XEQ IND Z       ; execute operator
VIEW ST X       ; view result
RTN
```

References

- Shunting-yard algorithm
- Dijkstra's original description of the Shunting yard algorithm

Also note the “AOSKY6”and “AOSKY5” utilities included in the module for a convenient bulk user key assignment for the last two AOS programs.

```
01 *LBL "AOSKY5"
02 "+"
03 61
04 PASN
05 "-"
06 51
07 PASN
08 "*"
09 71
10 PASN
11 "/"
12 81
13 PASN
14 "="
15 41
16 PASN
17 "^"
18 -12
19 PASN
20 "("
21 24
22 PASN
23 ")"
24 25
25 PASN
26 END
```

Formula Evaluation ; by Stefan Fegert

From "HP-41 in der Praxis"

A typical problem for the author's field of study (computer science) is the evaluation of an expression given in algebraic form

One can solve this problem in Pascal and with recursive functions, but also with this program "INF16".

The expression may contain the following symbols:

- The digits from 0 to 9

- The operation signs +, - *

- Parentheses

Whereby the numbers may only be single digits.

In addition, the formula may need to be broken as it cannot be longer than 24 characters.

The principle of the program is based on calculating a partial expression, which is enclosed in parentheses, and to replace it with a special character whose ASCII value corresponds to the register in which the value of the parenthesis is located.

If multiplications occur in the parentheses, then they are also replaced by special characters and treated in the same way as the parentheses. Only when all parentheses and multiplications have been replaced, addition and subtraction are performed from left to right.

With this the parenthesis is calculated, and the program looks for another 'close parenthesis'. If no more are found, the expression is finished.

Instructions.

1. Load the program and start it
2. Enter formula, where the characters "less than" and "greater than" represent the brackets (< , >)
3. After the result is given, press R/S for a new start.

"INF16" 1 317 bytes I 46 REG I Size 44 I Peripherals: none

Example:

(2*3+2-(2*3)+8)*2 = 20

<u>Data Registers</u>

00 -	Counter for special characters = brackets.
01 – 04	ASCII codes = values in brackets
05 -	Counter for special characters = products
06 - 13	ASCII codes = values of products
14 -	Pointer in ALPHA-REG for bracketing
15 -	Counter for read characters
16 – 40	ASCII codes of the characters read in up to 40
41 -	Pointer at multiplication
42 -	Sum at evaluation in LBL 25
43 -	Intermediate memory in LBL 50

Program listing.

```
01  *LBL "FORMULA"
02  CF 05
03  *LBL 00
04  SF 25
05  "FRMLA"
06  PURFL
07  CF 25
08  6
09  CRFLAS
10  CLRG
11  "` ?"
12  AON
13  STOP
14  AOFF
15  APPREC
16   E
17  STO 00
18  *LBL 02
19  CLX
20  SEEKPT
21  GETREC
22  16
23  STO 15
24  62
25  POSA
26  X<0?
27  GTO 10
28  STO 14
29  *LBL 03
30  RCL 14
31  AROT
32  ATOX
33  X<>Y
34  CHS
35  AROT
36   E
37  ST- 14
38  RDN
39  X<>Y
40  60
41  X=Y?
42  GTO 05
43  RDN
44  STO IND 15
45   E
46  ST+ 15
47  GTO 03
48  *LBL 05
49  RCL 14
50   E
51  +
52  AROT
53  RCL 00
54  XTOA
55  RDN
56   E
57  +
58  CHS
59  AROT
60  CLX
61  SEEKPT
62  DELREC
63  APPREC
64  RCL 15
65   E
66  -
67  0,016
```

```
68  +
69  CLA
70  STO 15
71  *LBL 06
72  RCL IND 15
73  XTOA
74  DSE 15
75  GTO 06
76  XEQ 20
77  STO IND 00
78   E
79  ST+ 00
80  GTO 02
81  *LBL 10
82  XEQ 20
83  FIX 0
84  >"* = "
85  ARCL X
86  >" **"
