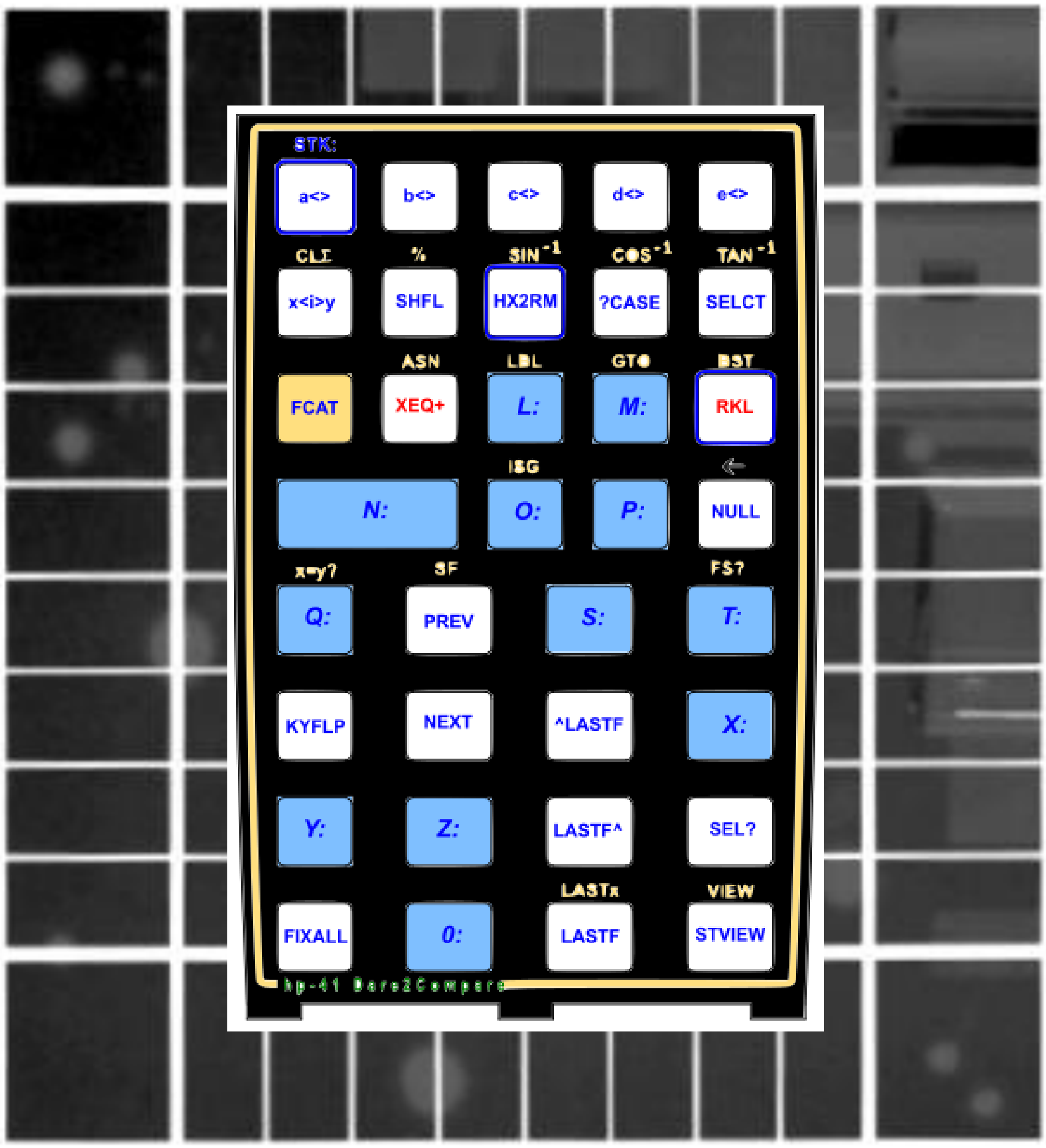
HP-41 WARP_CORE+

Revision K15

With RCL Math, SHUFFLE, and Full Stack Tests

Including Auto-Complete Advanced XEQ+ Mode

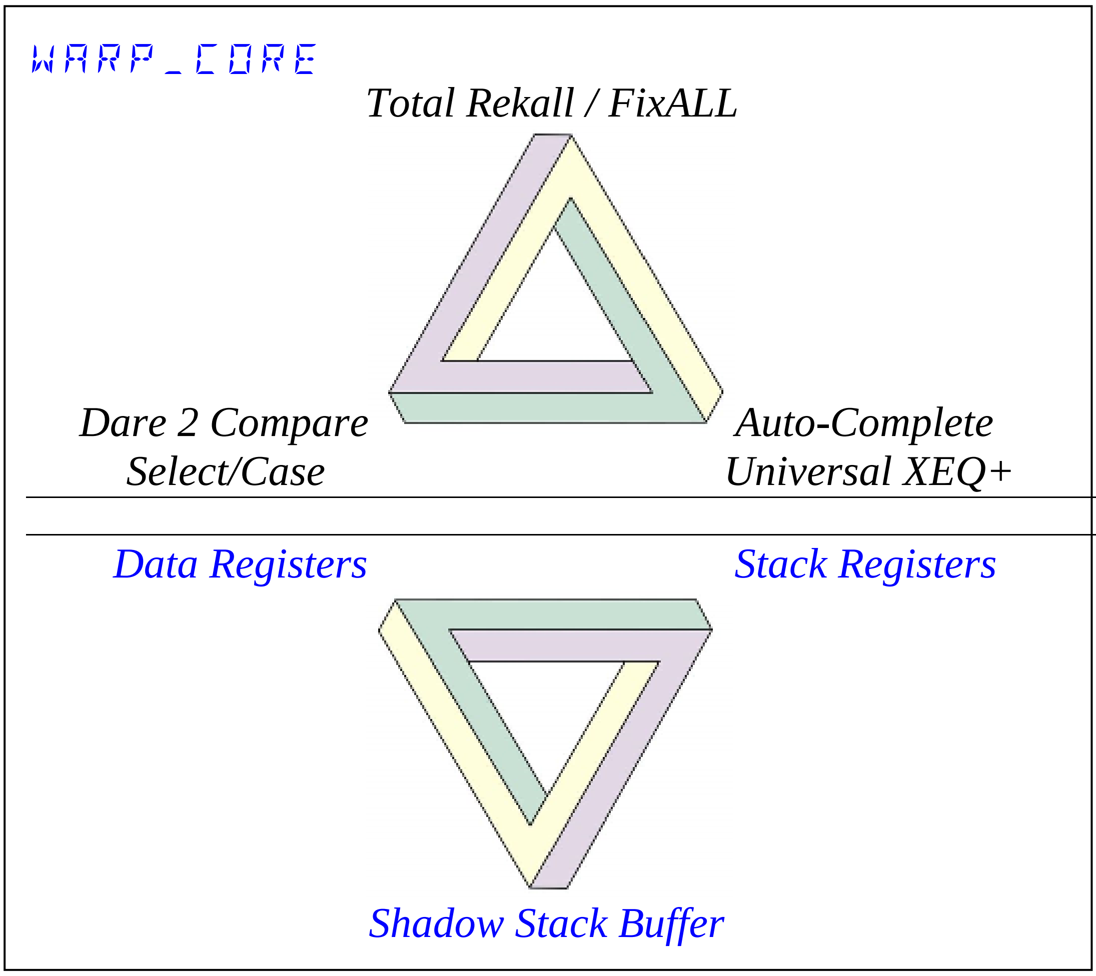
& Fix ALL mode for accurate number display.



Written and programmed by Ángel Martin

January 31, 2023

This compilation revision 4.3.37
Copyright © 2014 -2023 Ángel Martin



Screen captures taken from V41, Windows-based emulator developed by Warren Furlow.
See www.hp41.org

Acknowledgments. - Thanks to the MCODE pioneers and grand masters who published their work in PPC Journal and other sources, such as Ken Emery (and alter-ego Skiwd), Clifford Stern, Doug Wilder, Håkan Thörngren, Frits Ferwerda and Nelson F. Crowle amongst others for their powerful functions, real examples of solid MCODE programming.

Many thanks to Greg J. McClure and Poul Kaarup for their contributed functions in the auxiliary FAT. Everlasting thanks to the original developers of the HEPAX and CCD Modules – real landmark and seminal references for the serious MCODER and the 41 system overall. With their products they pushed the design limits beyond the conventionally accepted, making many other contributions pale by comparison.

WARP_CORE 2023+
HP-41 Module

Table of Contents

Figure 0: Interaction between the different function launchers.

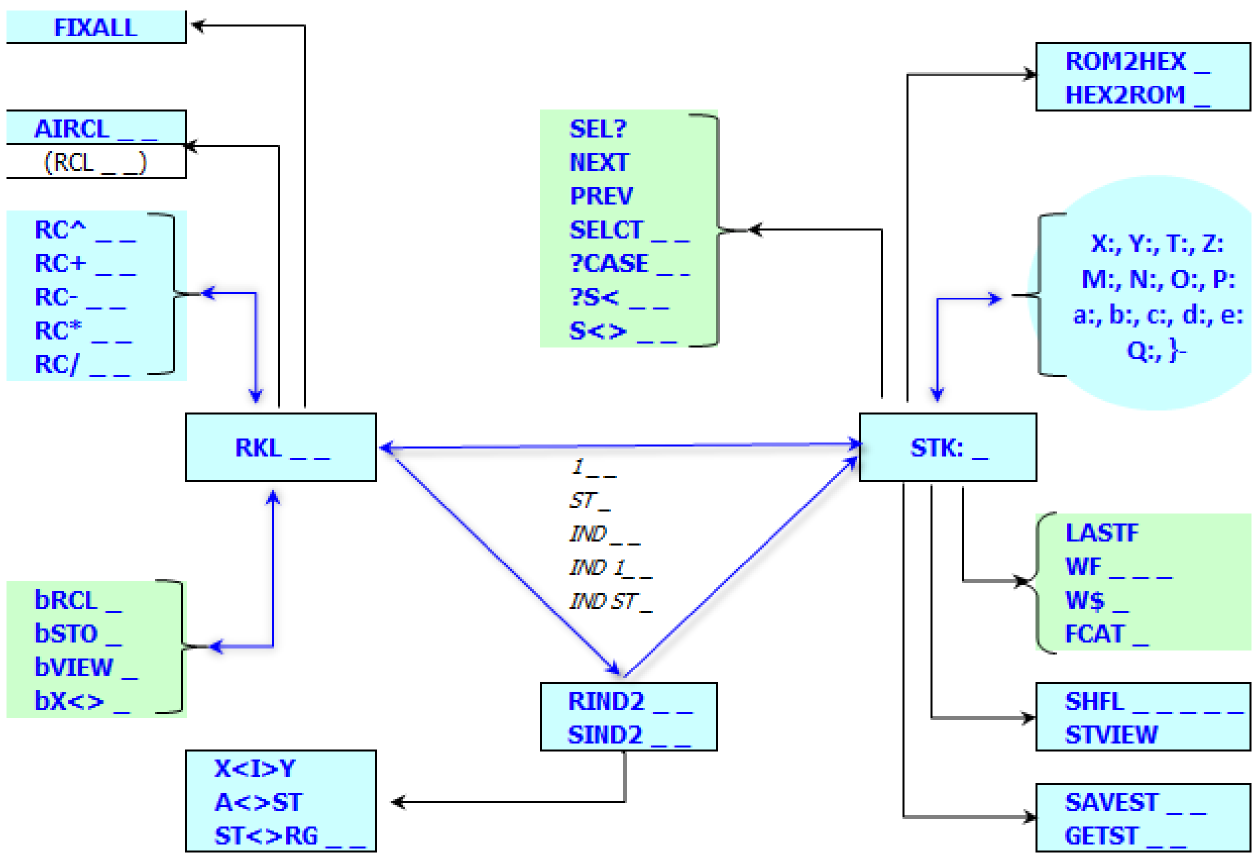
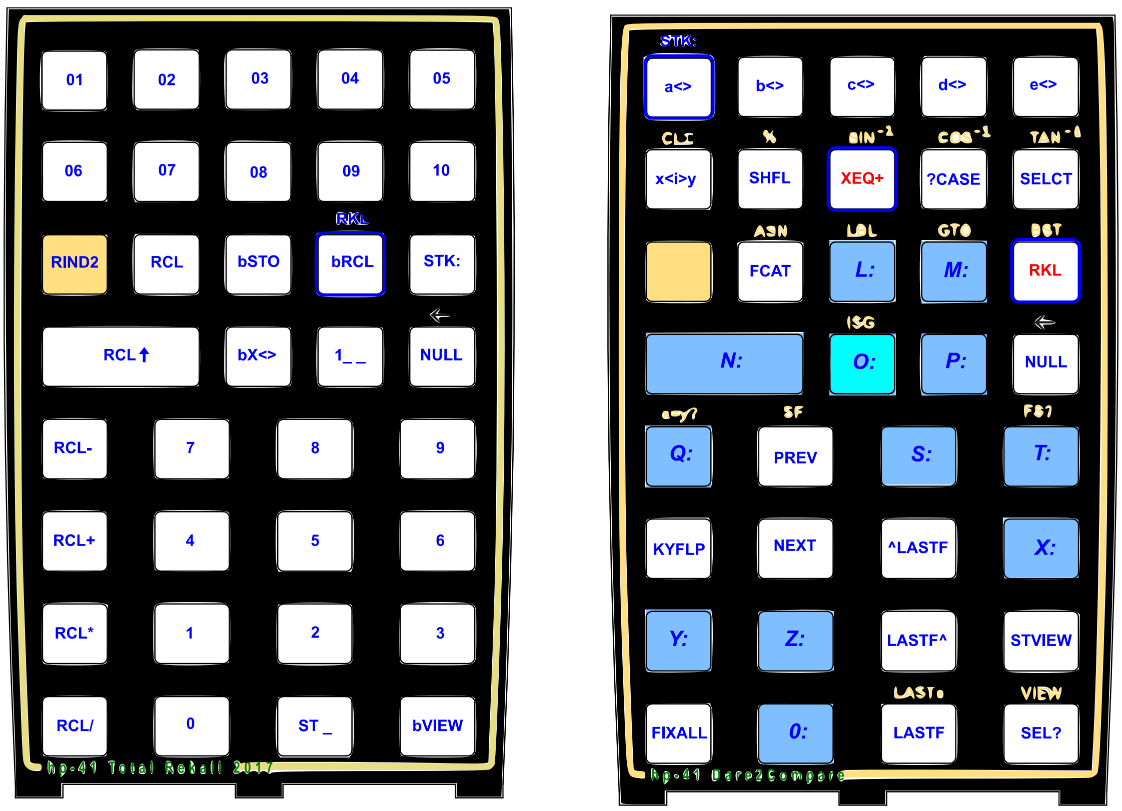


Figure 1: RKL Hot keys (left) and Main Overlay (right).



Summary Function Table.

#	Function	Description	Input	Dependency	Type	Author
0	**-WARP CORE+**	*Lib#4 Check & Splash*	none	Lib#4	MCODE	*Ángel Martin*
1	**ED+**	**Enhanced ASCII File Editor**	FName in ALPHA	Lib#4	MCODE	*Hp – Á.Martin*
2	**XEQ+**	**Auto-Complete Mode**	**Initial letter, hot keys**	Lib#4	MCODE	*Ángel Martin*
3	**?CASE _ _**	is case value	Value in prompt / Next Line	Lib#4	MCODE	*Ángel Martin*
4	**RKL _ _**	*Enhanced RCL function*	Prompts for RG#.	Lib#4	MCODE	*Ángel Martin*
5	**RC- _ _**	RCL Subtraction	RG# in prompt / Next Line	Lib#4	MCODE	*Ángel Martin*
6	**RC+ _ _**	RCL Addition	RG# in prompt / Next Line	Lib#4	MCODE	*Ángel Martin*
7	**RC* _ _**	RCL Multiply	RG# in prompt / Next Line	Lib#4	MCODE	*Ángel Martin*
8	**RC/ _ _**	RCL Divide	RG# in prompt / Next Line	Lib#4	MCODE	*Ángel Martin*
9	**RC^ _ _**	RCL Power	RG# in prompt / Next Line	Lib#4	MCODE	*Ángel Martin*
10	**RIND2 _ _**	RCL IND IND (IND ...)	RG# in prompt / Next Line	Lib#4	MCODE	*Ángel Martin*
11	**SELCT _**	selects variable	RG# in prompt / Next Line	Lib#4	MCODE	*Ángel Martin*
12	**SHFL _ _ _ _ _ _**	Stack Shuffle	five stack regs in prompt	Lib#4	MCODE	*Ángel Martin*
13	**R0R4 _ _ _ _ _ _**	Register Shuffle	Five Reg numbers in prompt	Lib#4	MCODE	*Ángel Martin*
14	**SIND2 _ _**	STO IND IND (IND ...)	RG# in prompt / Next Line	Lib#4	MCODE	*Ángel Martin*
15	**A<>RG _ _**	Alpha Exchange	RG# in prompt / Next Line	Lib#4	MCODE	*Ken Emery*
16	**WARP _ _**	*Sub-function Launcher*	Index / Name at prompt	Lib#4	MCODE	*Ángel Martin*
17	**SST+**	*Continuous SST/BST*	Name in prompt	Lib#4	MCODE	*Nelson Crowle*
18	**Y<> _ _**	Swap Y and Register	RG# in prompt / Next Line	Lib#4	MCODE	*Greg McClure*
19	**Z<> _ _**	Swap Z and register	RG# in prompt / Next Line	Lib#4	MCODE	*Ángel Martin*
20	**T<> _ _**	Swap T and register	RG# in prompt / Next Line	Lib#4	MCODE	*Ángel Martin*
21	**L<> _ _**	Swap L and register	RG# in prompt / Next Line	Lib#4	MCODE	*Ángel Martin*
22	**M<> _ _**	Swap M and register	RG# in prompt / Next Line	Lib#4	MCODE	*Ángel Martin*
23	**N<> _ _**	Swap N and register	RG# in prompt / Next Line	Lib#4	MCODE	*Ángel Martin*
24	**O<> _ _**	Swap O and register	RG# in prompt / Next Line	Lib#4	MCODE	*Ángel Martin*
25	**P<> _ _**	Swap P and register	RG# in prompt / Next Line	Lib#4	MCODE	*Ángel Martin*
26	**ST<>RG _ _**	Stack Exchange	RG# in prompt / Next Line	Lib#4	MCODE	*Ángel Martin*
27	**-STKTST**	*Function Builder*	Prompts for Reg and operation	Lib#4	MCODE	*Ángel Martin*
28	**?0= _ _**	Equal to Zero Test	RG# in prompt / Next Line	Lib#4	MCODE	*Ángel Martin*
29	**?0# _ _**	Different from Zero Test	RG# in prompt / Next Line	Lib#4	MCODE	*Ángel Martin*
30	**?0< _ _**	Greater than Zero test	RG# in prompt / Next Line	Lib#4	MCODE	*Ángel Martin*
31	**?0<= _ _**	Greater than/Equal to Zero Test	RG# in prompt / Next Line	Lib#4	MCODE	*Ángel Martin*
32	**?0> _ _**	Less than Zero Test	RG# in prompt / Next Line	Lib#4	MCODE	*Ángel Martin*
33	**?0>= _ _**	Less than/ Equal to Zero Test	RG# in prompt / Next Line	Lib#4	MCODE	*Ángel Martin*
34	**?X= _ _**	Equal to X test	RG# in prompt / Next Line	Lib#4	MCODE	*Ángel Martin*
35	**?X# _ _**	Different from X test	RG# in prompt / Next Line	Lib#4	MCODE	*Ángel Martin*
36	**?X< _ _**	Greater than X test	RG# in prompt / Next Line	Lib#4	MCODE	*Ángel Martin*
37	**?X<= _ _**	Greater than/Equal to X test	RG# in prompt / Next Line	Lib#4	MCODE	*Ángel Martin*
38	**?X> _ _**	Less than X Test	RG# in prompt / Next Line	Lib#4	MCODE	*Ángel Martin*
39	**?X>= _ _**	Less than or equal to X test	RG# in prompt / Next Line	Lib#4	MCODE	*Ángel Martin*
40	**?Y= _ _**	Equal to Y test	RG# in prompt / Next Line	Lib#4	MCODE	*Ángel Martin*
41	**?Y# _ _**	Different from Y test	RG# in prompt / Next Line	Lib#4	MCODE	*Ángel Martin*
42	**?Y< _ _**	Greater than Y test	RG# in prompt / Next Line	Lib#4	MCODE	*Ángel Martin*
43	**?Y<= _ _**	Greater than or equal to Y test	RG# in prompt / Next Line	Lib#4	MCODE	*Ángel Martin*
44	**?Y> _ _**	Less than Y Test	RG# in prompt / Next Line	Lib#4	MCODE	*Ángel Martin*
45	**?Y>= _ _**	Less than or equal to Y test	RG# in prompt / Next Line	Lib#4	MCODE	*Ángel Martin*
46	**?Z= _ _**	Equal to Z test	RG# in prompt / Next Line	Lib#4	MCODE	*Ángel Martin*
47	**?Z# _ _**	Different from Z test	RG# in prompt / Next Line	Lib#4	MCODE	*Ángel Martin*
48	**?Z< _ _**	Greater than Z test	RG# in prompt / Next Line	Lib#4	MCODE	*Ángel Martin*
49	**?Z<= _ _**	Greater than or equal to Z test	RG# in prompt / Next Line	Lib#4	MCODE	*Ángel Martin*
50	**?Z> _ _**	Less than Z Test	RG# in prompt / Next Line	Lib#4	MCODE	*Ángel Martin*
51	**?Z>= _ _**	Less than or equal to Z test	RG# in prompt / Next Line	Lib#4	MCODE	*Ángel Martin*
52	**?T= _ _**	Equal to T test	RG# in prompt / Next Line	Lib#4	MCODE	*Ángel Martin*