87  AVIEW
88  FIX 4
89  STOP
90  GTO 00
91  *LBL 20
92  6
93  STO 05
94  *LBL 21
95  42
96  POSA
97  X<0?
98  GTO 25
99   E
100  -
101  STO 41
102  AROT
103  ATOX
104  ATOX
105  ATOX
106  XEQ 42
107  STO IND 05
108  RCL 05
109  XTOA
110  RCL 41
111  CHS
112   E
113  -
114  AROT
115   E
116  ST+ 05
117  GTO 21
118  *LBL 25
119  ATOX
120  X=0?
121  GTO 30
122  XEQ 51
123  STO 42
124  *LBL 26
125  42
126  ATOX
127  X=0?
128  GTO 30
129  ATOX
130  XEQ IND Y
131  STO 42
132  GTO 26
133  *LBL 30
134  RCL 42
135  RTN
136  *LBL 42
137  XEQ 50
138  *
139  RTN
140  *LBL 43
141  XEQ 50
142  +
143  RTN
144  *LBL 45
145  XEQ 50
146  -
147  RTN
148  *LBL 50
149  STO 43
150  RCL Z
151  XEQ 51
152  X<> 43
153  XEQ 51
154  RCL 43
155  X<>Y
156  RTN
157  *LBL 51
158  48
159  X<=Y?
160  SF 05
161  FS? 05
162  -
163  FC?C 05
164  RCL IND Y
165  END
```

Alpha Function Interpreter ; by Erik Christensen

PPCCJ V10N1 p33

This. program interprets a RPN representation in ALPHA. It decodes the function one character at a time. It is useful to have a lot of functions that can be saved in X-Memory sometimes. A function can be up to 24 chr. long. The operations are limited to +, -, *, /, Y^X, and % but can be expanded to meet your needs. The variables are restricted to A-J which are actually registers 1-10. The symbols that can be used are as follows:

Character	Description	RPN	ALPHA
A-J	Variables	RCL 01, RCL 02…	AB..J
0-9	Numbers	9 ENTER 5	95
%	Percent	4 ENTER 5 %	95%
^	Powers	3 ENTER 9 Y^X	39^
+	Addition	3 ENTER 8 +	38+
-	Subtraction	8 enter 3 -	83-
*	Multiplication	9 ENTER 6 *	96*/
/	Division	9 ENTER 5 /	95/
,	Pause, show X	PSE	,
?	Stop for input	PROMPT	?

Examples of formulas.

A=BH ; area=base x height would be "BH*"

E=1/2MV ; energy=1/2 m.v^2 would be "12/A*B2"

Instructions.

Step 1.- At the prompt "NAME?" enter the function name into ALPHA. It can be up to 7 chr long. If such a name is in X-Memory, then the program pulls the function out and runs it. (go to step 3) If this name is a new one then go to step 2 to create the formula. If you just press R/S then the function name stays the same as last time.

Step 2.- At the prompt "FORMULA?" enter the chr sequence that represents the formula. For example AB+ would be A+B

Step 3.- See viewing of "VAR? A-J" and then function. Set values of variables A-J by doing value (STO) [A] through [J]. When ready to run the function (R/S)

Step 4.- Function will be shown being "eaten" in the display. If the program stops with some of the function still in ALPHA, then key in an input for "?" and [R/S]. After that. Go to step 1

The whole stack can be used by the formula. When you make a new function it is automatically saved in X-Mem. For future use. The program itself uses register 00, and the variables are registers 01 – 10, leaving a SIZE 011. Variables A-J are key-mapped as Reg.

01 to 10. They need not initialization every time the function is executed. The program is 26 regs (179 bytes).

Additions can be made in the following manner:

say you want the letter “M” to stand for MOD. Type: { ALPHA, CLA, M, ALPHA, ATOX, GTO.196. PRGM, LBL 77 (the number in X), MOD, PRGM}.

Now if you made the function “ABMC+” it would be the same as the RPN sequence, RCL 01, RCL 02, MOD, RCL 03, +

Program listing:

```
ALPHA FUNCTION INTERPRETER By Erik Christensen (10041)

01 LBL "AFI"
02 LBL 04
03 "NAME?"
04 XEQ 02
05 4
06 SF 25
07 RCLPTA
08 FS? 25
09 GTO 01
10 CRFLAS
11 "FORMULA?"
12 XEQ 02
13 APPREC
14 LBL 01
15 CLST
16 SEEKPT
17 "VAL? A-J"
18 AVIEW
19 GETREC
20 PROMPT
21 CLST
22 LBL 03
23 AVIEW
24 STO 00
25 RDN
26 ATOX
27 X() 00
28 XEQ IND 00
29 GTO 03
30 LBL 65
31 LBL 66
32 LBL 67
33 LBL 68
34 LBL 69
35 LBL 70
36 LBL 71
37 LBL 72
38 LBL 73
39 LBL 74
40 LBL 75
41 X() 00
42 64
43 -
44 X() 00
45 RCL IND 00
46 GTO 03
47 LBL 42
48 *
49 RTN
50 LBL 43
51 +
52 RTN
53 LBL 45
54 -
55 RTN
56 LBL 47
57 /
58 RTN
59 LBL 94
60 Y X
61 RTN
62 LBL 37
63 %
64 RTN
65 LBL 02
66 AON
67 AVIEW
68 CLA
69 STOP
70 AOFF
71 RTN
72 LBL 44
73 VIEW X
74 PSE
75 RTN
76 LBL 63
77 PROMPT
78 RTN
79 LBL 48
80 LBL 49
81 LBL 50
82 LBL 51
83 LBL 52
84 LBL 53
85 LBL 54
86 LBL 55
87 LBL 56
88 LBL 57
89 RCL 00
90 48
91 -
92 RTN
93 LBL 00
94 CLD
95 STOP
96 GTO 04
97 END

REG 26
SIZE 011
X-FUNCTIONS
BYTES 179
```

Improved Alpha Function Interpreter

By Erik Christensen, PPCCJ V10N5 p10

The alpha function interpreter from V10 N1 P33a has been improved to accommodate Algebraic functions rather that RPN representations. This will allow more direct entry of formulas, without having to worry about stack gymnastics and order of execution.

The old !unctions +, - , *, *I*.^. and % are included. Varlables are limited to A-J as before, and the numbers used as constants can range from 0-9 and .1 to .9. One level of parenthesis can be accessed, using “<” and “>” as the open and closed parenthesis.

The program woks exactly as before as seen by the user, but the program has been totally rewritten using a different interpreting scheme. The only difference is the structure of the formulas that you enter. Some examples of formulas and their equivalent representations to be types into the ALPHA register:

Actual Formula	ALPHA Representation
A/B	A/B
A*B	A*B
A+B	A+B
A-B	A-B
A+1	a+1
A+0.10	A+.1
1% of A	1%A
A^B	A^B
(A+B)/2	A+B/2
2/(A+B)	2/<A+B>
1/2+A+B^2	1/2*A*<B^2>
SQRT(A^2+B^2	<A^2>*<B^2>^.5
(A+B)/(C+0.9)	A+B/<C+.9)
(A+1)^2+0.3	A+1^2+.3

The functions are interpreted from left to right, one or two characters at a time.. Up to a 24-character function can be saved in memory, like the previous program. The HP-41 stack is transparent to the user, so it need not be worried about. The program is 41 bytes longer than the last one, and needs SIZE 013 because registers 00, 11, and 12 are used by the interpreter The total byte count is 238, and 34 registers are used for program space/ Happy Formulating!

Any questions, comments suggestions, send a letter to the address below, or call 1-206-852-6719 after 3PM weekdays.

Program listing:

PPC

A F I   Improved   By, Erik Christensen (10041)

```
01 LBL "AFI"
02 LBL 09
03 "NAME?"
04 XEQ 11
05 SF 25
06 RCLPTA
07 FC?C 25
08 GTO 03
09 SF 25
10 POSFL
11 FC?C 25
12 GTO 09
13 GTO 12
14 LBL 03
15 4
16 CRFLAS
17 "FORMULA?"
18 XEQ 11
19 APPREC
20 LBL 12
21 CLST
22 SEEKPT
23 "VAL? A-J"
24 AVIEW
25 SF 25
26 GETREC
27 FC?C 25
28 GTO 09
29 PROMPT
30 CLST
31 STO 00
32 60
33 POSA
34 X≠0?
35 GTO 08
36 "⊢+"