#	Function	Description	Input	Dependency	Type	Author
53	?T# _ _	Different from T test	RG# in prompt / Next Line	Lib#4	MCODE	*Ángel Martin*
54	?T< _ _	Greater than T test	RG# in prompt / Next Line	Lib#4	MCODE	*Ángel Martin*
55	?T<= _ _	Greater than or equal to T test	RG# in prompt / Next Line	Lib#4	MCODE	*Ángel Martin*
56	?T> _ _	Less than T Test	RG# in prompt / Next Line	Lib#4	MCODE	*Ángel Martin*
57	?T>= _ _	Less than or equal to T test	RG# in prompt / Next Line	Lib#4	MCODE	*Ángel Martin*
58	?L= _ _	Equal to L test	RG# in prompt / Next Line	Lib#4	MCODE	*Ángel Martin*
59	?L# _ _	Different from L test	RG# in prompt / Next Line	Lib#4	MCODE	*Ángel Martin*
60	?L< _ _	Greater than L test	RG# in prompt / Next Line	Lib#4	MCODE	*Ángel Martin*
61	?L<= _ _	Greater than or equal to T test	RG# in prompt / Next Line	Lib#4	MCODE	*Ángel Martin*
62	?L> _ _	Less than L Test	RG# in prompt / Next Line	Lib#4	MCODE	*Ángel Martin*
63	?L>= _ _	Less than or equal to L test	RG# in prompt / Next Line	Lib#4	MCODE	*Ángel Martin*

This module also includes a VERY large set of sub-functions arranged in an Auxiliary FAT, as follows:

0	**-STK SWAPS**	Section Header		Lib#4	MCODE	*Ángel Martin*
1	a<> _ _	Swap a and register	RG# in prompt / Next Line	Lib#4	MCODE	*Ángel Martin*
2	b<> _ _	Swap b and register	RG# in prompt / Next Line	Lib#4	MCODE	*Ángel Martin*
3	c<> _ _	Swap c and register	RG# in prompt / Next Line	Lib#4	MCODE	*Ángel Martin*
4	d<> _ _	Swap d and register	RG# in prompt / Next Line	Lib#4	MCODE	*Ángel Martin*
5	e<> _ _	Swap e and register	RG# in prompt / Next Line	Lib#4	MCODE	*Ángel Martin*
6	}-<> _ _	Swap |- and register	RG# in prompt / Next Line	Lib#4	MCODE	*Ángel Martin*
7	Q<> _ _	Swaps Q and registers	RG# in prompt / Next Line	Lib#4	MCODE	*Ángel Martin*
8	?M= _ _	Equal to M test	RG# in prompt / Next Line	Lib#4	MCODE	*Ángel Martin*
9	?M# _ _	Different from M test	RG# in prompt / Next Line	Lib#4	MCODE	*Ángel Martin*
10	?M< _ _	Greater than M test	RG# in prompt / Next Line	Lib#4	MCODE	*Ángel Martin*
11	?M<= _ _	Greater than or equal to M test	RG# in prompt / Next Line	Lib#4	MCODE	*Ángel Martin*
12	?M> _ _	Less than M Test	RG# in prompt / Next Line	Lib#4	MCODE	*Ángel Martin*
13	?M>= _ _	Less than or equal to M test	RG# in prompt / Next Line	Lib#4	MCODE	*Ángel Martin*
14	?N= _ _	Equal to N test	RG# in prompt / Next Line	Lib#4	MCODE	*Ángel Martin*
15	?N# _ _	Different from N test	RG# in prompt / Next Line	Lib#4	MCODE	*Ángel Martin*
16	?N< _ _	Greater than N test	RG# in prompt / Next Line	Lib#4	MCODE	*Ángel Martin*
17	?N<= _ _	Greater than or equal to N test	RG# in prompt / Next Line	Lib#4	MCODE	*Ángel Martin*
18	?N> _ _	Less than N Test	RG# in prompt / Next Line	Lib#4	MCODE	*Ángel Martin*
19	?N>= _ _	Less than or equal to N test	RG# in prompt / Next Line	Lib#4	MCODE	*Ángel Martin*
20	?O= _ _	Equal to O test	RG# in prompt / Next Line	Lib#4	MCODE	*Ángel Martin*
21	?O# _ _	Different from O test	RG# in prompt / Next Line	Lib#4	MCODE	*Ángel Martin*
22	?O< _ _	Greater than O test	RG# in prompt / Next Line	Lib#4	MCODE	*Ángel Martin*
23	?O<= _ _	Greater than or equal to O test	RG# in prompt / Next Line	Lib#4	MCODE	*Ángel Martin*
24	?O> _ _	Less than O Test	RG# in prompt / Next Line	Lib#4	MCODE	*Ángel Martin*
25	?O>= _ _	Less than or equal to O test	RG# in prompt / Next Line	Lib#4	MCODE	*Ángel Martin*
26	?P= _ _	Equal to P test	RG# in prompt / Next Line	Lib#4	MCODE	*Ángel Martin*
27	?P# _ _	Different from P test	RG# in prompt / Next Line	Lib#4	MCODE	*Ángel Martin*
28	?P< _ _	Greater than P test	RG# in prompt / Next Line	Lib#4	MCODE	*Ángel Martin*
29	?P<= _ _	Greater than or equal to P test	RG# in prompt / Next Line	Lib#4	MCODE	*Ángel Martin*
30	?P> _ _	Less than P Test	RG# in prompt / Next Line	Lib#4	MCODE	*Ángel Martin*
31	?P>= _ _	Less than or equal to P test	RG# in prompt / Next Line	Lib#4	MCODE	*Ángel Martin*
32	?Q= _ _	Equal to Q test	RG# in prompt / Next Line	Lib#4	MCODE	*Ángel Martin*
33	?Q# _ _	Different from Q test	RG# in prompt / Next Line	Lib#4	MCODE	*Ángel Martin*
34	?Q< _ _	Greater than Q test	RG# in prompt / Next Line	Lib#4	MCODE	*Ángel Martin*
35	?Q<= _ _	Greater than or equal to Q test	RG# in prompt / Next Line	Lib#4	MCODE	*Ángel Martin*
36	?Q> _ _	Less than Q Test	RG# in prompt / Next Line	Lib#4	MCODE	*Ángel Martin*
37	?Q>= _ _	Less than or equal to Q test	RG# in prompt / Next Line	Lib#4	MCODE	*Ángel Martin*
38	**-SELECT FNS**	Section Header				

39	**?S=**	Equal to S test	Data in sel and target	Lib#4	MCODE	*Ángel Martin*
40	**?S#**	Different from S test	Data in sel and target	Lib#4	MCODE	*Ángel Martin*
41	**?S<**	Greater than S test	Data in sel and target	Lib#4	MCODE	*Ángel Martin*
42	**?S< _ _**	Greater than or equal to S test	Data in sel and target	Lib#4	MCODE	*Ángel Martin*
43	**?S> _ _**	Less than S Test	Data in sel and target	Lib#4	MCODE	*Ángel Martin*
44	**?S>= _ _**	Less than or equal to S test	Data in sel and target	Lib#4	MCODE	*Ángel Martin*
45	**NEXT**	increment selection	SEL variable	Lib#4	MCODE	*Ángel Martin*
46	**PREV**	decrement selection	SEL variable	Lib#4	MCODE	*Ángel Martin*
47	**S<> _ _**	Swap Selected & Target Regs	Target Reg in prompt	Lib#4	MCODE	*Ángel Martin*
48	**SEL?**	Shows the selected variable	SEL variable	Lib#4	MCODE	*Ángel Martin*
49	**SRCL**	Recalls Value in Selected var	None	Lib#4	MCODE	*Ángel Martin*
50	**SSTO**	Stores value in selected var	Value in X	Lib#4	MCODE	*Ángel Martin*
51	**SVIEW**	Shows Selected var contents	SEL variable Value	Lib#4	MCODE	*Ángel Martin*
52	**-WARP FNS**	Shows Splash Screen	none	Lib#4	MCODE	*Nelson F. Crowle*
53	**A<>ST**	Exchange Alpha & Stack	Values in ALPHA and stack	Lib#4	MCODE	*Ángel Martin*
54	**AIRCL _ _**	Integer ARCL	Prompts for rg#	Lib#4	MCODE	*Ángel Martin*
55	**AUXFAT**	Shows pages w/ Aux FAT	none	Lib#4	MCODE	*Ángel Martin*
56	**bRCL _**	Buffer reg recall	buffer reg# (1-5)	Lib#4	MCODE	*Ángel Martin*
57	**bSTO _**	Buffer reg Storage	buffer reg# (1-5)	Lib#4	MCODE	*Ángel Martin*
58	**bVIEW _**	Buffer Reg View	Buffer reg# (1-5)	Lib#4	MCODE	*Ángel Martin*
59	**bX<> _**	Buffer Reg Exchange	buffer reg# (1-5)	Lib#4	MCODE	*Ángel Martin*
60	**CPYBNK _"_:_**	Copies Bank#	Bank#, from-to pages	Lib#4	MCODE	*Ángel Martin*
61	**DSNEX**	Decrement & skip if not Equal	Control word in X	Lib#4	MCODE	*Ángel Martin*
62	**FINDX**	Find register containing X	Value in X	Lib#4	MCODE	*Ángel Martin*
63	**FIXALL**	Activates Fix ALL mode	none	Lib#4	MCODE	*Ángel Martin*
64	**GETST _ _**	Get Status Regs from File	# Regs, FileName	Lib#4	MCODE	*Ángel Martin*
65	**HX2ROM A_:_**	From Hex code to ROM#	Hex code	Lib#4	MCODE	*Greg McClure*
66	**INFO$ _**	Shows Function Info	Inputs for Name	Lib#4	MCODE	*Ángel Martin*
67	**IOBUS _**	Shows Bus by category	0,1,2,3 for Page types	Lib#4	MCODE	*Ángel Martin*
68	**ISLEX**	Increment and Skip if Equal	Control word in X	Lib#4	MCODE	*Ángel Martin*
69	**KAFLP _**	Flips ALL Key assignments	none	Lib#4	MCODE	*Ángel Martin*
70	**KYFLP _**	Flips Key assignments	Pressed key	Lib#4	MCODE	*Ángel Martin*
71	**^LASTF _**	**Prompts for FName to add**	**Buffer #9**	Lib#4	MCODE	*Ángel Martin*
72	**LASTF^**	**Starts LastF review**	**Hot keys, Buffer #9**	Lib#4	MCODE	*Ángel Martin*
73	**POPRTN**	Pop RTN stack from Buffer	None	Lib#4	MCODE	*Poul Kaarup*
74	**PUSHRTN**	Push RTN stack to buffer	none	Lib#4	MCODE	*Poul Kaarup*
75	**ROM2HX_ _:_ _**	From ROM# to Hex Code	ROM id#	Lib#4	MCODE	*Greg McClure*
76	**RTN?**	Tests for pending RTNs	YES/NO, skips if False	Lib#4	MCODE	*Doug Wilder*
77	**RTNS**	Number of pending RTNs	Pust in X, Lifts Stack	Lib#4	MCODE	*Ángel Martin*
78	**SAVEST _ _**	Save Status Regs	#Regs, FileName	Lib#4	MCODE	*Ángel Martin*
79	**SFLNCH _**	Sub-function Launcher-launcher	Page# in Prompt	Lib#4	MCODE	*Ángel Martin*
80	**ST<>Σ**	Swap Stack and ΣRegs	none	Lib#4	MCODE	*Nelson F. Crowle*
81	**STVIEW**	Full Stack View	None	Lib#4	MCODE	*Ángel Martin*
82	**X<I>Y**	Exchange IND(X) & IND(Y)	Values in X, Y	Lib#4	MCODE	*Nelson F. Crowle*
83	**X=YZ?**	Double Comparison	Values in X, Y, Z	Lib#4	MCODE	*Ken Emery*
84	**X=YZT?**	Triple Comparison	Values in Stack	Lib#4	MCODE	*Poul Kaarup*
85	**XEQ ' _**	Executes CAT1 function	Values in buffer	Lib#4	MCODE	*Ángel Martin*
86	**XEQ$ _**	Universal Execute	Prompts for Name	Lib#4	MCODE	*Ángel Martin*
87	**-XTRA FNS**	Shows Splash Screen	none	Lib#4	MCODE	*Nelson F. Crowle*
88	**?MEM**	System Indicators	Shows Memory Left	Lib#4	MCODE	*Ángel Martin*
89	**0<> _ _**	Register Clearimg	RG# in prompt / Next Line	Lib#4	MCODE	*Ángel Martin*
90	**A<>A**	ALPHA Reverse	Text in ALPHA	-	MCODE	*Paul Kaarup*
91	**ALPHB**	Alphabetize	Sorts alphabetically	-	MCODE	*Poul Kaarup*
92	**BFVIEW**	View Buffer	Buf id# in X	Lib#4	MCODE	*Ángel Martin*
93	**EASTER**	Easter Date Finder	Year in X	Lib#4	MCODE	*Kari Pasanen.*
94	**WF$ _**	*Sub-function Launcher by Name*	Name in prompt	Lib#4	MCODE	*Ángel Martin*

95	**LODB _ _**	Load Bytes in RAM	Byte codes in prompts	Lib#4	MCODE	*Nelson F. Crowle*
96	**LODB+ _ _**	Load Bytes in RAM	Byte Codes in prompts	Lib#4	MCODE	*Nelson F. Crowle*
97	**METRON**	Metronome	Beats per min in X	Lib#4	MCODE	*Mark Power*
98	**PGCAT**	Page Catalog	Press key to halt listing	Lib#4	MCODE	*Steen Petersen*
99	**POP**	POP LIFO Launcher	shows I:A:F:X:Z:T:R	Lib#4	MCODE	*Doug Wilder*
100	**PUSH**	PUSH LIFO Launcher	shows I:A:F:X:Z:T:R	Lib#4	MCODE	*Doug Wilder*
101	**PROMT**	Variable Prompt	Number of fields in X	Lib#4	MCODE	*Nelson F. Crowle*
102	**REC-**	Previous ASCII Record	FileName in ALPHA	-	MCODE	*Ángel Martin*
103	**REC+**	Next ASCII record	FileName in ALPHA	-	MCODE	*Ángel Martin*
104	**REC+X**	Advance Record by X	Fname in ALPHA, x in X	-	MCODE	*Ángel Martin*
105	**RSORT**	Sorts {R00-R03}	None	Lib#4	MCODE	*Ángel Martin*
106	**SSORT**	Sorts XYZT stack	none	Lib#4	MCODE	*Ángel Martin*
107	**RCLΣ**	Recall Σregs to Stack	Data in Stack	Lib#4	MCODE	*Ken Emery*
108	**STOΣ**	Stores Stack in ΣRegs	Data in ΣRegs	Lib#4	MCODE	*Mark Power*
109	**XIND2 _ _**	X<> IND IND	RG# in prompt / Next Line	Lib#4	MCODE	*Ángel Martin*
110	**XROM$ _**	XROM Call Decoder	Program name in ALPHA	Lib#4	MCODE	*Klaus Huppertz*
111	**CAT+ _**	Sub-function CATALOG	has HOT keys	Lib#4	MCODE	*Ángel Martin*

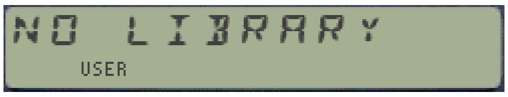
Pink Background: New functions in the Bank-Switched versions.

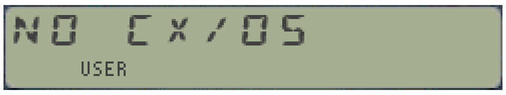
General remark about the overarching module architecture.

This project has grown substantially from the initial Total-Rekall sketches in the early module. Making it all work in a wholistic manner hasn't been easy, especially consideing that the code has had multiple revisions and additions in the last couple of years, some of them obvious afterthoughts. This somehow is a less-than-ideal implementation from the programming side, lacking an all-inclusive, tops-down approach from the scratch. Nevertheless looking at the final results you wouldn't notice any negative impact of the implementation in the actual usage of the functionality.

Module Dependencies.