37 -1
38 AROT
39 GTO 01
40 LBL 08
41 ATOX
42 XEQ 06 •
43 STO 00
44 LBL 01
45 AVIEW
46 ATOX
47 X=0?
48 GTO 02
49 RCL 00
50 ATOX
51 XEQ 06
52 XEQ IND Z
53 STO 00
54 GTO 01
55 LBL 02
56 RCL 00
57 CLD
58 STOP
59 GTO 09
60 LBL 06
61 64
62 X>Y?
63 GTO 07
64 -
65 RDN
66 RCL IND T
67 RTN
68 LBL 07
69 RDN
70 60
71 X=Y?
72 GTO 10
73 RDN
74 48
75 -
76 X=0?
77 RTN
78 X>0?
79 RTN
80 RDN
81 ATOX
82 48
83 -
84 10
85 /
86 RTN
87 LBL 10
88 RUP
89 STO 12
90 ATOX
91 XEQ 06
92 STO 11
93 LBL 05
94 AVIEW
95 62
96 POSA
97 X=0?
98 GTO 04
99 RDN
100 ATOX
101 RCL 11
102 ATOX
103 XEQ 06
104 XEQ IND Z
105 STO 11
106 GTO 05
107 LBL 04
108 ATOX
109 RCL 12
110 RCL 11
111 RCL 00
112 RTN
113 LBL 45
114 -
115 RTN
116 LBL 43
117 +
118 RTN
119 LBL 42
120 *
212 RTN
122 LBL 47
123 /
124 RTN
125 LBL 94
126 Y/X
127 RTN
128 LBL 37
129 %
130 RTN
131 LBL 11
132 AON
133 AVIEW
134 CLA
135 STOP
136 AOFF
137 RTN
X-FUNCTIONS USED
SIZE 013, REG 34
```

Alpha Stack on the HP-41C ; by Godwin Stewart

https://www.hpmuseum.org/forum/thread-12050.html?highlight=alpha+stack

while back, another member of the MoHPC forum mentioned a program he'd written for the HP-41C that allows the user to manage multiple alpha registers. I responded saying that I had written something similar many moons ago that behaved like a LIFO (last-in-first-out) stack rather than an indexed array of datasets and said that I'd look it up.

I have no idea whatsoever what I did with my little utility so I decided to rewrite it, purely and simply. So here it is.

NB: This program creates and manages a data file in Extended Memory called "ASTACK". If you already have a file of that name, it will be deleted! Note also that running these programs uses Flag 01 and trashes registers R07-R10. Finally, since this uses Extended Memory, you will need to run this on a 41CX or SwissMicros DM41, or on a 41C or 41CV with the "X-Function" module.

The size of the data file created in Extended Memory depends on the depth of the alpha stack that you want to create. Two registers are needed for a header in the file and four registers per stack level are needed. So, if you want a stack that's 6 levels deep, for example, then you'll need room in your Extended Memory for a file that's 2+6*4= 26 registers in size. You'll actually need 28 registers free because the calculator also steals 2 registers for its own internal housekeeping when you create a file in X-Mem.

The three utilities provided are "ASINIT", "APUSH" and "APOP".

ASINIT
Run this with the depth of the desired stack in X, The '41's own error detection will prevent you from creating a file that's too big or from running this on a machine with no "X-Function" module installed (remember, the 41CX and the DM41 have this module baked into their ROM).

APUSH
This will save the current contents of your alpha register onto the alpha stack and return 0 in X, unless the stack is already full, in which case you'll get -3 back instead. If the alpha stack hasn't been initialized (by running ASINIT) then you'll get -1 back in X.

APOP
This takes the string on the top of the alpha stack and transfers it into the '41's alpha register, removing it from the stack. If all went well, X will contain 0 after returning from this program. If the alpha stack has not yet been initialized then you'll get -1 back, or if the stack was already empty (everything already popped off it) when you called APOP then you'll get -2 back.

You can go and grab these utilities here: alpha-stack.zip

Software provided, as usual, as a text listing, a .raw file and a PDF with bar codes.

Program listing:

```
LBL "ASINIT"
XEQ 10
SF 25
PURFL
CF 25
ABS
INT
STO Y
4
*
2
+
CRFLD
RDN
SAVEX
CLX
SAVEX
RTN
LBL 10
FS?C 01
GTO 11
ASTO 07
ASHF
ASTO 08
ASHF
ASTO 09
ASHF
ASTO 10
LBL 11
"ASTACK"
RTN
LBL 20
XEQ 10
CLX
SF 25
SEEKPTA
FS?C 25
RTN
-1
RTN
LBL "APOP"
SF 01
XEQ 20
X#0?
RTN
GETX
GETX
X#0?
GTO 01
-2
RTN
LBL 01
4
*
2
-
SEEKPT
CLA
7.01
GETRX
LBL 03
ARCL 07
ARCL 08
ARCL 09
ARCL 10
FS?C 01
GTO 04
1
SEEKPT
GETX
1
SEEKPT
-
SAVEX
LBL 04
CLST
RTN
LBL "APUSH"
CF 01
XEQ 20
X#0?
RTN
GETX
GETX
X=Y?
GTO 02
1
SEEKPT
+
SAVEX
4
*
2
-
SEEKPT
7.01
SAVERX
CLA
SF 01
GTO 03
LBL 02
-3
END
```

Alpha stack

Program Registers Needed: 28

Row 1 (1 - 2)

Row 2 (3 - 11)

Row 3 (12 - 20)

Row 4 (21 - 28)

Row 5 (29 - 33)

Row 6 (33 - 41)

Row 7 (41 - 44)

Row 8 (45 - 53)

Row 9 (54 - 61)

Row 10 (62 - 68)

Row 11 (69 - 77)

Row 12 (78 - 80)

Row 13 (80 - 88)

Row 14 (89 - 96)

Row 15 (97 - 104)

Row 16 (104)

X-Registers ALPHA ; by Tyann

https://www.hpmuseum.org/forum/thread-7384.html?highlight=SIZA

Here is a program that simulates several Alpha registers in a file,

SIZA creates the file of X (integer) registers and returns in X the number of registers xmemory consumed,

ASTOA copies the register Alpha in register No. X,

ARCLA copies the register X in the Alpha register,

A <> exchanges register Alpha with register n ° X

A =? Test the equality between the register Alpha and the register n° X, returns 1 if =, 0 otherwise in X.

CLRA deletes the registers and destroys the file.

Registers start at 1, register 0 is used for exchanges and tests but can be used if needed. The number of the Alpha register to be used must be set to X (integer).

The file named XALPHA is the current file and must remain so for instructions to work.

ASTOA, ARCLA and A <> preserve the stack.
A =? Preserves Alpha.

Program listing.

```
LBL "SIZA"
"XALPHA"
1
+
STO Y
3,7
*
INT
1
+
CRFLAS
" "
LBL 00
APPREC
DSE Y
GTO 00
CLA
RTN
LBL "ASTOA"
SEEKPT
DELREC
INSREC
RTN
LBL "ARCLA"
SEEKPT
GETREC
RTN
LBL "A<>"
X<>L
CLX
SEEKPT
DELREC
INSREC
X<>L
SEEKPT
GETREC
DELREC
X<>L
SEEKPT
INSREC
X<>L
SIGN
SEEKPT
GETREC
DELREC
X<>L
SF 25
SEEKPT
FC?C 25
APPREC
INSREC
X<>L
CLX
SEEKPT
GETREC
X<>L
RTN
LBL "A=?"
STO L
SEEKPT
POSFL
X≠Y?
GTO 02
ALENG
X<>Y
CLX
XEQ "ASTOA"
X<>L
XEQ "ARCLA"
X<>Y
ALENG
X<>L
XEQ "ARCLA"
X<>L
X≠Y?
GTO 01
-
CLX
1
RTN
LBL 01
RDN
LBL 02
CLX
RTN
LBL "CLRA"
"XALPHA"
PURFL
CLA
END
```

Managing X-Mem Registers ; by Peter Reiter

From "HP-41 Hilfen und Anwendungen"

As you will surely know, the HP-41CX has max. 319 data memory.

Since this can be too little in some cases, especially if you have stored long programs, I have put down on paper a program with which you can store data in a file, whereby also a register arithmetic is possible. If you now have 2 memory expansion modules in the computer, you have 601 data registers available.

Program Instructions.

Before you can use the save commands, if the file you want to use already exists you must save its name in the ALPHA register and prepare the program with XEQ "FLREG". Thus the file for the data storage is the current file. If there is no file for the "register arithmetic in the extended memory:", then enter the file name in the ALPHA register and the number of required registers in the X-stack. Also start the software with XEQ "FLREG". Now the file for the register arithmetic is present and besides this file is the current file.

The individual commands can be used by means of an XEQ instruction or key assignment or via the label "FLAR". The label "FLAR" makes it possible to execute different file register operations in a frequently repeating program loop, where the operation to be executed is announced in the ALPHA register.