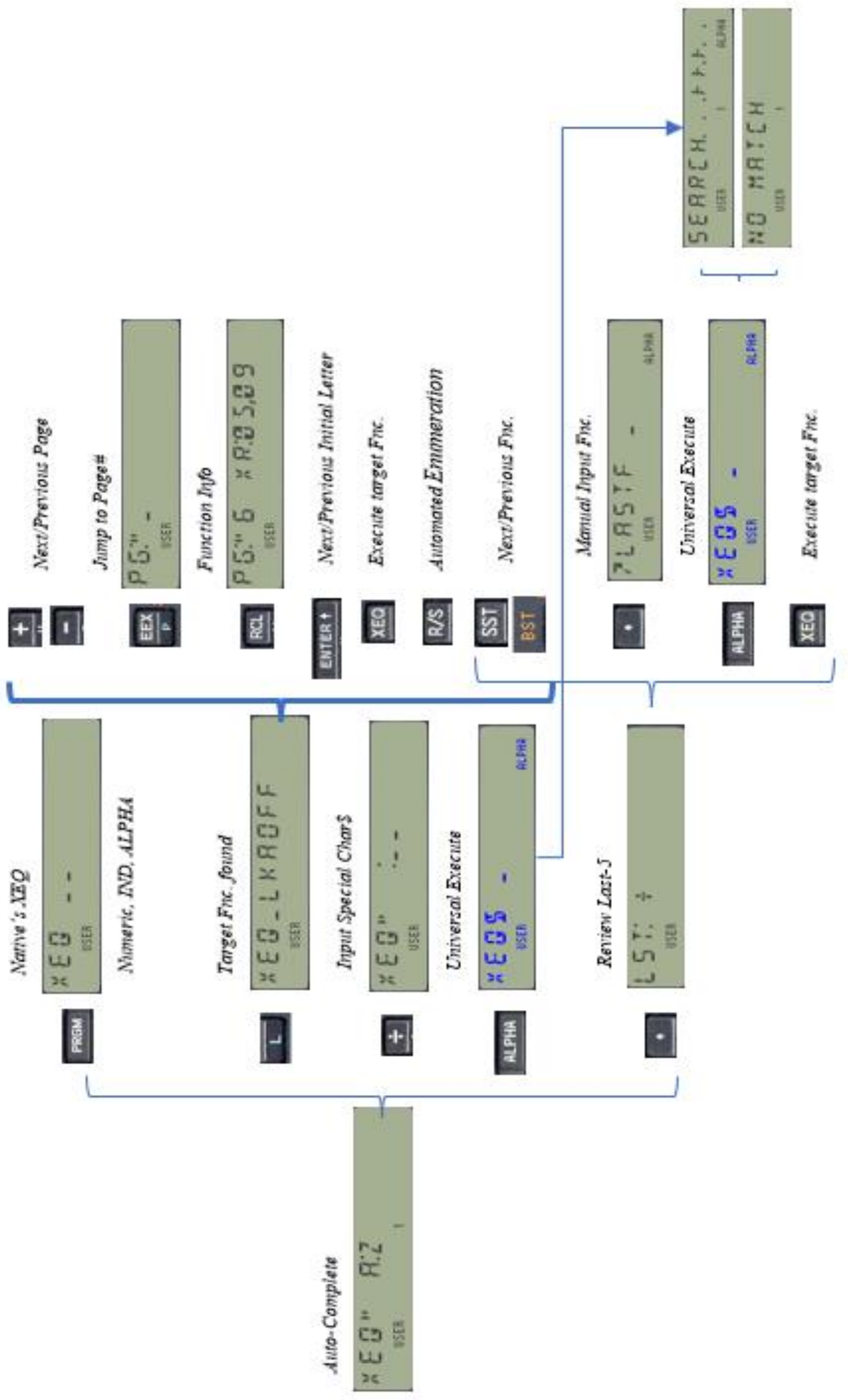
The WARP_Core module is a Library#4-aware module, and therefore requires the Library#4 (revision R58 or higher) to be plugged in the calculator. It also requires the CX OS, as some CX internal routines are used. If the Library#4 is missing or the machine is not a CX the errors will halt it to avoid likely problems.


NO LIBRARY
USER


NO CX/OS
USER

Also note that the WARP_Core is a bank-switched module: its footprint is only 4k in the I/O bus, yet there are three 4k-pages involved holding the code. This is important to properly configure it using hardware devices like Clonx/NoV_RAM or MLDL2k. For the CL board, the module id# is not surprisingly "**WARP**", and it will automatically be plugged using **PLUG**. Note that WRAP is not compatible with page#6 – avoid plugging it in that location.

Note also that ***you should avoid plugging it together with another bank-switched module (with a 4k footprint) sharing the same logical external port.***



XEQ+ Navigation

What's new in the 2020-23 "WARP_Core" editions?

1. The Auto-complete mode. **XEQ+**

If you've been following the evolution of the "Total_Rekall" module you'd no doubt expect grand and important new things of a major revision like this one – and you won't be disappointed, because this edition includes the all-new, long-awaited, Auto-Complete mode for XEQ functions.

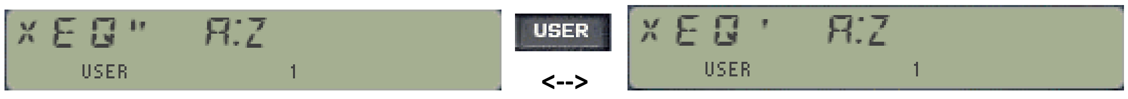
When you call the **XEQ+** function a new mode of execution opens up to the user; one where instead of spelling the complete function names at the alpha prompt, only the first initial letter is entered and the calculator does the rest for you – with a few control hot-keys to navigate the complete system (CAT'2), from page #3 up to the top in page #F for the plug-in modules, and page #1 for the native OS functions (in CAT'3).

This is akin to the "auto-complete" functionality popular on other systems, very useful to assist in the selection of those available functions in the current ROM configuration. Because of the finite number of possible options (with an absolute total maximum of 630 functions when all pages are filled up with modules each having 64 entries in their FATs), limiting the auto-completion to the first character is not a shortcoming, but a practical design criterion to keep the code size and execution times within reasonable parameters.

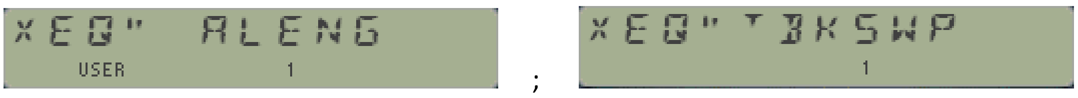
Using Auto-Complete.

In short: the function **XEQ+** starts a new mode by prompting for an initial character letter or number. When that selection is made and after a short search time (negligible on the CL for sure) it will present all functions currently available in the bus that begin with that letter - commencing the search in page#3 up until page #F. The listing can be done manually (SST) or continuous (R/S), and several navigation keys are included: jump page, back-up page, next function, previous function, next letter, previous letter.

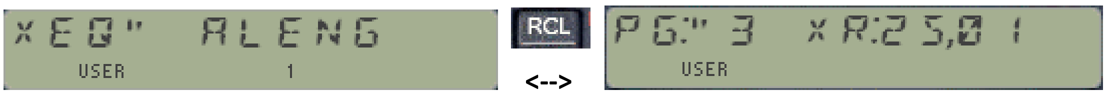
The initial prompt is ready to look in the plug-in section of the system bus, i.e. from page #3 up to page #F (15d). This is indicated by a double-quotes character in the display. Note that this representation changes automatically to a single-quotes character if the target function is located in the O/S, i.e. for the "native" functions in CAT'3. You can use the USER key to toggle between both:



For XROM functions, both MCODE functions and FOCAL programs will be shown:



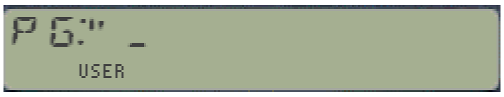
Once you've locked on your target function simply press XEQ to execute it, or [ ] ASN to assign it to the key of your choice. If you're not sure this is your choice (say duplicates or similarly spelled ones exist), pressing RCL will show you some vital signs of the function, such page# and XROM id#

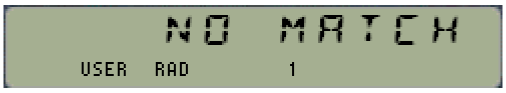


Pressing the ENTER^ navigation key, you can change the letter sought to the next one alphabetical, always starting at the current page and moving upwards.

Manually Changing the Searched Page.

You can move up or down one page using the + or - control keys. If no target exists in the next page using + the engine will keep looking look in pages above it, but not so using - for the previous page. Eventually if no function starting with the selected letter exists, a "NO MATCH" information message will be briefly shown left-justified, and the current function will persist. You can also force the searched page by using the EEX or [**P**] hot-key, and inputting the initial page to start the search from. The same upwards/downwards behavior applies when there are no target functions in a forced page location, using the + or - control keys for pages jumping, or the EEX key for a forced destination:





Functions from CAT'3

The native OS functions are fully supported by the Auto-Complete engine. You can access the OS area either by typing "0", "1", or "2" directly at the page prompt triggered by EEX, or decreasing the page# using - while a function from pg# 3 is shown (provided that such first letter is also available in the OS group, as per the previous descriptions).

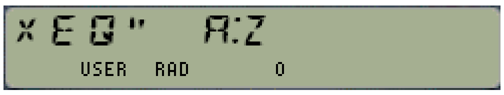
In case you wonder, page#4 is simply skipped over, while pages 0-1-2 (indistinctly) default to page#1 to include the CAT'3 functions as well – so this functionality includes the standard functions of the calculator (such as BEEP, FACT, MOD, SDEV, etc.). This support includes their inclusion on the LASTF list for quick access of recently executed functions.

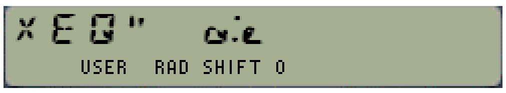
Back-door to the Standard XEQ

The PRGM key is also active as a hot key to invoke the native XEQ function. Use it if you want to revert to the standard OS method to access numeric labels or a local label (A-H, a-e) within a user program, or to spell the function name in ALPHA mode; by pressing ALPHA and then spell the name as usual. However this method is now superseded by the "Universal Execute" as will be described later on.

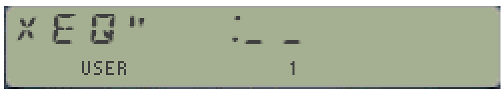
Typing in Special Characters.

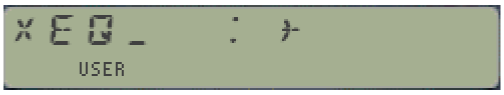
Lower case characters (a-e), numbers and all other key-able special chars (like %, Σ, ^, #, $, etc.) are accessed using the shifted keys in the standard ALPHA keyboard. Simply press the [SHIFT] key to toggle between the upper/lower case modes:





Another option is provided pressing the / key at the main prompt, to use special characters – even if not key-able but allowed in function names. This makes it possible to search for function names staring with "µ", the forwards and backwards geese, or all the little men just to name a few.



"2E" =>


Automated enumeration.

If you'd rather see an automated enumeration of the options then pressing [R/S] will show all functions meeting those criteria up until the end of the bus. You can quit the listing at any time pressing any key, and then press [XEQ] or [ASN] to perform the action once halted.

The enumeration will end with the last target function displayed in the LCD – not showing the "NO MATCH" error message in this case. At this point you are still in the **XEQ+** mode, so the hot keys continue to be available.

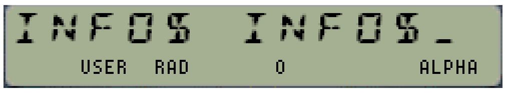
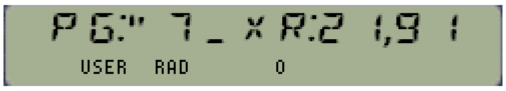
Note that (with the exception of the native OS group), functions are not listed in alphabetical order, but in *sequential order,* as they're found in the respective FAT's of the modules currently plugged in the calculator. The only condition is that they all begin with the letter chosen at the initial prompt.

Firing blanks with INFO$

If all you want to do is finding out the function's Page# and XROM data, you can use **INFO$** instead if **XEQ$**. This variant will search for the (sub) function which name is typed at the prompt, and if found it'll show the information screen directly. Note however that the sought for (sub) function will not be executed, nor will it be saved in the LAST-7 buffer.

INFO$ can be accessed directly from the **–STKT** launcher using the [SHIFT] [XEQ] shortcut.

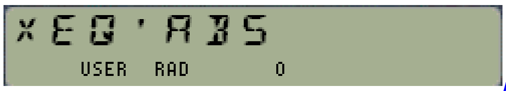
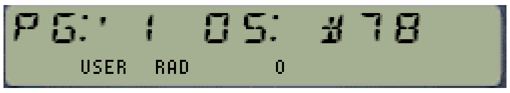
For example, try using **INFO$** on itself to find out its own function parameters:

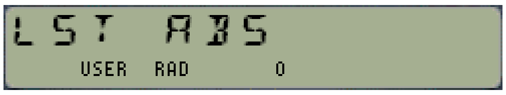
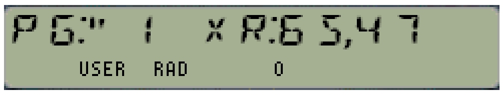

```
INFO$ INFO$_
USER RAD 0 ALPHA
```
=>

```
PG:" 7_XR:21,91
USER RAD 0
```

There you go, according to that it's the 91st. sub-function (the underscore tells that, more on this shortly) of the WARP Core module, XROM id#=21 and currently plugged into page #7

Caveat Emptor:

For O/S functions in the mainframe this information will always show page#1 (which it's correct) but the rest is somewhat a "poetic license", since they don't belong to any plug-in ROM and therefore it makes no much sense to refer to XROM id# or FAT indices. It's also different from the same information screen shown from the **XEQ+** facility, as it's shown below for the **ABS** function:


```
XEQ'ABS
USER RAD 0
```
, RCL

```
PG:' 1 OS: #78
USER RAD 0
```
Where #78 indicates it's the 78th entry in the internal address table,


```
LST ABS
USER RAD 0
```
, RCL

```
PG:" 1 XR:65,47
USER RAD 0
```
Here the XR: data is not intuitively obvious to decipher but it's related to the function code for assignment purposes.

Sub-functions are included in the search.

The latest revisions of the WARP module provide the capability to also include the sub-functions in the search, thus they will be shown when the first-letter criteria are met. In case you are not familiar with them, this is a special functionality present in several advanced modules than breaks the 64-function FAT barrier of the O/S. - see the list below for details.

Sub-functions are structured in Auxiliary FATs, different from the main FAT atop of each module page. Since they are not included in the main FATs, the O/S knows nothing about them and therefore they are not accessible by the standard XEQ function. This means they need another way to be invoked – and typically each of those advanced modules has at least a dedicated launcher. More about this later, in the "Universal Execute" section.

Module	Aux FAT Location	Launchers	# Sub-funs
41Z Deluxe	Middle of Lower page	ZF$, ZF#	62
AMC_OS/X	Middle of page	XF$, XF#	22
CL X-Mem Manager	Middle of page	YF$, YF#	22
Formula Evaluation	Middle of page	SF$, SF#	24
X-Mem TWIN	Middle of Page	TF$, TF#	21
HEPAX_4H	Top of Bank-3, Middle of bank-3	HEPAX, XF$	21 + 25
HP-16C Simulator	Middle of page	16$, 16#	62
PowerCL Xtreme	Top of Bank-3, Top of bank-4	XQ1$, XQ2$	89 + 89
SandMath 4x4	Middle of Upper page	ΣF$, ΣF#	117
SandMatrix	Middle of Lower page	ΣM$, ΣM#	63
WARP_Core	Middle of page	WF$, WF#	112
Total System	**Indistinct**	**XEQ$**	**729**

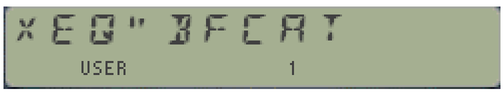
Table 1: Advanced Modules w/ Auxiliary FATs

The sub-functions are found whether they are located in (1:) an Auxiliary FAT or (2:) a Banked FAT atop the page – and (3:) as combination of both situations, i.e. an auxiliary FAT located at the *middle of a banked page* . This last case is only used by the HEPAX modules, and it's of relative interest because it just includes the replica of the X-Functions – only meaningful for non-CX systems.

The representation of the Sub-functions found during the enumeration is different from that of the (standard) Main functions: the single or double quotes character used by native and XROM main functions respectively is replaced by:

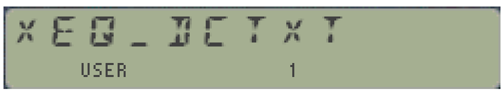
- An underscore character if the sub-function is in the **auxiliary** FAT (middle of the page)
- An *overscore* character when the sub-function is atop of a **banked** page, and
- A *colon/overscore* character when the sub-function is atop of a **second banked** page

See below the examples showing main function **BFCAT** and sub-function **DCTXT** (both from the AMC_OSX module), and with sub-functions **BFVIEW** and **BANKS?** from the PowerCl_Extreme module (in banks 3 and 4 respectively):

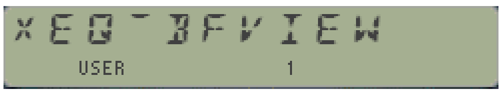

XEQ" BFCAT
USER 1

Main function

;

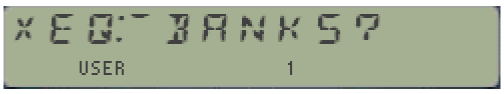

XEQ_DCTXT
USER 1

Auxiliary FAT


XEQ¯BFVIEW
USER 1

Banked FAT, bk#3

;


XEQ:¯BANKS?
USER 1

Banked FAT, bk#4

Note that the same punctuation convention is used in the "PG#:" and "LAST:" information screens.

Executing Sub-functions.

For complete location information of a sub-function, we need to know its index# within the Auxiliary FAT, and the address location of said Aux. FAT – which, you'd remember, may even be in a banked page as well. With that information at hand it's possible to direct the program pointer to the beginning of the sub-function code for manual execution.

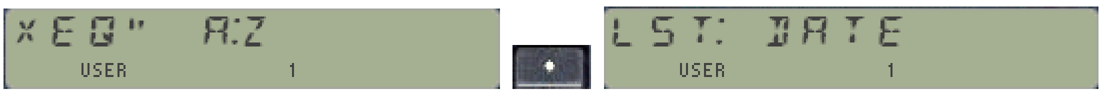
However, that method won't work when the sub-function needs to be entered into a program. In that instance the information required besides its index# within the Aux. FAT is the actual sub-function **launcher** code. This is exactly how sub-functions are entered in a program directly from the **XEQ+** prompt, which does all the legwork identifying the suitable launcher and proceeding by inserting two program steps with the launcher code and the index, as non-merged parameter.