FLREG

This part of the program is used to create a file or, if it is already present, to declare it to be the current file. Create file (no file exists yet): File name = ALPHA register

In each case program start with XEQ "FLREG

FLAR

The "FLAR" command allows to work indirectly with del file operations of the program. Save the command to be processed in the ALPHA register and, after entering the data according to the operation, start XEQ "FLAR".

Example: to add 50.4 to the contents of the 10th. register:

"FL+", 10 , ENTER^, 50.4, XEQ "FLAR"

FLSTO

Save data to file (like the STO command).

Before program start:

Y-stack = File register number
X stack = Number

After processing:

Y-stack = File register number
X stack = Number

FLRCL

Recall data from the file (like the RCL command).

Before program start:

Y-Stack = n/a
X stack = Number

After processing:

Y-stack = File register number
X stack = Number

FL Arithmetic. (FL+, FL-, FL*, and FL/)

These operations are like the STO arithmetic, in that the calculation is made with the contents of the file register and the value in the X-Register.

Before program start:

Y-stack = File register number
X stack = Number

After processing:

Y-stack = File register number
X stack = Number

FL<>RG (not supported by FLAR !)

This program section exchanges the contents of a data registers within the contents of a file register.

Before program start:

Y-stack = Data register number
X stack = File Register number

After processing:

Y-stack = File register number
X stack = Number

On completion, the contents of the data and file registers are left in the T and Z registers n respectively.

FLIND (Not supported by FLAR !)

In the Y-stack, specify the file register that contains the indirect parameter for the instruction specified in the ALPHA register and the number stored in the X-stack.

ALPHA: function to perform
Y: IND Register number
X: number to add

The result will be left in the X-register on completion – in addition to the register IND Y

For example, if the File register #6 contains the value "4", the following sequence will add 5 to the File register #4:

ALPHA: "ST+", ALPHA, 6, ENTER^, 5, XEQ "FLIND"

The result will be left in the X-register on completion – in addition to register #4 in this case) – and therefore the stack will be lifted.

FL<>FL (not supported by FLAR)

Exchange between two file registers, specified by their numbers in X- and Y- stack registers

The previous registers values are left in the Z- and T- registers.

FLCLX

Clearing of a set of file registers whose FROM-TO range numbers are specified in the Y- and X-registers.

Program listing.

1	**LBL "FLREG"**
2	SF 25
3	R^
4	CLX
5	SEEKPTA
6	FC?C 26
7	GTO 15
8	RTN
9	LBL 15
10	RDN
11	CRFLD
12	RTN
13	LBL A
13	**LBL "FLAR"**
14	SF 00
15	R^
16	ASTO X
17	GTO IND X
18	**LBL "FLSTO"**
19	XEQ 22
20	X<>Y
21	SEEKPT
22	X<>Y
23	SAVEX
24	RTN
25	**LBL "FLRCL"**
26	XEQ 22
27	SEEKPT
28	GETX
29	RTN
30	**LBL "FL+"**
31	LBL 16
32	XEQ 22
33	X<>Y
34	SEEKPT
35	X<>Y
36	GETX
37	FS?C 01
38	GTO 18
39	FS?C 02
40	GTO 19
41	FS?C 03
42	GTO 20
43	+
44	LBL 21
45	X<>Y
46	SEEKPT
47	X<>Y
48	SAVEX
49	RTN
50	**LBL "FL-"**
51	SF 01

```
52    GTO 16
53    LBL "FL*"
54    SF 02
55    GTO 16
56    LBL "FL/"
57    SF 03
58    GTO 16
59    LBL 18
60    X<>Y
61    -
62    GTO 21
63    LBL 19
64    *
65    GTO 21
66    LBL 20
67    X<>Y
68    /
69    GTO 21
70    LBL 22
71    FC?C 00
72    RTN
73    CLX
74    RDN
75    RTN
76    LBL "FL<>RG"
77    XEQ 22
78    SEEKPT
79    GETX
80    X<> IND Z
81    X<>Y
82    SEEKPT
83    X<>Y
84    SAVEX
85    RCL IND Y
86    RDN
87    RDN
88    RTN
89    LBL "FLIND"
90    X<>Y
91    SEEKPT
92    X<>Y
93    GETX
94    X<>Y
95    GTO A
96    LBL "FL<>FL"
97    XEQ 22
98    SEEKPT
99    GETX
100   X<> Z
101   SEEKPT
102   GETX
103   X<> Z
104   SEEKPT
105   X<> Z
106   SAVEX
107   X<>Y
108   SEEKPT
109   X<> T
110   SAVEX
111   RDN
112   RDN
113   RTN
114   LBL "FLCLX"
115   XEQ 22
116   LBL 00
117   SEEKPT
118   ENTER^
119   CLX
120   XEQ "FLSTO"
121   RDN
122   E
123   +
124   X<=Y?
125   GTO 00
126   CLST
127   END
```

Here's an alternative set of routines that accomplish the same function, perhaps in a more straight-forward way. They are unfortunately not included in the module because the number of available FAT entries in the ROM was already depleted…

The DATA file name is assumed to be in ALPHA (or names for FSWAP):

```
LBL "FRIND"  IND rg# in X
SEEKPTA
CLX
GETX
LBL "FRCL"   rg# in X
SEEKPTA
GETX
RTN
LBL "FSIND"  rg# in Y, value in X
X<>Y
SEEKPTA
CLX
GETX
X<>Y
LBL "FSTO"   rg# in Y, value in X
X<>Y
SEEKPTA
X<>Y
SAVEX
RTN
LBL "FXIND"  rg# in Y, value in X
X<>Y
SEEKPTA
CLX
GETX
X<>Y
```

```
LBL "FX<>"   rg# in Y, value in X
X<>Y
SEEKPTA
X<>Y
GETX
X<>Y
SAVEX
RDN
RTN
LBL "FSWAP"          rgs# in Y,X
SEEKPTA              FL1,FL2 in ALPHA
GETX
ASWAP
RCL Z
SEEKPTA
GETX
RDN
RDN
SAVEX
ASWAP
RDN
RDN
SAVEX
X<>Y
RDN
RCL Z
END
```

Appendix.- *Valentín Albillo's STKN FOCAL Program*

Here's a verbatim copy of Valentín's article contributed to the Melbourne PPC Chapter. See this reference for all the details.

Program characteristics. –

This program simulates a N-level RPN stack, that is a stack with n registers (not just the 4 registers of the standard, built-in, 4-level stack). The value n is chosen by the user, and is limited only by available memory. Several functions are provided, ENTER, X<>)Y ,RDN, CLST, +, -, *, / , Y^X, LASTX, PI, and RCL. The rest of the functions are the built-in functions, for instance, GTO is the built-in GTO, SQRT, SIN, etc.

The program is 159 lines, 343 bytes. It requires SIZE n+12 for a n-level stack. All operations are very fast, even for large n, so the program may be used as easi4r as if it were the standard 4-level stack. All functions are supposed to be assigned to keys for its execution in USER mode.

ET (Enter) is assigned to 41 (ENTER), RD (Roll Down) to 22 (RDN), +N (addition) to 61 (+), -N (subtraction) to 51 (-), *N (multiplication) to 71 (*), /N (division) to 81 (/), PI to -82 (PI), CLN (Clear Stack) to -21 (CLΣ), RCLN (Recall) to 34 (RCL), XY (exchange to 21 (X<>Y), and ^N (power) to -12 (Y^X).

The stack behaves exactly like the original one. it lifts and performs the same, register duplication, etc, but for a minor detail: RCL after ENTER does not overwrite the number in **X** but the stack is lifted. This has been done intentionally but can be changed to the overwrite mode easily. Except for this sequence, all other functions perform as you would expect, the upper register replicates each time the stack drops because of a two-umber operation, etc.

RCLN, when executed, prompts for an argument with the standard RCL _ _ , and the program stays in a PSE loop, waiting for you to enter-the argument for the desired register. This can be 00 thru 10 (both included) and from n+12 upwards, where n is the number of levels of your stack. So, when using STO, remember that you have registers 00 thru 10 and n+12 upwards for your use. R11, R12 are used as scratch, and R13 thru R(n+11) are used to store part of the stack.

Instructions.

- Make all the necessary assignments, set USER mode
- Use the stack as normal, first, XEQ "STKN" => N=?
- Enter the desired number of levels, n R/S =>READY
- From now on, think of the 41C as a n-level stack machine, and execute desired functions accordingly. Take into account that STO should be used only with addresses 00 thru 10 and n+12 up, and the same is true for RCL. The argument for RCL is entered during a pause. RCL after ENTER does not overwrite X but lifts the stack first.