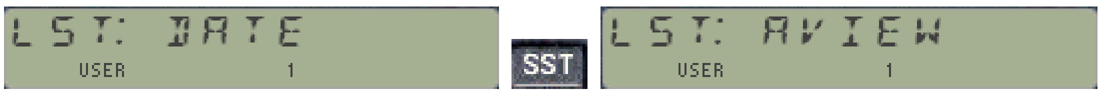
You should also check the "Searching for Auxiliary FATs" section to learn more about this subject, and to get familiar for the sub-functions **AUXFAT** and **SFLNCH**, very handy tools to manage these advanced structures across the entire system bus.

The Extended LAST-7 facility.

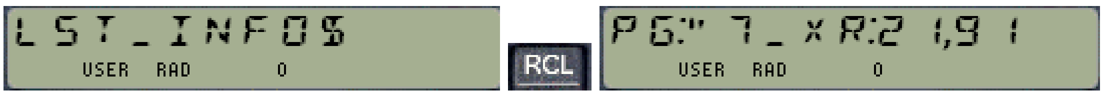
The [XEQ] operation will also add the executed function automatically to the enhanced **LASTF** facility, which now holds up to **seven** entries (say, didn't LastF stand for "last-five"? ;-). The storage includes both **Main** functions from the O/S or plug-in modules, and **Sub-functions** from auxiliary FATs, either in the main bank or in a bank-switched one. It is performed automatically and needs no user intervention. Two new utilities allow the user to review and execute these **(LASTF^)**, plus a manual mode to enter main functions into the list if so desired (**^LASTF)**.

- The first time you press the radix key you're invoking the **LASTF^** sub-function. This gives you the opportunity to re-access the last seven functions stored in the buffer, simply use the [SST] key to scroll the list, then press XEQ to execute it.

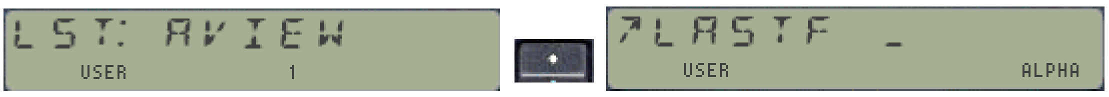




- Like with the [XEQ+] functions, pressing the [RCL] key here also brings up the *function information screen*, showing the Page# and XROM data of the function displayed in LAST-7 registers. This is very convenient if you want to double check that it's the right function before executing it.



- Pressing the radix key a second time invokes the **^LASTF** sub-function, which shows an editable field to manually enter a function name for its inclusion in the Last-Seven buffer - from where you can access it using the method described above.



It's therefore important to remark that **sub-functions will also be stored in the LAST-7 buffer**. This universal coverage guarantees that *any* command accessed via the **XEQ+** facility is logged in the Lasf-7 buffer. The only restriction for their recovery is that the plug-in modules are not moved between accesses to the LAST-7 facility. Implementing this level of coverage wasn't trivial, and it definitely ran into the "low of diminishing returns" – a lot of complex code to cover fringe cases – but the end result (and hopefully user experience too) is much more complete in this way.

LASTF stepping on LAST-7 Toes – and vice versa.

The LAST-7 engine uses buffer #9 to store the function id's. But as you probably already knew, the same buffer is also used for the same purpose by the individual sub-function launchers available in several advanced modules listed in table 1. These have pre-assigned locations within the buffer registers, b1 – b7, as seen in the tables below, therefore using them will override the corresponding entry placed there by XEQ+ . For example, using **ΣF#** or **ΣF$** in the SandMath is going to use the b3 register, thus overwriting whatever was put in it previously by **XEQ+** or **XEQ$.** The very WARP itself can create this issue too, since using **WF#** and **WF$** will overwrite the contents of the b5 buffer register.

How big is the offense? Well, for NUMERIC launchers it's a misdemeanor as both entries are compatible, but for ALPHABETIC launchers that's not the case, and they will likely display garbage characters when accessed by LAST-7

Buffer id#	Buffer Reg	Type	Used by:
09	B6	ID# or Name$	Formula Eval
	B5	ID# or Name$	CL-Xmem/WarpCore/GJM ROM
	B4	ID# or Name$	SandMatrix
	B3	ID# or Name$	SandMath
	B2	ID# or Name$	PowerCL / AMC_OS/X
	B1	ID# or Name$	41Z Deluxe / 16C Emulator
	B0	ID#	Header – Standard 41Z Module

Table 2.- LASTF from individual Launchers

Buffer id#	Buffer Reg	Type	Used for:
09	b6	ID# & addr	Sixth function
	b5	ID# & addr	Fifth function
	b4	ID# & addr	Fourth function
	b3	ID# &addr	Third function
	b2	ID# & addr	Second function
	b1	ID# & addr	First function
	b0	admin	Header

Table 3: LAST-7 from XEQ+ / XEQ$

Note that the LAST-7 engine uses a LIFO approach to store the information, whereby the registers are pushed up with every new entry and the last function is always in the bottom register b1.

Understanding the Function Search process.

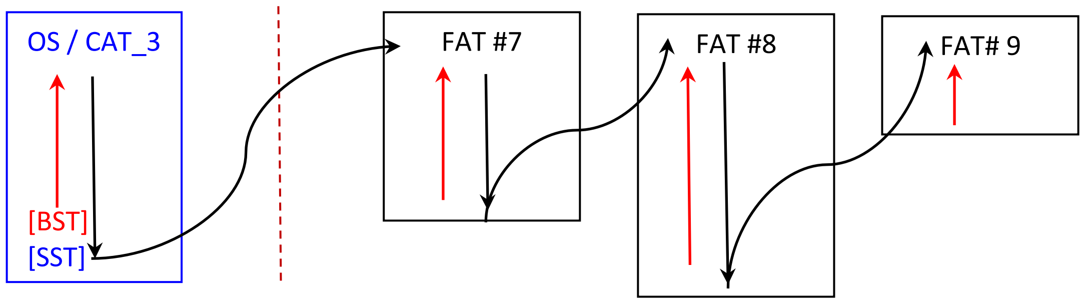
The prompt always shows the "domain" block used for the search, either the OS area or the I/O Bus:

• A single quote indicates OS area:	XEQ '
• Double quotes indicate the I/O bus:	XEQ "
• Underscore denotes sub-functions in Auxiliary FATs	XEQ_
• Upper-score denotes sub-functions in Banked FATs	either XEQ¯ or XEQ:¯

The search always starts in page #3 – which holds the extended functions FAT in the CX. If no functions starting with the target letter exist in that FAT, then the search continues in page#5, and keeps going up until page #F. Once that end is reached, the original prompt is shown if still no targets are found, i.e. there's no roll-over at this point.

When a function is found you can list the following starting with the same letter using [SST], which will automatically increase the page *within the domain block* when the current FAT is completed. This means it will show functions either within the OS, or within the I/O bus but not across the divide!

You can also move back to the previous function using [BST], *which will also move back to the previous page when the top of FAT has been reached.* Note that it's easy to know that the FAT always starts at the first byte within the page, but moving backwards the code needs to determine the end of the FAT in the previous page - by reading the number of functions in its second byte.



The figure above does not show Auxiliary or Banked FATS, yet the same functionality exists with them for the most part. There are, however, two important differences between the [SST] and [BST] enumeration features.

- The first one occurs when a gap is in-between pages; i.e. there's an empty page or a blank (page with no FAT), or no functions meeting the target criteria). In that situation the gap will be skipped moving upwards (the code will keep trying pages up until page #F is reached) but *the gap won't be crossed moving downwards*. Note that the same consideration applies to the [+] and [-] navigation keys: going upwards will skip blank pages (gaps) *but moving downwards will not*.

- The other important difference has to do with the sub-functions. The rule is that Auxiliary FATs are always included in the search, on either direction – but Banked FATs are only scanned going upwards. Therefore, sub-functions in Auxiliary FATs will be enumerated in both directions – but those in Banked FATs will be skipped going backwards.

Remember that you can always force the page# to look within, either by moving sequentially to the next/previous page (with a target letter present in both pages in the [-] case), or by jumping directly to a specific page# using [EEX]. This is how you can move to the OS area, i.e. pages #0 to #2: either by pressing [-] while a function from page #3 is locked-on, or by jumping directly to any of the first three pages (0-2).

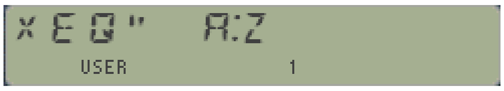
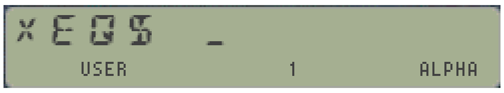
2.- *The Universal Execute.* **XEQ$**

The Auto-Complete mode is a very powerful way to "navigate" the entire system bus, looking for functions and sub-functions using only the initial letter of their names. This is often speedier and more convenient that the standard { XEQ, ALPHA } approach of the O/S – which requires typing correctly the complete name, - needed to be fully known by the user.

But each situation is different and sometimes it may be more convenient to use the direct full-name spelling method. The trouble child here are the sub-functions, invisible to the O/S and therefore not seen by the native XEQ function – regardless of its prowess, which aren't to be underestimated.

The solution is the new "*Universal Execute*" function, **XEQ$**, which allows you to type main function names, as well as sub-function names – located either in Auxiliary FATs or in Banked-switched FATs ! Therefore knowing the dedicated launcher to access a particular sub-function is no longer needed, freeing up the casual user from that requirement for the complete utilization of the full potential of the system.

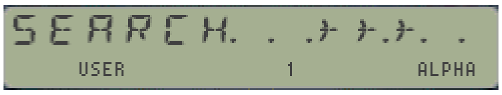
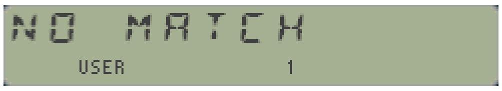
Accessing **XEQ$** is as simple as pressing the ALPHA key at the **XEQ+** prompt, or during the enumeration of the selected (sub)function. Once you do it the display will change to an editable field and ALPHA will be active for the typing of the name:


```
XEQ"    A:Z
  USER         1
```
ALPHA

```
XEQ$   _
  USER         1        ALPHA
```

Note: Because **XEQ$** is itself a sub-function, it's also possible to access it using the Warp Sub-function Launchers – either numerically with **WF#** and its index# = 085 , or alphabetically with **WF$**.

XEQ$ *replaces all Alphabetical sub-function launchers* from the advanced modules, as it supports manual (interactive) execution and Program entry of (sub)functions in a FOCAL program in RAM – pretty much like its "navigator" counterpart, **XEQ+**

The (sub)function search commences scanning the OS and the plug-in bus for matches, i.e. pretty much like the native XEQ except that FOCAL Labels in RAM programs will be ignored. If the name isn't found the code will sequentially scan all bus pages looking for Auxiliary and Banked FATs, and scan their contents for a suitable match. During the process the display shows an information message as seen below


```
SEARCH. . .+ +.+. .
  USER         1        ALPHA
```
perhaps:

```
NO  MATCH
  USER         1
```

LASTF support of XEQ$

The most beneficial aspect of the universal execute is possibly that all functions invoked will be added to the LAST-7 buffer for later accessibility. This includes OS functions from CAT'3, and MCODE functions or FOCAL programs from plugged-in module. Having them saved in the buffer can become very handy during long programs data entry.

But there's more: like it was the case in the **XEQ+** "navigator" mode, sub-functions found using **XEQ$** are also included in the LAST-7 buffer as mentioned before.

Caveat Emptor: Note that the latest revisions of the modules listed in Table-1 are needed for the Universal Execute to work with sub-functions. Older revisions will trigger the "NO MATCH" message, but other than that shouldn't cause any harmful disruptions to the system.

Sub-function access.

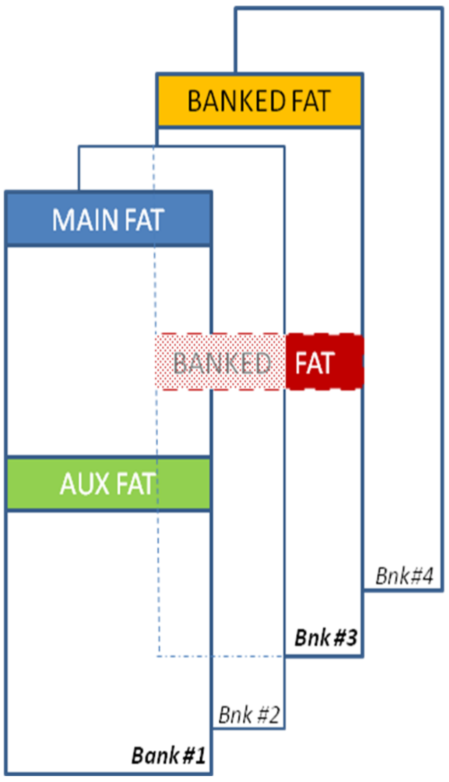
The way sub-functions are accessed depends on whether they're being entered in a program or used directly in manual mode.

- In PRGM mode the sub-function needs two program steps, the first one with the corresponding *sub-function launcher* and a second one with the *index in the auxiliary* FAT. It's therefore up to the **XEQ+** facility to identify the launcher (in the main bank of the current page) and figure out its corresponding index within the Auxiliary FAT.

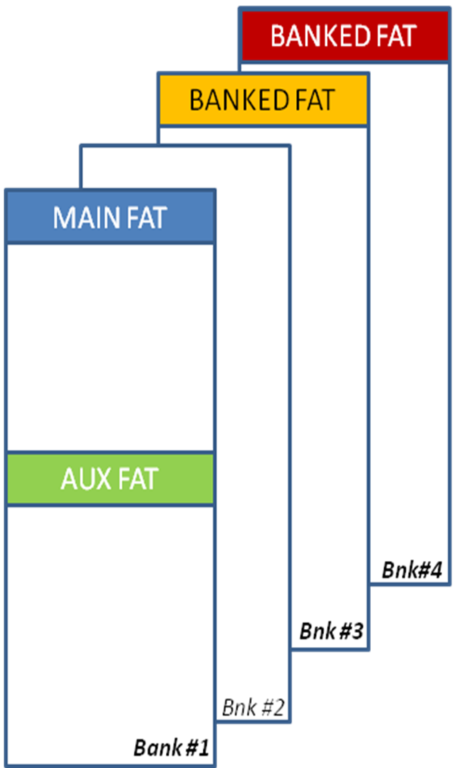
This works flawlessly even if there are two Auxiliary FATs in different banks, like it happens in the PowerCL_Extreme and the HEPAX_4H modules (see diagrams below) – automatically selecting the appropriate of the two launchers. This is a very robust implementation, and the program steps entered will work as long as the module is plugged in the calculator - regardless of which **page**. Not bad, if you think about it.

- In the manual case the **XEQ+** facility will simply send the program pointer to the address where the code for the called function starts, be that in a main bank or in a banked-switched one (circumstance that will require activating the target bank previously too). This will start the execution of the sub-function.

This case is not as fool-proof; however, consider for example that you access the sub-function **BANKED** in the PowerCL_Extreme with the module plugged in page #7. As explained before, the sub-function current address will be stored in the LAST-7 buffer for ulterior access via **LASTF**, but let's say you relocate the PowerCL module to a different page in-between, and then access the LAST-7 engine: what will be the consequence? The old (wrong) address is stored and will potentially play some havoc. Not a very likely scenario though, but it's not totally impossible and therefore it's good to be aware of it.



HEPAX_4H



POWER_CL

Note for MCODErs.

The table below shows the information stored in the 75 register fields depending on the type of function. This information is stored there when executing the function using either **XEQ+** or **XEQ$,** and will later be processed by the **LASTF^** facility to show the (sub)-function name and XROM data upon request.

Type	C[MS]	C[M]	C[S&X]
O/S Mainframe	"0"	"- - - - - - **1**ADR"	"- - -"
XROM Main Function	"0"	"- - - - - - FADR"	"CDE"
Sub-Function	bk# (0-3)	"- - - - - - FADR"	"000"

Remarks:

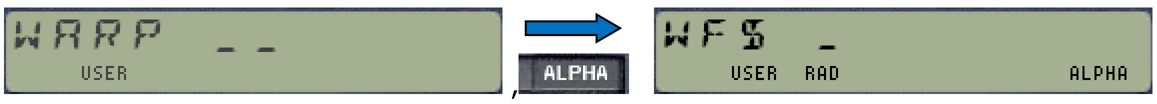
- "FADR" is the function's FAT entry address, and not the function's execution address – which could be obtained from "FADR" calling the [GTADR5] routine in the Library#4. Here the "F" character represents the page# (or "1" for the O/S mainframe functions), and the "A" character is either zero (for main FATs and banked FATs atop the page) or "8" for Aux FATs, in whichever bank.

- "CDE" are the three rightmost characters of the function HEX code, which is obtained from "FADR" calling the [FNCODE] routine located at 0xp6EA. Having this is very valuable when it comes to executing the function: we'll call [RAK70] in the Library#4, which works even if the plug-in module containing the function were to be relocated between the initial XEQ$ action and the re-call via LAST-7.

You may wonder why the information for sub-functions stored in the LAST-7 buffer is the FAT address, instead of the combination of its launcher code plus the index#. After all, such alternative is used in PRGM mode, so why couldn't it also be the method for manual mode? All that would be needed is to fill the A.X field with the index# (in hex) and send the program pointer to the launcher function itself, right?

That would certainly work if the implementation had followed the standard method defined in the O/S to prompt for the index parameter (using the upper bits of the function title chars)… but not such luck! As it turns out this is a self-inflicted problem because most of the sub-function launchers do *not* use said standard O/S method, but a custom one that mimics the same functionality but also allows for ALPHA key pressing – to switch to the launcher by name version – which isn't possible with the OS method.

For example using **WARP** in the Warp module, you can either enter an index number of press ALHA to switch to the text entry mode:



Note that ALPHA is automatically active when entering in the **WF$** prompt (this saves one keystroke). Note as well that pressing ALPHA again without any characters typed in *will use the current text in ALPHA instead.* – unless ALPHA is blank, in which case the second pressing will be ignored.

This may appear as a too subtle an enhancement to care for, but it is very useful in program mode in order to harmonize the standard and enhanced methods.

Overlays and Underlays.

The **XEQ+** mode is a new way to navigate the variable environment of the calculator that doesn't require you know the exact function spelling, nor that you do the actual typing of the letters – but it's much more than an alternative for machines with defective ALPHA key ;-)

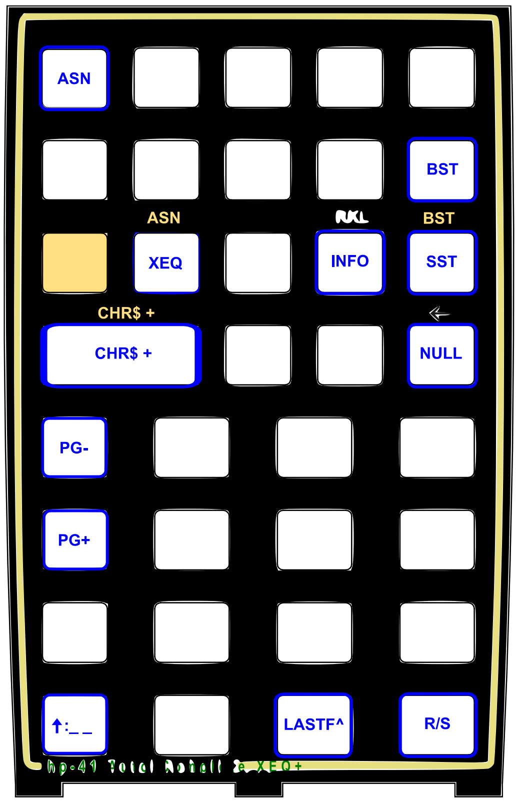
The picture below shows the available hot keys at different stages of the operation. Some are active at the initial "A:Z" prompt – like ^:_ _ for special character input; whilst others are applicable to the shown selection – such INFO, XEQ, and ASN.