So, you. see, it is as easy to use as if it were the normal stack. Now let's compute an example taken from TI adds…

Compute 1 + 2 * 2.5^(3/7) = ?
if' we want to key in the problem left-to-right, we need a 5-level stack (minimum),

```
XEQ "STKN"      => N=?;  ,
5  R/S   => READY'
ENTER  2  ENTER  2.5  ENTER  3  ENTER  7 ;
[/N] => 0.43 ;  [YX] => 1.48 ;   [*N] => 2.96  ;   [+N] => 3.96 ,
FIX 9           => 3.961936296
```

so, the problem was keyed in left-to-right. This is a very good advantage of a n-level stack, you can hold up to n-1 pending operations. Using the standard 4-level stack, up to 3 operations may be left pending, and problems requiring more pending operations cannot be keyed left-to-right and have to be rearranged. But, using a, say, 15-1evel stack, you can hold as many as 14 pending operations, and thus, you can confidently key in any - problem left to right, without rearranging anything. That's the usefulness of the program. You can also use it when leaving someone your 41c, and that person is not very used to RPN, show him how to use ENTER ,RIN ,and X<>Y, and let the 15 (say) level stack do the rest !

RPN STACK OF N LEVELS (by Valentin Albillo) (4747)

```
01 LBL"STKN"
02 "N=?"
03 PROMPT
04 11
05 +
06 1 E3
07 /
08 13
09 +
10 STO 11
11 13.012
12 STO 12
13 XEQ"CLN"
14 "READY"
15 PROMPT
16 LBL"XY"
17 FS?C 04
18 CF 22
19 FS?C 22
20 GTO 10
21 X() IND 11
22 RTN
23 LBL 10
24 XEQ 06
25 X()Y
26 RTN
27 LBL"RD"
28 XEQ"XY"
29 DSE 12
30 DSE 11
31 GTO 01
32 RTN
33 LBL 01
34 LASTX
35 X() 11
36 FRC
37 STO 12
38 1 E3
39 *
40 X() 12
41 RCL 12
42 +
43 X() 11
44 STO L
45 RDN
46 .012
47 ST+ 12
48 RDN
49 RTN
50 LBL 07
51 FS?C 04
52 CF 22
53 FC?C 22
54 GTO 06
55 X()Y
56 XEQ 06
57 X()Y
58 LBL 06
59 ISG 11
60 ISG 12
61 GTO 02
62 STO IND 11
63 RTN
64 LBL 02
65 13.012
66 STO 12
67 RDN
68 LASTX
69 X() 11
70 FRC
71 13
72 +
73 X() 11
74 STO L
75 RDN
76 STO IND 11
77 RTN
78 LBL"ET"
79 XEQ 07
80 SF 04
81 RTN
82 LBL 03
83 FS?C 04
84 CF 22
85 FS?C 22
86 RTN
87 ISG 11
88 GTO 10
89 RCL 11
90 FRC
91 13
92 +
93 STO 11
94 RDN
95 LBL 10
96 RCL IND 11
97 X() IND 12
98 RCL 11
99 FRC
100 RCL 12
101 INT
102 +
103 STO 11
104 RDN
105 X()Y
106 DSE 12
107 DSE 11
108 GTO 01
109 RTN
110 LBL"+N"
111 XEQ 03
112 +
113 RTN
114 LBL"-N"
115 XEQ 03
116 -
117 RTN
118 LBL"*N"
119 XEQ 03
120 *
121 RTN
122 LBL"/N"
123 XEQ 03
124 /
125 RTN
126 LBL"^N"
127 XEQ 03
128 Y^X
129 RTN
130 LBL"LX"
131 XEQ 07
132 LASTX
133 RTN
134 LBL"PI"
135 XEQ 07
136 PI
137 RTN
138 LBL"CLN"
139 XEQ 01
140 CLST
141 CF 04
142 CF 22
143 LBL 05
144 STO IND 11
145 DSE 12
146 DSE 11
147 ISG 12
148 GTO 05
149 RTN
150 LBL"RCLN"
151 XEQ 07
152 "RCL __ "
153 AVIEW
154 LBL 04
155 PSE
156 FC?C 22
157 GTO 04
158 RCL IND X
159 END
```