Use the back-arrow key to restart the process or to cancel out to the OS.

The ALPHA key is used to trigger the "Universal Execute", **XEQ$**. Use it if you want to type the complete (sub)function name directly at the prompt, which enables ALPHA automatically.

The PRGM key is also active as a hot key to revert to the native XEQ function. Use it if you want to revert to the standard OS method to spell the function name in ALPHA mode, simply press PRGM, ALPHA and then spell the name as usual.

The operation is very dynamic and therefore not easy to describe with a static overlay. The best way to learn is by using it a few times. Seeing is believing; try it out and chances are soon it'll become one of your favorites. A real keeper!



Q-Note: The Q-register contents is overwritten during the Stack Comparison <u>in Manual mode</u>; therefore, they're meant to be used in running programs only.

3.- The Enhanced ASCII File Editor. { **ED+** }

Below is the article posted in the hp-forum describing the Enhanced Editor as a patch for the CX OS. Note that the version in the Warp_Core module is fully self-contained and thus **does not require the patch**, but the description is applicable to the implementation here, which is a tad more complicated than the patch for the CX because it uses a port-dependent scheme, as obviously the module could be plugged in any of the I/O external bus pages (8-F). This required changing the original ?NCXQ calls to three-byte calls, with the unpleasant consequence of losing the C-register as valid parameter-passing resource.

As a result, in the Warp_Core the code **uses the stack register "L" for scratch** – which means that every time you call **ED+** *the contents of the LastX register will be lost*. Make sure you make up for this in your FOCAL programs if needed.

41CX: Adding Lower Case & Special Chars to ASCII File Editor.

The standard ASCII file Editor in the 41CX has no support for lower case and other special characters. As a consequence, those chars need to be entered first in ALPHA and then manually transferred to the ASCII file using APPCHR or APPREC; either way the user needs to exit the editor, make the manual transfer, and call ED again.

With this patch entering lower-case and special characters is simply done by typing the designated key from within ED itself, no need for intermediate cumbersome steps.

The special chars keyboard layout is the same one available for ALPHA mode on the OS/X Module (and in turn on the original CCD Module) with USER off. There are two conceptual differences though:

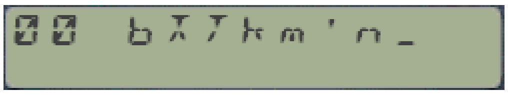
1. Since **ED** uses the USER key to move the cursor one position to the left, that key cannot be the mode flag in this case. Activation of the lower case & special chars is done ***switching ALPHA off*** instead.

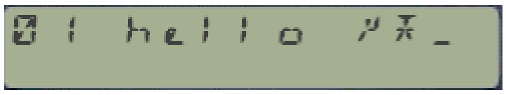
2. With ALPHA switched off, the "native" **ED** uses the numeric keys to enter digits, radix and the unary minus sign. That is not changed, and therefore imposes a design for the rest of available choices. This forces an inverted scheme for the layout compared to the OS/X, as follows:

- *Special characters* are in the same positions as in the OS/X but accessed using non-Shifted keys. The exceptions to this rule are the "little men" characters, which use letters [A], [B], [C], and [K] instead.

- *Lower-case letters* are accessed using SHIFTED keys - from SHIFT-A for "a" thru SHIFT-Z for "z". The only exception being "l" and "m", which use the non-shifted keys "L" and "M" (as LBL and GTO are reserved for the insertion mode and go-to-record functions within ED).

Easier to use it than to describe it - especially if you have the old CCD overlay at hand. The important thing is that none of the standard features or character layout in the original ED are altered in any way.





Patching the CX ROM.

All changes are confined within the bank-switched page of the CX-Extended Functions, i.e. ROM_5B. If you use the 41-CL or an emulator capable of altering the OS sector (like V41), then all you have to do is replace said ROM_5B with the new one containing the patch.

Patching instructions. Three steps are necessary:

Step #1. There are only three bytes to change in the original ED code, which is good news since that code is 1,001 bytes long!. The bytes to change are located at 0x5F62, 0x5EF0 and 0x5EF1; where:

0x5F62 has a jump-if-carry to address 5F51 (37F JC - 17d). The jump distance needs to be changed to point at 5F4C instead, i.e.

```
5F62    357     JC -22d
```

0x5EF0/F1 has a call to [BLINK] (265 , 020) - this needs to be replaced with a non-conditional jump to address 0x5BF1:

```
5EF0    3C5     ?NC GO
5EF1    16E     ->5BF1
```

Step #2. Next we need to add the following code at the jumped-to location (which is conveniently empty in the original ROM), to process the key-presses and triage them accordingly:

Code:
```
5BDF    1B0     POPADR    get calling address
5BE0    170     PUSHADR   keep it in RTN stack
5BE1    03C     RCR 3     move pg# to C<3>
5BE2    0A6     A<>C S&X  get absolute TBL adr
5BE3    1BC     RCR 11    rotate to ADR field
5BE4    066     A<>B S&X  put reference in A[S&X]
5BE5    11A     A=C M     preserve this address
5BE6    330     FETCH S&X read KEYCODE
5BE7    2E6     ?C#0 S&X  value non-zero?
5BE8    14D     ?NC GO    NO,  Skip one line and RTN
5BE9    032     ->0C53    [SKIP1]
5BEA    23A     C=C+1 M   add offset until
5BEB    366     ?A#C S&X  are they different?
5BEC    01B     JNC +03   no, exit loop
5BED    23A     C=C+1 M   next addr field
5BEE    3BB     JNC -09   loop back
5BEF    330     FETCH S&X get func. address
5BF0    3E0     RTN       and return
5BF1    066     A<>B S&X  sought-for value
5BF2    130     LDI S&X   beginning of table
5BF3    3F9     CON:      [LWRCAS]
5BF4    1F6     C=C+C XS  "6F9"
5BF5    106     A=C S&X   start of table
5BF6    37D     ?NC XQ    search ADDR in table
5BF7    16C     ->5BDF    [SRCHR1]
5BF8    02B     JNC +05   [GOTCHA]
5BF9    265     ?NC XQ    blink screen
5BFA    020     ->0899    [BLINK]
5BFB    3C9     ?NC GO    return to main code
5BFC    17A     ->5EF2    [CURSR2]
```

```
5BFD    106      A=C S&X   replaced char#
5BFE    075      ?NC GO    take over from here
5BFF    17A      ->5E1D    [VALID1]
```

__Step #3.__ The code above relies on a character table that needs to be added to the ROM. We do this in another empty section, not to disturb any existing code - as follows:

Code:

```
56F9    041    shift-"A"                     5726    077    "w"
56FA    061    "a"                           5727    058    shift-"6"
56FB    042    shift-"B"                     5728    078    "x"
56FC    062    "b"                           5729    059    shift-"Y"
56FD    043    shift-"C"                     572A    079    "y"
56FE    063    "c"                           572B    05A    shift-"1"
56FF    044    shift-"D"                     572C    07A    "z"
5700    064    "d"                           572D    03D    shift-"2"
5701    045    shift-"E"                     572E    10C    "m"
5702    065    "e"                           572F    03F    shift-"3"
5703    046    shift-"F"                     5730    021    "|"
5704    066    "f"                           5731    020    shift-"0"
5705    047    shift-"G"                     5732    101    "pi"
5706    067    "g"                           5733    064    "D"
5707    048    shift-"H "                    5734    05B    "["
5708    068    "h"                           5735    065    "E"
5709    049    shift-"I "                    5736    05D    "]"
570A    069    "i"                           5737    07E    "F"
570B    04A    shift-"J "                    5738    01F    "spat"
570C    06A    "j"                           5739    025    "G"
570D    04B    shift-"K"                     573A    040    "@"
570E    06B    "k"                           573B    01D    "H"
570F    04C    "L"                           573C    023    "#"
5710    06C    "l"                           573D    03C    "I"
5711    04D    "M"                           573E    028    "("
5712    06D    "m"                           573F    03E    "J"
5713    04E    shift-"N"                     5740    029    ")"
5714    06E    "n"                           5741    05E    "N"
5715    04F    shift-"O"                     5742    027    " ' "
5716    06F    "o"                           5743    024    "P"
5717    050    shift-"P"                     5744    022    " " "
5718    070    "p"                           5745    02D    "-"
5719    051    shift-"Q"                     5746    05F    "_"
571A    071    "q"                           5747    02B    "+"
571B    052    shift-"7"                     5748    026    "&"
571C    072    "r"                           5749    02A    "*"
571D    053    shift-"8"                     574A    060    "t"
571E    073    "s"                           574B    02F    "/"
571F    054    shift-9"                      574C    05C    "\"
5720    074    "t"                           574D    02C    shift-radix
5721    055    shift-"U"                     574E    03B    ";"
5722    075    "u"                           574F    03F    shift-"?"
5723    056    shift-4"                      5750    021    "|"
5724    076    "v"                           5751    03A    shift-"/"
5725    057    shift-"5"                     5752    100    upper "_"
5753    000    <end of table>
```

That's all there's to it folks - enjoy your enhanced ED+ !

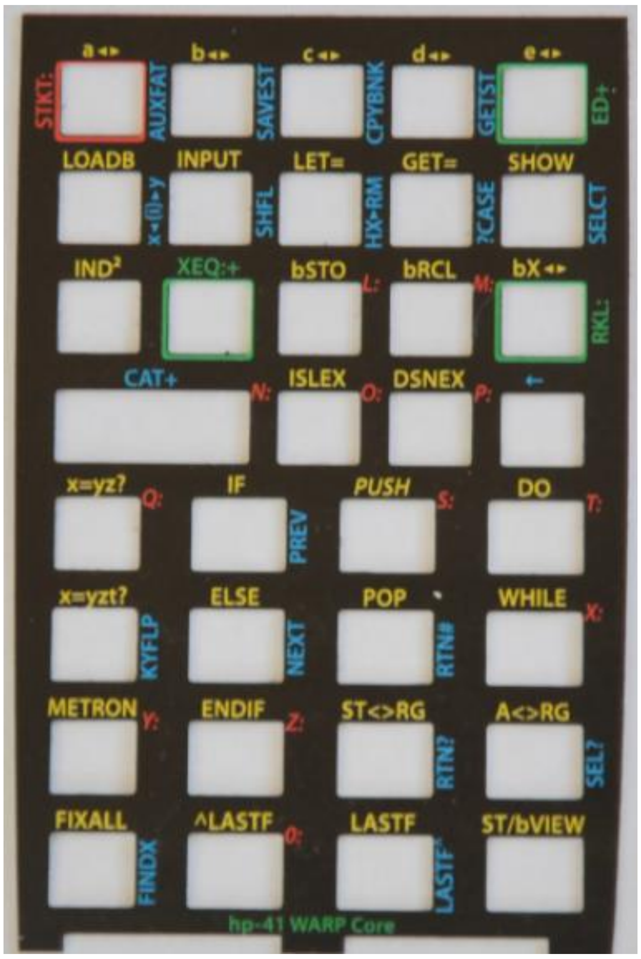
PS. With this enhancement, it's possible to enter any formula expression used by the Formula Evaluation module directly in an ASCII record. Refer to the following for details:
http://www.hpmuseum.org/forum/thread-862...evaluation

PPS. To *visualize* the lower-case letters in the LCD you need to use a half-nut machine, and also apply the patch provided by JF-Garnier in the following link:
http://www.hpmuseum.org/cgi-sys/cgiwrap/...?read=1205

WARP Top-Level Overlay.

Besides the one for the **XEQ+** facility, the WARP module also has a top-level overlay, which obviously includes an entry for the enhanced Text Editor **ED+**, the SELCT/CASE functions, the General Stack Comparisons facility **-STKT**, as well as many other functions and sub-functions from the module. All of these will be described in the following sections of the manual.

This overlay is somehow different from the standard concept in that it also fosters a few functions from the Formula Evaluation module. Why is that? Because combining these two modules makes a lot of sense from the programmability and synergy standpoint, really taking the 41 environments to new realms.



The functions from the Formula Evaluation Module are as follows:

- **IF**, **ELSE**, **ENDIF** ; evaluated on formula expressions in ALPHA – entered with **^FRMLA**
- **DO**, **WHILE** ; evaluated on formula expressions in ALPHA – entered with **^FRMLA**
- **LET=**, **GET=**, **SHOW** ; for direct assignment of variables to the Shadow buffer registers (indeed very similar to **bSTO**, **bRCL,** and **bVIEW** in this module, but featuring one additional buffer register).

It comes without saying that clicking on these functions without the Formula_Eval module plugged in will only show the corresponding XROM codes, but no actual execution will take place. You can however use them to enter them in a user program, of course.

4.- Continuous SST and BST. { **SST+** }

And to finalize the 'What's New' section let's review the latest addition to the module: a continuous mode for the Single-Step and Back-Step functions, repeating the execution while the function key remains depressed (auto-repeat operation).

This operation requires assigning the **SST+** function **to two keys**, one direct for the SST operation and another shifted for the BST case. Obvious candidates are of course the BST and SST keys on the keyboard, since the perfect mnemonics are already there and the continuous mode replaces the original (which is still accessible if needed simply by leaving the USER mode). This design only needs one XROM entry in the ROM's FAT, which is always a good thing.

Note that when using the shifted variant (for continuous BST), the SHIFT annunciator will stay on after you release the assigned key – and you'll need to press the SHIFT key to deactivate it (or move on pressing a shifted key if that's what you want of course).

The auto-repeat SST/BST mode is only active during program editing and review - i.e. when the PRGM annunciator is on. No operation occurs if PRGM is off, showing the "NULL" message on the display after a short while. Finally, like the native counterparts, the **SST+** function is not programmable. (Yes, you can force it into a program by pressing the assigned key *without the module plugged in*, but that won't do you any good so… don't!).

Implementation details

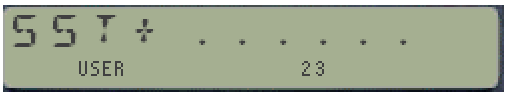
SST+ makes use of a poorly documented feature of the 41 O/S called "*immediate execution*" (IE), which takes advantage of the constant monitoring done by keyboard parsing routines looking for depressed keys. If these routines encounter an IE function, it transfers the execution to it using a shortcut to process it immediately without doing any of the usual between-instruction handlings. This trick effectively provides an auto-repeat operation of the IE function while the key remains depressed.

The program line enumeration speed is not adjustable. It is set to a convenient response for a quick advance or backtrack within the program code, but it's not meant to be used for code review purposes (too fast for that). On the 41-CL the function makes an additional pause if TURBO mode is selected.

To make room for **SST+** the functions **WF#** and **WF$** have been consolidated into one ("WARP") that *accepts both numeric and alphabetical inputs*. So, like the native XEQ function, at the **WARP _ _** prompt you can either enter a numeric value for the sub-function index, or press ALPHA followed by the sub-function name. Note that **WF$** remains available as a sub-function as well.

Credits

SST+ is based on **SST^** and **BST^** functions written by Nelson F. Crowle and originally available in the NFCROM. The MCODE is much shorter than other continuous SST implementations, such as **CSST** written by Phil Tri – available in other modules and formerly included in the WARP module as well but now replaced by this simpler and easier-to-use implementation.



Note that SST+ does not work on the DM-41X because the software emulation implemented on that machine doesn't have support for the immediate execution functionality.

MCODE Listing for SST+

Header	A412	09E	"^"	
Header	A413	014	"T"	Auto-repeat SST
Header	A414	013	"S"	
Header	A415	013	"S"	Nelson Crowle
SST^	A416	000	NOP	NOT programmable
	A417	000	NOP	and immediate (!)
	A418	1FD	?NC XQ	wait a little - CL compatible
TH keys	A419	12C	->4B7F	[WAIT4L] - Enables RAM
)	A41A	104	CLRF 8	
	A41B	3B8	READ 14(d)	see if SHIFT is ON
	A41C	23C	RCR 2	move right 8 "flags"
	A41D	3D8	C<>ST XP	commit to status
	A41E	38C	?FSET 0	was UF 47 set?
	A41F	013	JNC +02	no, skip
	A420	108	SETF 8	yes, -> [BST]
SST^	A421	3D8	C<>ST XP	restore status
	A422	388	SETF 0	bearer of the news??
	A423	3B8	READ 14(d)	get user flags
	A424	17C	RCR 6	move
	A425	3D8	C<>ST XP	will set UF 31 later on
	A426	30C	?FSET 1	key pressed??
	A427	360	?C RTN	no, end here
	A428	375	?NC XQ	
	A429	088	->22DD	[SSTBST]
	A42A	10C	?FSET 8	was UF 47 set?
	A42B	0CF	JC +25d	yes, -> [BST]
SST^	A42C	141	?NC XQ	
	A42D	0A4	->2950	[GETPC]
	A42E	3F8	READ 15(e)	get current LineNum
	A42F	00C	?FSET 3	PRGM editing?
	A430	083	JNC +16d	no, -> LB_AE7B
	A431	2E6	?C#0 S&X	
	A432	3DD	?C XQ	
	A433	0A9	->2AF7	[NXLSST]
	A434	065	?NC XQ	
	A435	050	->1419	[GETLIN]
	A436	14C	?FSET 6	
	A437	013	JNC +02	LB_AE74
	A438	046	C=0 S&X	
	A439	226	C=C+1 S&X	
	A43A	013	JNC +02	LB_AE77
	A43B	266	C=C-1 S&X	
	A43C	3E8	WRIT 15(e)	Update LineNum
	A43D	0DD	?NC XQ	
	A43E	08C	->2337	[PUTPC]
EXIT	A43F	248	SETF 9	
EXIT2	A440	171	?NC XQ	Update Ann's
	A441	01C	->075C	[ANNOUT] - leaves RAM selected
	A442	189	?NC GO	
	A443	016	->0562	[DFRST8]
BST	A444	379	?NC XQ	
	A445	0A0	->28DE	[BSTEP]
	A446	39D	?NC XQ	TOGGLE SHIFT FLAG
	A447	07C	->1FE7	[TGSHF1]
	A448	3BB	JNC -09	[EXIT]

General Introduction – 'Dare to Compare'.

Welcome to unexplored territories, a journey taking the venerable hp-41 platform to places it probably hasn't been before: meet the "*Dare 2 Compare*" version of the Total_Rekall module, with the following new bells & whistles:

- Enhanced launchers and function prompts that interact with one another and are "aware" of previous choices. Refer to the sketch in previous pages for details.

- Added a secondary FAT with 112 sub-functions, amongst them all test functions on the stack registers {M-Q} – to complement the {T to L} set implemented as main functions).

- Automatic entering for main functions of non-merged arguments as second program lines. For instance: **Z<=T?** . This feature was a must, after I learned how to do it while developing the CLXPREGS module.

- For sub-functions, a *triple-non-merged argument scheme* using three program steps. For instance: **M>= IND Z?,** whereby only the third parameter is entered manually.

- Added functions **SELCT** and **?CASE** – a pseudo SLECT-CASE implementation that allows comparison of any "variable" (i.e. register, including the stack and indirect) defined by SELCT and stored in the buffer - with a hard value (integer) entered at the ?CASE prompt.

- New direct register exchange (not using the stack) between the register selected by **SELCT** and the target chosen by **S<>**, also supporting indirect, stack and combination of both. Features housekeeping utilities like **NEXT**, **PREV**, and **SEL?** to show, increment and decrement the selected register variable. Useful for program algorithms to save explicit re-selections.

- Direct comparison to zero for any register (direct, indirect, stack), with the "Zero-group" functions. For instance: **?0# 23 ,** or: **?0>= IND 08.** Also includes the sub-function **0<>** for a clearing register option not altering the stack.

- Implements the "emergency storage buffer" with six data registers in case you run out of regular ones. You can store, recall, view, and Exchange the buffer registers with the X register at any time. Also, you can use this buffer with functions **PUSHRTN** and **POPRTN** to extend the RTN stack length.

- An all-new stack shuffle function **SHFL**, that allows altering the five main stack registers XYZTL according to a register pattern entered as a five-field prompt in manual mode, or in an ALPHA string during program execution. Selective register clearing is also possible using zero as the register description in the strings.

- New functions to search for Auxiliary FATs (**AUXFAT**) and their corresponding launchers (**SFLNCH**) – help you manage the advanced features in the system.

Very tricky stuff, and not simple to make it all tick at unison - but the results are nothing short of amazing if I may say it. Reading this manual should help you digest the new functionality and apply it to practical examples as well.

Note: To make all these additions and enhancements possible it was needed to remove the UMS (Unit Management System) from the earlier versions of the Total_Rekall module. The UMS with Constants Library is available in the PowerCL and PowerCl_Extreme modules. The UMS without the constants library is also available in the dedicated "UMS Module" for those of you without a 41CL (say what? a temporary situation hopefully…)

The Sub-Function Catalog. { **CAT+** }

CAT+ provides usability enhancements for admin and housekeeping. It invokes the sub-function CATALOG; with *hot-keys for individual function launch and general navigation*. Users of the POWERCL Module will already be familiar with its features, as it's exactly the same code – which in fact resides in the Library#4 and it's reused by other modules, like the 41Z, SandMath, and SandMatrix as well.

The hot-keys and their actions are listed below:

[**R/S**]:	halts the enumeration
[**SST/BST**]:	moves the listing one function up/down
[SHIFT]:	sets the direction of the listing forwards/backwards
[**XEQ**]:	direct execution of the listed function – or entered in a program line
[**ENTER^**]:	moves to the next/previous section depending on SHIFT status
[**<-**]:	back-arrow cancels the catalog

One limitation of the sub-functions scheme that you'll soon realize is that contrary to the main functions, *they cannot be assigned to a key for the USER keyboard*. Typing the full name with **WF$** _ (or entering its index at the **WARP** _ _ _ prompts) is always required. This can become annoying if you want to repeatedly execute a given sub-function. The **LAST Function** implementation described below certainly minimizes this issue for repeat executions of the last sub-function called, without a dedicated key assignment required.

Launchers and Last Function functionality. { **WARP** , **WF$** }

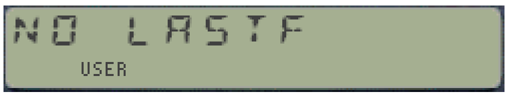
This module includes full support for the "LASTF" functionality. This is a handy choice for repeat executions of the same function (i.e. to execute again the last-executed function), without having to type its name or navigate the different launchers to access it. The implementation is not universal – it only covers functions invoked using the dedicated launchers, but not those called using the mainframe XEQ function. The following table summarizes the launchers that include this feature:

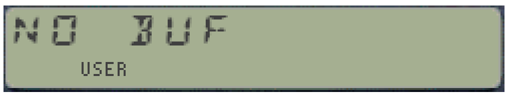
Module	Launchers	LASTF Method
WARP Core	**-STKT _**	Captures (sub)fnc id#
	RKL _ _	Captures (sub)fnc id#
	WF$ _	Captures fnc NAME
	WARP _ _ _	Captures (sub)fnc id#
	CAT+ (XEQ')	Captures (sub)fnc id#

LASTF Operating Instructions

The Last Function feature is triggered by pressing the radix key (decimal point - the same key used by LastX) at the "ST: " prompt. When this feature is invoked, it first shows "LASTF" briefly in the display, quickly followed by the last-function name. Keeping the key depressed for a while shows "NULL" and cancels the action. In RUN mode the function is executed, and in PRGM mode it's added as a program step if programmable, or directly executed if not programmable.

If no last-function record yet exists, the error message "NO LASTF" is shown. If the buffer #9 (used to store the last function id# code) is not present, the error message is "NO BUF" instead.


```
NO LASTF
USER
```

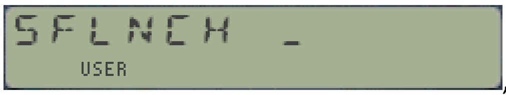

```
NO BUF
USER
```

Searching for Auxiliary FATs. { AUXFAT , SFLNCH }

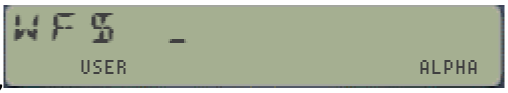
With the spread of advanced modules, it's become challenging to know how many of them have auxiliary FAT's holding sub-functions. To assist on this subject, the WARP_Core adds two new functions as described below.

Sub-function Launcher-launcher (no typo).

SFLNCH will scan the page entered at the prompt for Auxiliary FAT. If one is found, the corresponding sub-function launcher will be launched, offering the user to type the sub-function name. For example, for the Warp_Core itself:


SFLNCH _
USER

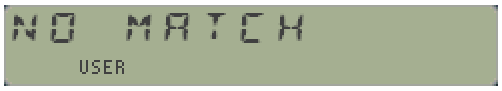
, => ,


WF$ _
USER ALPHA

The input will be restricted from #6 to #F, as those are the only pages that may have a secondary FAT. Typing any other character will simply be ignored by the function, and the prompt will persist. If there's no module plugged in the chosen page, or if the ROM has no FAT you'll get the usual error messages "NO ROM" or "NO FAT" correspondingly.

The search starts at the top of the page, looking for code structure common to all sub-function facilities, involving the consecutive presence of several MCODE instructions. Note that depending on the actual location of those instructions within the 4k page the search time may be long.

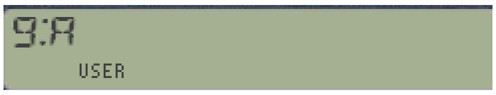
When the sub-function launcher code is found the function will transfer the execution to it, presenting the ALPHA prompt for the sub-function name spelling. If no launcher code is found, the function will show a "NO MATCH" message.


NO MATCH
USER

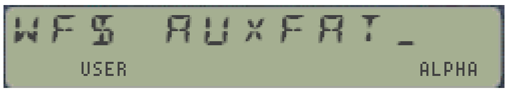
Enumeration of Pages with Secondary FATs.

AUXFAT will scan the calculator bus looking for auxiliary FATs in all the pages, starting with pg# 6. A list will be compiled and presented when the scanning has completed (i.e. all pages until pg# F have been searched).

For example, with the WARP_Core and the Formula_EVAL modules plugged in, the function returns the following result (which is helpful to find out on which pages are meaningful for **SFLNCH**):


9:A
USER

AUXFAT will "see" the secondary FATs from the PowerCL and the HEPAX_4H modules, even though they are is in a bank-switched page. It will not however see the original HEPAX secondary FATs. Note also that **AUXFAT** is itself a sub-function, and therefore needs to be called using **WF$** (or **WF#** with index #045)

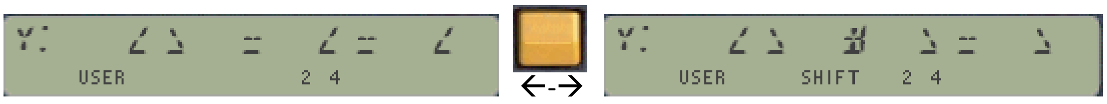

WF$ AUXFAT_
USER ALPHA

The main Function Launcher **{ –STKT }**

Considering the number and nature of the functions included in this module it isn't surprising that the launcher method has been once again the chosen approach. You can access any of the stack swaps and test functions with a few keystrokes using a single function, the "Function Builder" -.

The driving parameter for the function is the stack register, thus the expected input at the "ST _" main prompt is to be the corresponding stack register letter {X, Y, Z, T, L, M, N, O, P, Q,} – which will be placed on the left side of the display in a second prompt to chose the specific action to perform.

Once the stack register is chosen, the second prompt offers a selection of options in a menu-like fashion with two screens toggled by the SHIFT key to fit the seven choices available:



Once the individual register is selected, a common feature in all functions is that the prompt accepts IND _ _ , and ST _ arguments using the SHIFT and RADIX keys as with the native OS implementation. The combined IND ST _ is also allowed of course.

Dynamic Register Update: the "NEXT" choice.

Pressing the [SST] key *will update the function builder main prompt*; changing the source register sequentially in a cyclic sequence each time [SST] is pressed. This saves time and keystrokes, making it easier to use in spite of its comprehensive functionality. Note also that pressing the back-arrow will revert back to the main prompt, requesting a register to start the process.

Where are the upper status registers? {"a" to "e"}

All 16 stack register swaps are available, either as main functions or in the auxiliary FAT as sub-functions. This is the case of the upper stack registers {a-e}, that can be accessed directly from the main launcher pressing the corresponding top-row key. Just be careful with these!!

Because of their relative small practical application, the tests of the upper status registers were replaced by the Zero-testing set, You can still use them as the second argument at the stack addressing prompt, for instance you could do: T<> a, or: Z<> c if wanted.

Special Guest "Zero"

In addition to the 10 stack registers mentioned before you can also enter zero "0" at the main "ST_" prompt to invoke the Zero-comparison test function – so considered it to be the invited guest to the stack for these purposes. Note this is not Data Register R00, but the value "0" for the comparison. Note also that *swapping with "zero" brings the current reg. value to X besides clearing the chosen register.*

Reversed RPN Logic?

Contrary to the standard native functions on the 41 OS, all the individual test comparison functions feature the question mark at the beginning of its name. This is just a nomenclature choice but has no bearing on the actual operation of the functions. In a program the same "Skip line if False" rule applies if the test result is not true, whereas in manual mode the "YES'/'NO" messages will be triggered for the True/False cases as usual.

The "Total_Rekall" Dilemma . { RKL }

One of the obvious shortcomings of the HP-41 OS is the lack of RCL math functions: even if they are less necessary than the STO math and perhaps easily replaced by combination of other standard functions, it is a sore omission that has been the previous subject of different implementation attempts to close that gap.

The first component is naturally the addition of individual RCL math functions, like **RC+**, **RC-**, **RC***, **RC/** and **RC^**(the bonus one). These can be written without much difficulty, even supporting INDirect register addressing, but with two major restrictions:

1. Operating in manual mode only, and
2. Excluding the Stack registers from the register sources.

The first limitation can be overcome using the non-merged function approach, whereby the argument of the function in a program is given in the next program line following it. This is stack-neutral so doesn't interfere with the intermediate calculations.

To solve the stack addressing one needs to resort to heavier wizardry, basically writing extra code to replace the OS handling of the prompting in these functions – which is based on the PTEMP bits of the function name. *The custom prompting is therefore completely under the control of the function, and not facilitated by the OS*. It is arguably a small net benefit compared to the required effort, but as the only remaining challenge it was well worth tackling down.

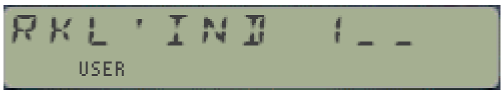
Once the technique was developed it was relatively easy to apply to other functions, like the stack exchange and comparison tests – if you can you envision instructions like: "Y<> IND M", or "Z<=N?" to give just two examples. Unfortunately, the Library#4 was already full, so the subroutines are only available on this module.

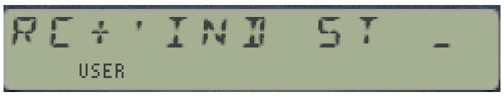
RCL Math on steroids: The Extended RKL Launcher.

In addition to the four "standard" arithmetic operations this module includes **RC^**, for the Recall Power function – which will calculate the REG-th. power of the value in X, i.e. X= e^(RG# * ln X).

The other additional case is **AIRCL**, which will append to ALPHA the integer part of the value stored in the data register. It also supports the stack and indirect values, such as IND ST X.

All RCL functions feature a *prompt lengthener* to directly access registers in the 100-111 range. You can activate this by pressing the EEX key at any of their prompts. Note that from 112 and up you'll be either accessing Stack registers or INDirect addresses, as shown in the next pages (see table 1.1)





In terms of usability, note that you can switch amongst the five RCL math functions pressing the corresponding arithmetic key at their prompt. You can also revert back to the **RKL** function simply pressing the [SST] key twice during any of their prompts (this toggles between the RKL group and the main launcher described in the following section).

To save program bytes, **RKL** will automatically revert to the standard RCL *when entered as a program step*. Lastly, you can manually revert to the native RCL pressing the [RCL] key again at its prompt. When you do this in program mode the standard OS is used for efficient line entering of the standard cases, i.e. RCL 27 in a single program step as opposed to using the non-merged approach. More on this subject later on.

Programmability: arguments Look-up Table

All functions and sub-functions are fully programmable. When entered into a program *the argument will be automatically entered as a second program line after the main function*. This line will not be executed; rather the function will read the value during the program execution. Note also that this works seamlessly for direct data registers up to R111, with *no need for manual adjustment for extended range, INDirect and Stack register arguments* (refer to the table below for details).

For INDirect registers 80 Hex (or 128 dec) is *automatically added* to the register number.

Examples:	Z<> IND 25	=>	Z<>	followed by 152
	RC/ IND 16	=>	RC/	followed by 144

For Stack arguments 70 Hex (or 112 dec) is *automatically added* to the "Stack index" number.

Examples:	Z<> T	=>	Z<>	followed by 112 (T index = 0)
	RC+ Y	=>	RC+	followed by 114 (Y index = 2)

For combined INDirect Stack arguments, F0 hex (or 240 dec) is *automatically added* to the stack index, or 240 decimal

Examples:	Z<> IND Z	->	Z<>	followed by 241
	RC* IND M	=>	RC*	followed by 243

The table below shows the transition zones graphically:

Argument	Shown as:	Argument	Shown as:	Argument	Shown as:
100	00	112	T	124	b
101	01	113	Z	125	c
102	A	114	Y	126	d
103	B	115	X	127	e
104	C	116	L	128	IND 00
105	D	117	M	129	IND 01
106	E	118	N	130	IND 02
107	F	119	O	131	IND 03
108	G	120	P	132	IND 04
109	H	121	Q	133	IND 05
110	I	122	|-	134	IND 06
111	J	123	a	…	…

Table 1: Register index mapping.

A few exceptions to the rule.

A couple of functions in the module do not allow stack arguments in their prompts. These functions are **A<>RG** and **ST<>RG**. You can use any register number and INDirect addressing but not Stack registers as the destination – neither the combination IND ST even if it is possible to invoke it. These functions use the standard method provided by the OS to build the prompts, which as it was mentioned before lacks the complete flexibility offered by the newer functions.

Warning: Be aware that the merged lined will not be automatically created for these two functions. If you enter them in a program, you must add the argument manually as an additional program step.

Direct Register Comparisons.

A fact that may be easily overlooked is that besides doing intra-stack register comparisons, these functions also allow direct comparison of any of the main stack registers with any data register in RAM. Furthermore, the Zero group allows direct comparison with zero on any data register as well, not just the stack.

This provides more flexible programming choices, saving programming steps and keeping the stack unaltered as there's no need to bring the register content to X/Y in order to make the comparisons.

Some examples:

?X< 13	=> is R13 > X?	?0# 05	=> is R05 different from zero?
?T>= 16	=> is R16 <= T?	?0> 11	=> is R11 less than zero?

Example: Armed with these new functions bubble-***sorting the stack*** is a fairly simple task – as long as you remember that *multi-line functions cannot be directly placed after a test*:

```
01  LBL "STSRT        08  ?Z<= (T)          15  GTO 00
02  X>Y?              09  GTO 00            16  Y<> (Z)
03  X<>Y              10  Z<> (T)           17  LBL 00
04  ?Y<= (Z)          11  LBL 00            18  X>Y?
05  GTO 00            12  X>Y?              19  X<>Y
06  Y<> (Z)           13  X<>Y              20  END
07  LBL 00            14  ?Y<= (Z)
```

Be aware that in program mode the function arguments will be automatically added as non-merged steps – this will be described in the following pages.

Stack Exchange vs. Test Functions

There is no fundamental difference in the eligible stack registers for exchange functionality vs. direct comparisons. All the status registers except the "lazy-T" }-(10) have the same set, although some functions are in the main FAT, and some others are in the Auxiliary FAT. This is again due to the limited number of entries in the FAT, which imposed some selection between registers, based on likely importance and usability.

In terms of functionality, the table below shows the available choices for a direct approach, and which ones are only available indirectly, as a second argument of the particular function.

Register	Exchange	Tests
X	Main	Main
Y	Main	Main
Z	Main	Main
T	Main	Main
L	Main	Main
M	Main	Sub-fcn
N	Main	Sub-fcn
O	Main	Sub-fcn
P	Main	Sub-fcn

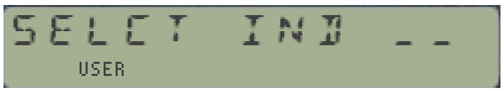
Register	Exchange	Tests
Q	Sub-fcn	Sub-fcn
|-	Sub-fcn	Indirect
a	Sub-fcn	Indirect
b	Sub-fcn	Indirect
c	Sub-fcn	Indirect
d	Sub-fcn	Indirect
e	Sub-fcn	Indirect
"0"	Sub-fcn	Main
Rnn	Main	Indirect

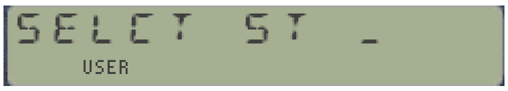
Lastly, non-stack Data Register swapping is missing from this set, but it's not forgotten - it's the subject of the next sections.

General-Purpose Comparison with SELCT / ?CASE

Perhaps the most versatile approach for register comparison is provided by the combination of functions **SLCT** and **?CASE.** With them you can test any register (chosen using SELCT) against a fixed integer value – which is provided as the argument for ?CASE.

The variable chosen by SELCT is stored in the header of buffer id#7 (the same one used for the "emergency storage" information). This may be a direct data register number, a stack register (adds 70 Hex), an indirect register (adds 80 Hex), or the combination of both (adds F0 Hex). Refer to the table in previous section for details. This is done automatically by the function, totally transparent to the user.



or:


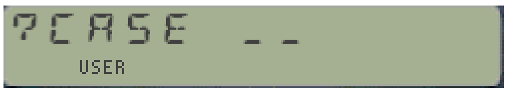
In program mode the variable for SELCT and the comparison value for ?CASE will be introduced as non-merged lines in program step following the main function – which is consistent with the other functions seen before that use the same schema. Note that comparison values are positive integers only.

If no variable has been selected previously, ?CASE will default to the X register (i.e. id# 73 Hex or 115 decimal – again no need for you to be concerned with that detail). Pressing [VIEW] at the SLCT prompt will show you the current variable stored in the buffer.

The variable will therefore continue to be in effect until another SELCT statement is used. This will allow you to make repeat comparisons without the need to have to recall the reference in every instance – and also without the need to have both the reference and the variable in the stack.

For example, to compare the value of data register R05 with the values 1,2,3 you'll use these instructions, which can be interspersed amongst all your program code (note that there's no need for an "END SELECT"-like instruction):

SELCT 05	loads the reference in buffer
?CASE 1	tests if R05=1?
Yes	
No	
…	
?CASE 2	tests if R05=2?
Yes	
No	
….	
?CASE 3	tests if R05=3?
Yes	
No	
…	



Note that the comparison value is directly provided in the prompt, and that a "by reference" comparison is not allowed (i.e. using a data register instead).

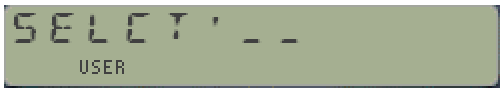
As the question mark would suggest, **?CASE** is a typical test function that will follow the "do if true / skip if false" rules when running in a program – or show the familiar "YES/NO" in manual mode.

*Remember not to place a non-merged function directly *after* a test function – doing so will create a problem as the OS does not recognize the non-merged steps as part of a single function!*

General-Purpose Exchange with SELCT / S<>

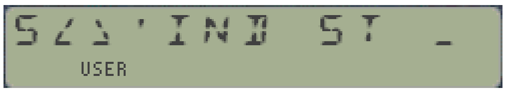
In a parallel implementation to the previous subject, you can also use the **SELCT** schema combined with the sub-function **S<>** to perform a data register exchange directly, i.e. with no need to bring either of their contents to the stack – which is so left undisturbed.

The advantages are clearly seen: the stack is not altered, and the same selected variable-register can be used for both case-equal comparison and register-exchanges. Both together offer possibilities to the smart FOCAL programmers, never too late to learn new tricks ;-)



```
SELCT'_ _
USER
```

Defines the selected variable-register



```
S<>'IND ST _
USER
```

Defines the target register to exchange.

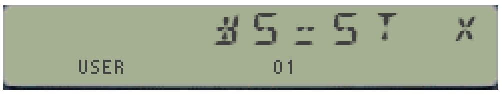
Like **SELCT** itself, **S<>** also supports indirect addresses, Stack addresses and combination of both – thus you could do flexible register exchanges, such as: IND ST M <> IND 34.

Here too the same table of parameters shown in figure-1 applies – refer to that table for details. Remember that the indirect reference will change if you alter the content of the register that holds the register pointer.

Showing and Recalling the selected variable.

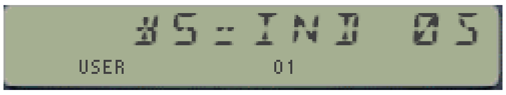
If you're not sure which is the selected variable you can press [R/S] at either of these function's prompts to invoke the **SEL?** Function – which recalls its value to the X-register, and *in manual mode also to the display.*

- **SEL?** shows the value currently selected. If no selection has been made the default value shown is the X register. Note that the selection of a variable does not require that the register exists at that point – the existence checks will be done when trying to access the contents of said register.



```
#S=ST   X
USER     01
```

or:



```
#S=IND 05
USER     01
```

Increasing and Decreasing the selected variable.

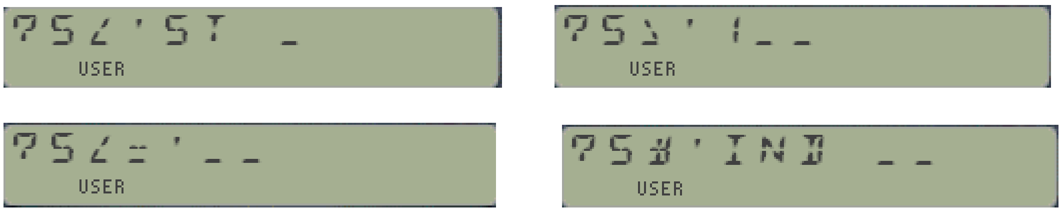
These sub-functions are related to the SEL# variable set by SELCT, as follows:

- **NEXT** and **PREV** increment and decrement the selected variable by one. No decrement will occur if the selection is R00. No changes will be made if no selection exists (which defaults to Stack "X"). These functions are very useful during program control for sequential access to different registers as selected variables. In manual mode the information message will show the new setting, as if **SEL?** had been invoked as well. Note that the stack will remain unchanged, i.e. *these functions don't recall the updated value to the X-register.*

Remark that NEXT/PREV have effect on the register number stored in the buffer header (i.e. the "S" variable), but not on the actual register contents. Also note that if an indirect or stack register is selected then the next/previous value is dictated by the "natural" register sequence, i.e. Stack_L comes after Stack_X, etc.

Value Comparison tests with selected variable.

Similarly, using the provided sub-functions you can compare the contents of the selected variable with any "target" register of your choice entered at the prompt. Like all tests functions in manual mode "YES/NO" is shown depending on the true/false condition; and in a running program one program line will be skipped when false, or when true it will continue with the line following the sub-function merged lines (which there'll be three of them as these are sub-functions!).



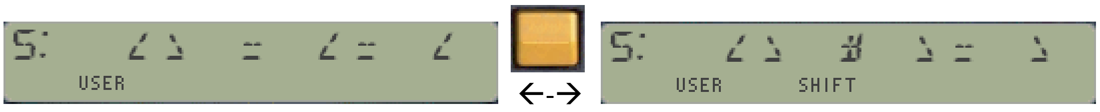
Note that the equal-to comparison **?S=** is different from **?CASE**; in both instances it is the content of the selected register what gets used as first value (i.e. "by reference") , but the second value differs: in the equal-to case it is the content of the target register being compared, whereas for ?CASE the comparison is against the value provided at the prompt (i.e. "by value").

Let's for example compare the contents of data registers R04 and R05. If we choose R05 as the selected variable, then R04 becomes the "target" to compare against, i.e. showing all the parameters as non-merged program steps:

```
01  SELCT (05)              01  SELCT (05)
02  5                       02  5
03  WF#                     03  WF#
04  40                      04  42
05  4          ?S< 04       05  4      ?S> 04
06  Yes                     06  yes
07  No                      07  no
```

The surrogate Stack Register "S".

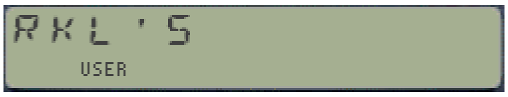
All of the variable comparison functions, as well as the exchange **S<>** and **?CASE** have been grouped under its own section within the main launcher **–STKT.** Either by pressing "S" or moving about the stack registers letters using [SST], the surrogate S-register screens offer the same functionality as the standard stack registers, as shown in the pictures below:

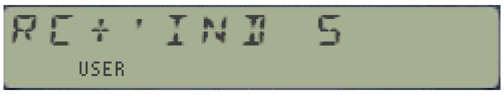


Note how this U/I has the same look & feel as the other stack registers. The fact that all the choices are sub-functions is completely transparent to the user – with the only exception of the need to manually add the parameter line in a program as described before in the manual.

Connecting "S" with RKL

We have just seen that even though "S" isn't a proper stack register it can certainly be handled as if it were. This metaphor has been extended (but not stretched) to include support for "S" as option of the RKL prompt, when the radix key is used for the stack registers. Thus, the contents of the current selected register can be recalled in this way – which also includes the IND addressing and the RKL math operations as well.

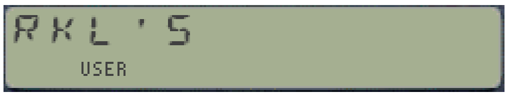


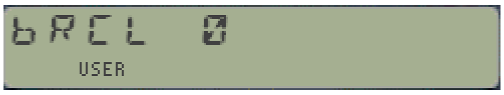
or:


Note however that in program mode the **RKL** instruction will be registered using the actual selected register number as parameter in the second line – not as a variable but as its actual value at the time when the instruction is entered in the program.

You can, however, use the sub-function **bRCL** instead (with parameter zero) – which will use the selected register in a running program, and thus it's completely equivalent to RKL "S" also in program mode. The caveat is the lack of IND and math operations in this case.

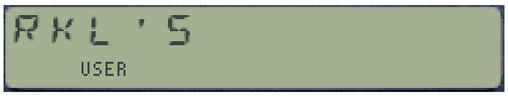
Using **bRCL** will be covered in a later section of the manual. For the time being just remember that, *both in manual and running program modes*:

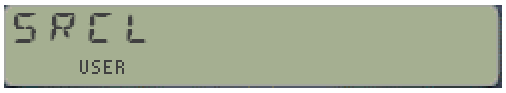


=


Storing, Recalling and Viewing the contents of "S"

You can always use the standard RCL, STO and VIEW instructions to recall, store and view the contents, but that requires knowing the value of the #SEL variable itself to use it as parameter. An easier way is also available with the sub-functions **SRCL**, **SSTO** and **SVIEW** - which don't need you to have such knowledge beforehand. Therefore, here's another equivalence for you:



=


The **SRCL**, **SSTO** and **SVIEW** sub-functions operate on the register which value is stored in #SEL.

Therefore these four keystroke sequences are equivalent:

01 SEL?	01 RKL "S"	01 SRCL	01 bRCL 0
02 RCL IND X	(not in a PRGM !)		

Note: If you prefer it, the "S" parameter designates a sort of indirect destination of the operation – as it'll use the data register which value is stored in #SEL variable. Yet it's also possible to use IND if you access it via RKL, like in " RKL IND S" – which could also be considered as a double indirection from a strict point of view.

Examples: Data Registers Bubble-Sort

The programs below show two practical examples of the new functions for data register sorting. Note the use of the non-merged program steps and the workaround required in the conditional tests to avoid jumping in-between non-merged lines. The second main label uses the control word bbb.eee in X to delimit the data registers range, whereas the first will use all the data registers currently available in the calculator.

```
01  *LBL "SRTALL"         all registers
02   SIZE?                Get current size
03   DSE X                get last reg index
04   E3                   format it
05   /
06  *LBL "SRTRGX"         bbb.eee in X
07  *LBL 01               main loop
08   ENTER^               push cnt'l word to Y
09   ENTER^               push it one more
10   SELCT  (IND Y)       select ind(bbb)
11   242
12   ISG X                bbb+1
13   GTO 00               skip until end is reached
14   RTN                  all done.
15  *LBL 00               inner loop
16   WF#   (?S>= IND X)   use the reverse test and a
17   43                   forced GTO to avoid jumping
18   243                  in between non-merged steps:
19   GTO 00               true, jump over
20   S<>   (IND X)        false, swap registers
21   243
22  *LBL 00
23   ISG Y
24   SELCT  (IND Y)       update selected register
25   242                  (cannot use NEXT !)
26   ISG X                update comparison register
27   GTO 00               repeat inner loop
28   X<> Z                recall control word
29   E-3                  decrease upper limit
30   -
31   GTO 01               repeat main loop
32   END                  end of program
```

Another approach for the all-registers case is shown below, using the **NEXT** instruction to update the selected register directly – as opposed to the indirect way in the previous example.

```
01  *LBL "SRTALL2"
02   SIZE?
03   DSE X
04   E3
05   /
06  *LBL 01
07   SELCT  (00)
08   ENTER^
09   ISG X
10   GTO 00
11   RTN
12  *LBL 00
13   WF#  (?S<= IND X)
14   41
15   243
16   GTO 00
17   S<>   (IND X)
18   243
19  *LBL 00
20   WF#   (NEXT)
21   45
22   ISG X
23   GTO 00
24   X<>Y
25   E-3
26   -
27   GTO 01
28   END
```

Tinkering with ISG and DSE: complement modes.

In the previous examples we have used the ISG function to increase the pointers to the data registers being compared. The code is a bit inefficient because the termination conditions are the opposite to the implemented in the standard ISG and DSE functions – i.e. here we loop while the condition is FALSE, which requires an additional GTO step to skip the RTN.

The complement functions are defined as follows:

- **ISLEX** "Increment X and Skip if Less or Equal", and
- **DSNEX** "Decrement X and Skip if Not Equal".

In both cases they only work on the X register, which is expected to have a control word in the form bbb.eee:ii , like the standard ISG and DSE. If the increment is not given (zero) the default value used is ii=1.

Using **ISLEX** instead of ISG X in the example programs will change the code to this:

```
06 *LBL "SRTRGX"           bbb.eee in X
07  *LBL 01                main loop
08   ENTER^                push cnt'l word to Y
09   ENTER^                push it one more
10   SELCT  IND Y  (242)   select ind(bbb)
11   ISLEX    (WF#  68)    bbb+1
13   RTN                   all done if (bbb+1) > eee
15  *LBL 00
16   …
```

And similarly, in SRTALL2:

```
06 *LBL 01
07   SELCT  (0)
08   ENTER^
09   WF#   (ISLEX)     bbb+1
10   68
11   RTN               all done if (bbb+1) > eee
12 *LBL 00
13   …
```

Another approach to deal with this contingency would have been using the **SKIP** function, available in some extension modules. When placed in the TRUE position it basically defeats the "do if true" rule, shifting the decision by one program step:

```
ISG X              ISG X              ISLEX
True               SKIP               (Un)True
False              False              (Not)False
…                  …                  …
```

Have we re-invented these wheels?

Certainly, there's some overlap between the new functions and the set included in the CX X-Functions, as shown on the table below:

CX-Function	X=NN?	X#NN?	X<NN?	X<=NN?	X>NN?	X>=NN?
WARP fnc.	?X=IND Y	?X# IND Y	?X< IND Y	?X<= IND Y	?X> IND Y	?X>= IND Y

However, the similarities end there - as the new functions expand the number of choices beyond the "IND Y" case, have a prompting U/I and perhaps most importantly they don't require altering the contents of the stack to perform the comparisons. Also in terms of byte usage both schemes are comparable, as the CX functions require at least one byte in Y to be used for register argument.

In terms of the Data Register exchange, there are also a couple of alternatives within the standard CX functions or other modules to perform equivalent actions, such as:

- **Rnn <> Rkk** can be done with **REGSWAP**, using nnn.kkk in X
- **Rnn <> Rkk** is also possible with **X<I>Y**, with "nn" in Y and "kk" in X (or vice-versa).

Which depending on the data register numbers may be more or less favorable in terms of byte count; see for example exchanging R10 and R25 below using the three approaches:

SELCT 10 **S<>** 25	10,025 **REGSWAP**	10, ENTER^ 25, **X<I>Y**
8 bytes, no stack	8 bytes, X used	7 bytes, both X,Y used

Compatibility with other Prompt Lengthener alternatives.

A more interesting comparison can be made with the other implementation of the Extended Prompts, like the ZENROM does using the EEX key, or even the Prompt Lengthener feature in the AMC_OS/X Module using the ON key.

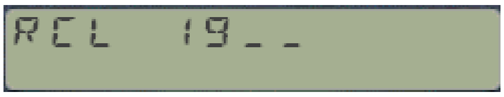
For these two implementations, the second byte of the RCL is *added to the same instruction in a program*, i.e. RCL 111 will be displayed as "RCL J", and similarly RCL 127 will show as "RCL e". This is clearly more efficient in byte usage; however *it does not support the RCL arithmetic operations* allowed by this module.

Note that the OS/X Prompt lengthener is only triggered with the standard OS-provided functions, and therefore won't appear at the custom prompt offered by "RKL _ _" or "RIND2 _ _"; nor by the ZENROM's after you have pressed the EEX key, i.e. "RCL 1_ _". Pressing the ON key in those instances will just turn the machine off.

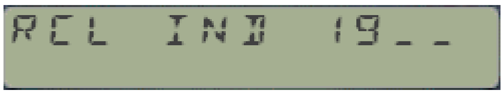
But you can have it both ways: if you have the OS/X Module plugged in (as every power user should :-) you can take advantage of this method by pressing again the RCL key at the RKL _ _ prompt: as mentioned before, this will revert to the standard RCL _ _, and then press ON to extend the field to three digits and enter "1xx" directly.

Say what, one-thousand registers?

It is also possible to press the EEX key while the OS/X extended prompt is up, which would add another field to it and so appearing to allow choices of data registers above 999 – if it weren't for the fact that such a thing can't physically exist on the normal machine (the 41CL is a different story). See for example the examples below, calling for a data register above 1,900:


RCL 19_ _

or:


RCL IND 19_ _

If you did that in PRGM mode, say entering 1900 in the prompt, surprisingly the end result turns out to be "RCL G" – which equals RCL 108. This can be explained by the (apparently unrelated) fact that MOD(1900, 128) = 108, i.e. we've gone full circle in data registers parlance.

Program Example – Congruence Equation

The program below is a direct translation of the original written by Thomas Klemm for the HP-42. See http://www.hpmuseum.org/forum/thread-1116.html

It solves for x in the equation: A * x = B mod N

The only changes pertain to the RCL math steps located at lines 14, 19, 22, and 68: simply add the register number as a second line after the RCL function as detailed in the table shown in page 7. (You can omit it on the case of zero).

```
00 { 104-Byte Prgm }
01▶LBL "CONG"
02 STO 00
03 STO 02
04 R↓
05 STO 01
06 R↓
07 STO 03
08 CLX
09 STO 04
10 1
11 STO 05
12▶LBL 07
13 RCL 02
14 RCL÷ 03
15 IP
16 STO 06
17 RCL 05
18 XEQ 02
19 RCL- 04
20 +/-
21 X<0?
22 RCL+ 00
23 X<> 05
24 STO 04
25 RCL 03
26 X<> 02
27 RCL 06
28 RCL× 02
29 -
30 STO 03
31 X≠0?
32 GTO 07
33 RCL 04
34 RCL 01
35▶LBL 02
36 STO 08
37 CLX
38 STO 09
39 X<>Y
40 STO 07
41▶LBL 04
42 2
43 ÷
44 ENTER
45 IP
46 STO 07
47 -
48 X=0?
49 GTO 05
50 RCL 08
51 RCL 09
52 XEQ 00
53 STO 09
54▶LBL 05
55 RCL 08
56 XEQ 01
57 STO 08
58 RCL 07
59 X≠0?
60 GTO 04
61 RCL 09
62 RTN
63▶LBL 01
64 ENTER
65▶LBL 00
66 RCL 00
67 RCL ST Z
68 RCL+ ST Z
69 X<Y?
70 RTN
71 R↓
72 -
73 +
74 END
```

Example: *5 * x = 3 mod 17*

Solution: `5, ENTER, 3, ENTER, 17, XEQ "CONG" => 4`

The Double Indirection: A solution in search of a problem?

Arguably a double indirection capability may be seen more as an extravaganza than as a useful feature. After all, how many times have you encountered a situation where the indirect index was itself depending on another variable, and doing so in a counter-like fashion?

Well those situations do exist, more often than none and with increased likelihood as you get into advanced algorithms and matrix applications – but I won't tire you with examples; rather here are functions **SIND2**, **RIND2** and **XIND2**, which perform a double STO/RCL/SWAP IND IND _ _

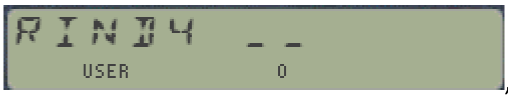
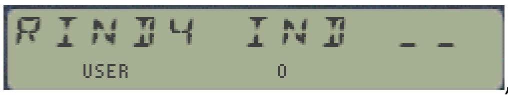
Enough to make your head spin a little, right? – Then you should try the TRIPLE indirection, available when you hit the shift key at that stage, ie:

SIND2 IND _ _	=	STO IND IND IND _ _	(Main function)
RIND2 IND _ _	=	RCL IND IND IND _ _	(Main function)
XIND2 IND _ _	=	X<> IND IND IND _ _	(Sub-function, thus needs **WF$** to launch)

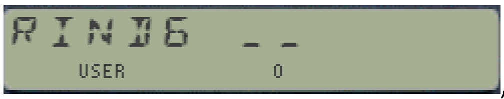
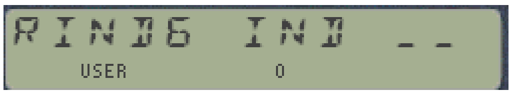
These functions use two (or three if SHIFTED) standard data registers to hold the arguments of the data register where the value is to be recalled from (RIND2), stored into (SIND2), or exchanged with stack reg X (XIND2). Better keep your register maps handy!

Going over the top: Multiple Indirection

Interesting things happen if you keep pressing the [SHIFT] key - as these functions support a *multiple indirection pattern* that allows redirecting the target registers as many as 10 levels (and beyond). The function prompt will change to reflect the current level, with a combination of even values and their IND options. For example, pressing [SHIFT] at the RIND2 IND _ _ prompt will bump the counter to:


```
RIND4 _ _
USER 0
```
, and then:

```
RIND4 IND _ _
USER 0
```
,

Followed by the screens shown below in a continuous sequence:


```
RIND6 _ _
USER 0
```
, and then:

```
RIND6 IND _ _
USER 0
```

Example: assuming the following registers contain the values shown below:

R10 = 0;	RCL 10	=	0	
R00 = 3;	RCL IND 10	=	3	
R03 = 5;	**RIND2** 10	=	5	
R05 = 7;	**RIND2** IND 10	=	7	
R07 = π	**RIND4** 10	=	π	
	RIND4 IND 10	=	5	
Then we have:	**RIND6** 10	=	7	, etc…

Note that this functionality is restricted to manual mode only, and when this function is used in a running program it'll be limited to a double indirection (or triple in the IND case).

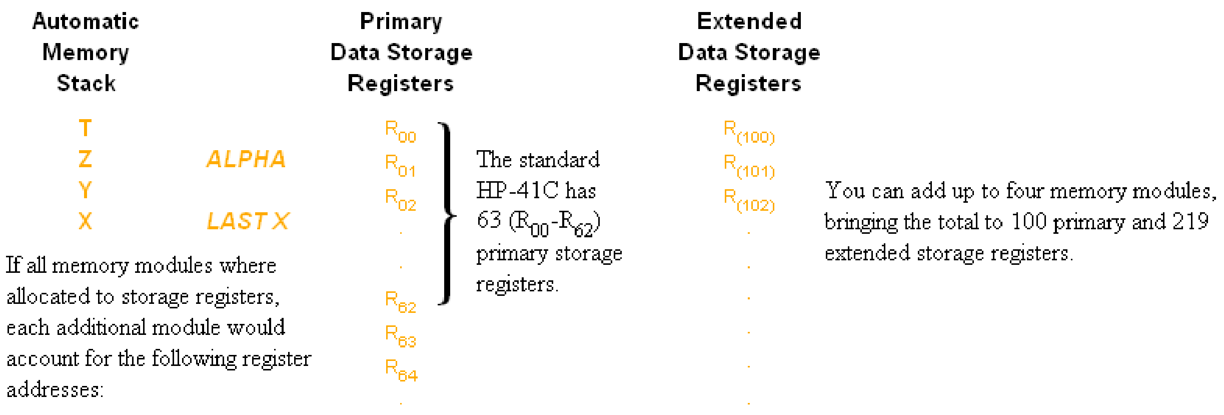
Application Example: Bubble Sort without data movement. *(By Greg McClure)*

```
;
; FIXED SORT -- Gregory J. McClure
;
; Does a non-destructive bubble sort of registers specified in another
; set of consecutive pointer registers.  The data to sort is not moved,
; but the pointer registers will be changed to reflect the numeric
; order (ascending) of the values indirectly pointed to by them.
; R00 thru R02 are used by the program.
;
; Example: R03-R06 contain 10, 12, 15, 18.
; R10, R12, R15, R18 contain the data to sort (4, 3, 2, 1).
; X contains 3.006 as descriptor of pointer register set, then SORT is run.
; When done, SORT will change R03-R06 to contain 18, 15, 12, 10.
; R10, R12, R15, R18 will be unchanged.
;
```

```
01 *LBL "SORT"
02 *LBL 10
03  STO 00       ; 1ST VALUE POINTER
04  STO 01       ; 2ND VALUE POINTER
05  ISG 01
06  STO 02       ; SAVE 1ST POINTER
07 *LBL 00
08  RIND2        ; TTRKALL DOUBLE IND READS
09  1
10  X>Y?
11  GTO 01       ; SKIP SWAP
12  RCL IND 00   ; RECALL POINTERS
13  RCL IND 01
14  STO IND 00   ; REVERSE POINTERS
15  X<>Y
16  STO IND 01
17 *LBL 01
18  ISG 00       ; BUMP VALUE POINTERS
19  ISG 01
20  GTO 00       ; MORE TO COMPARE
21  RCL 02       ; GET CURRENT POINTERS SET
22  E-3
23  -
24  ENTER^
25  INT
26  1.001
27  *
28  X=Y?
29  GTO 02       ; DONE
30  RCL 02
31  GTO 10
32 *LBL 02
33  "DONE"
34  AVIEW
35  END
```

Appendix.- A trip down to Memory Lane.

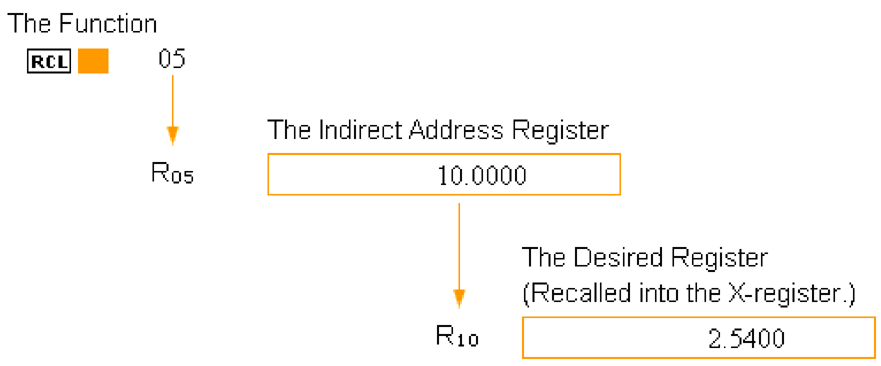
From the HP-41 User's Handbook.-



The standard HP-41C has 63 (R_{00}-R_{62}) primary storage registers.

You can add up to four memory modules, bringing the total to 100 primary and 219 extended storage registers.

If all memory modules where allocated to storage registers, each additional module would account for the following register addresses:


The Function
RCL
05
R05
The Indirect Address Register
10.0000
The Desired Register
(Recalled into the X-register.)
R10
2.5400

Storage Register Arithmetic

Arithmetic can be performed upon the contents of all storage registers by executing STO followed by the arithmetic function followed in turn by the register address. For example:

Opertion	**Result**
STO + 01	Number in X-register is added to the contents of register R_{01}, and the sum is placed into R_{01}. The display execution form of this is ST+.
STO - 02	Number in X-register is subtracted from the contents of register R_{02}, and the difference is placed into R_{02}. The display execution form of this is ST-.
STO × 03	Number in X-register is multiplied by the contents of register R_{03}, and the product is placed into R_{03}. The display execution form of this is ST×.
STO ÷ 04	Number in R_{04} is divided by the number in the X-register, and the quotient is placed into R_{04}. The display execution form of this is ST÷.

Say what, a Dynamic Display? The FIX ALL functionality.

Much more than a cosmetic affair, the ability to present only the non-zero decimal digits of a number has the value to provide additional information on the result: to the limit of the calculator resolution there are no further meaningful digits after the shown ones.

The FIX ALL feature is activated when you execute **FIXALL** (no arguments needed), and remains active until you change the display setting again using the standard FIX, SCI, or ENG functions.

Note that the representation will apply to the mantissa of the numbers, even if their exponents exceed E9; obviously limited by the numeric range of the calculator – which for the HP-41 is:

] -1 E100, -1 E-100 **[** {+} **]** 1 E-100, 1 E100 **[**

In case you're curious, the algorithms used by FIXALL are described below. You're also encouraged to check the SandMath Manual – an excellent reference for the design criteria for the RCL math functions. Note also that contrary to the SandMath's case, on this module the I/O_SVC interrupt polling technique is not used to link the standard RCL function with its extensions or the RCL Math sub-functions. No need for that, since a dedicated **RKL** replacement is used instead of the native one and our code takes complete control of the keyboard actions.

Formulas used – A general algorithm.

BCD numbers on the 41 platform are represented in the registers using the following convention:

```
"s|abcdefghij|xyz",
```

with one digit for the mantissa sign, 10 digits for the mantissa, one for the exponent sign and two for the exponent. This enables a numeric range between +/- 9,999999999 E99, with a "hole" around zero defined by the interval:]-1E-99, 1 E99[

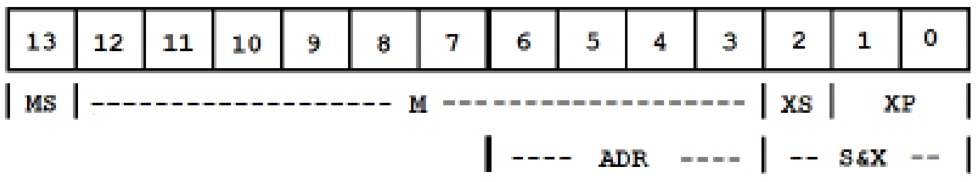
Let z# = number of mantissa digits equal to zero, starting from the most significant one (i.e. from PT=3 to PT=12). Then the fix setting to use is a function of the number in X , represented as follows:

1. If number >=1 (or x="0") - Let XP = value of exponent (yz). Then we have:

   FIX = max { 0 , [(9-z#) + XP] }

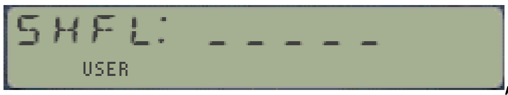
2. If number < 1 (or x="9") - Let |XP| = (100 – xyz) . Then we have:

   FIX = min { 9 , [(9-z#) + |XP|] }

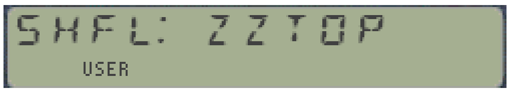


Stack Shuffling, Sorting, and selective Editing. { **SHFL** }

There are several functions in the native set to handle the stack registers, and certainly this module adds its dose of extensions and additions to the set, with the swap functions in particular being the best exponent. Many ways to skin this cat, but just in case you longed for more abstraction the function **SHFL** provides a general-purpose way to perform bulk stack alterations in a very convenient manner.


SHFL: _ _ _ _ _
USER

, i.e:


SHFL: ZZTOP
USER

SHFL prompts for five stack register letters, including the main XYZTL registers, or the Alpha registers MNOP, or even the Q register. Once the prompt is filled the contents of the main stack will be changed to reflect the sequence defined in the prompt. A few examples will clarify:

SHFL: XYZTL	leaves things unchanged – i.e. the "do nothing in 10 bytes" choice.
SHFL: YZTXL	performs the equivalent to RDN
SHFL: TXYZL	is equivalent to the standard R^
SHFL: XXXXL	fills the stack (except L) with the value in X

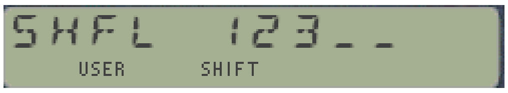
Other combinations will require two or more standard instructions or may not be easily possible without adding several of them – especially if you include the ALPHA registers to the choices. In this regard, the prompt allows Q(9) and the ALPHA registers as inputs, although a few considerations must be made:

- Register M is always used to hold the master string itself.
- Registers N,O,P are widely available.
- Remark that you'll be doing the equivalent to STO, but not to ASTO
- Register Q(9) is usually compromised, as it's used as scratch by the OS

Additionally, and continuing with the 'ZERO' theme as surrogate stack option - *you can also use the digit zero "0" in the input prompts*. This has the effect of clearing the corresponding stack register during the execution of the function. For example:

SHFL: 00000	is equivalent to CLST, STO L
SHFL: YX00L	is equivalent to X<>Y, RDN, RDN, CLX, RDN, CLX, RDN, RDN
SHFL: ZZT0P	copies Z to X,Y, T to Z, clears T and puts P in the LastX

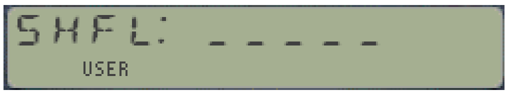
But wait, there's more: in the latest revision the function also *allows numeric digits in the prompt,* using [SHIFT] followed by the corresponding number key (0 – 9). This comes very handy to populate the stack registers *en masse,* useful for index setting, etc.

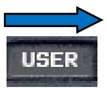

SHFL 123_ _
USER SHIFT

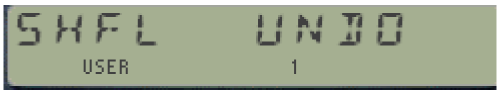
-> will enter 1 in X, 2 in Y, 3 in Z, etc…

Undoing the Stack re-arrangements

Pressing the [USER] key at the first prompt will undo the last stack re-arrangement, restoring the contents it had before the previous execution of **SHFL**. In manual mode this will be shown as follows:

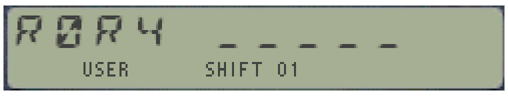

SHFL: _ _ _ _ _
USER



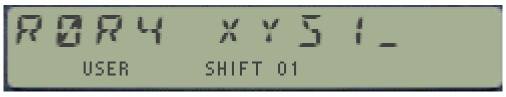

SHFL UNDO
USER 1

A Data Registers Shuffle. { R0R4 }

We've learned that entering numbers in the **SHFL** prompt is a shortcut to input those integer values in the stack registers on the fly, while doing the actual re-sorting. Thus, they're not to be confused with Data Register numbers. **R0R4** is the function to use if what you need is to *re-arrange the contents of data registers* using either the stack or the registers themselves. **R0R4** allows you to redefine what goes into the first five data registers {R00 – R04}, choosing from the stack values or from the existing contents of single-digit data registers {R00 – R09} as well.

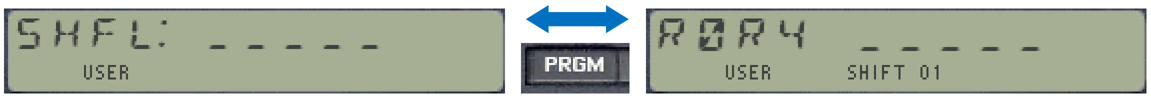


i e:



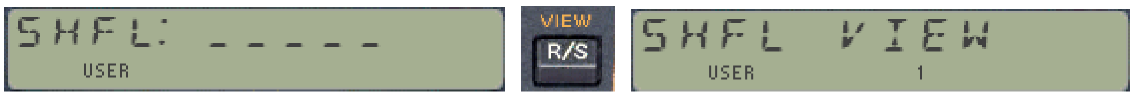
Simply enter the number of the data register (up to R09) or the letter of the stack/ALPHA register in the five-field prompt to select the source of the data value for each field. Note the needed use of [SHIFT] key to select either letters or numbers, as always. The input sequence defines the location order as well.

Note that **SHFL** and **R0R4** can be toggled pressing the [PRGM] key at the initial prompt:



Checking the results.

For a quick check of the results, you can press the [R/S] key to access use the "view" functionality. This will show a sequential list of the stack registers in L-X-Y-Z-T order (or the first five data data registers in R0R4's case of course) – a nice complement to help you keep your bearings at all times. Note that the sub-function **STVIEW** is always also available for an enumeration of the stack registers. **STVIEW** is accessible pressing [R/S] at the main **STK:** launcher screen.

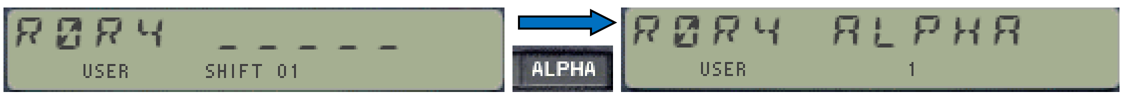


Program Usage.

Entering these functions in a program will follow the standard rule, i.e. the **SHFL** instruction will be placed in a single program step. You need to remember to manually add the master string as ALPHA step in the instruction *before* it.

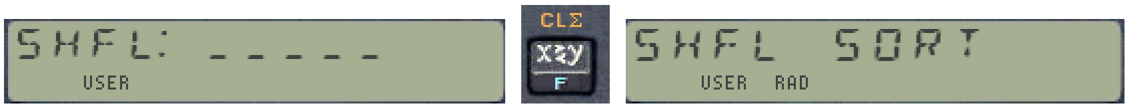
If [ALPHA] is empty the [**UNDO]** functionality will be triggered to restore the stack contents as it was prior to the previous execution of SHFL/R0R4. Note that a DATA ERROR message will come up (and the program execution will halt) should that string contain any invalid character – but it will ignore characters beyond the fifth one starting from the RIGHT of ALPHA (i.e register M).

Note that also *in manual mode* there's the possibility to use the current content of ALPHA without having to retype it at the prompts – all you need to do is just press [ALPHA] as a shortcut.



Sorting is also possible

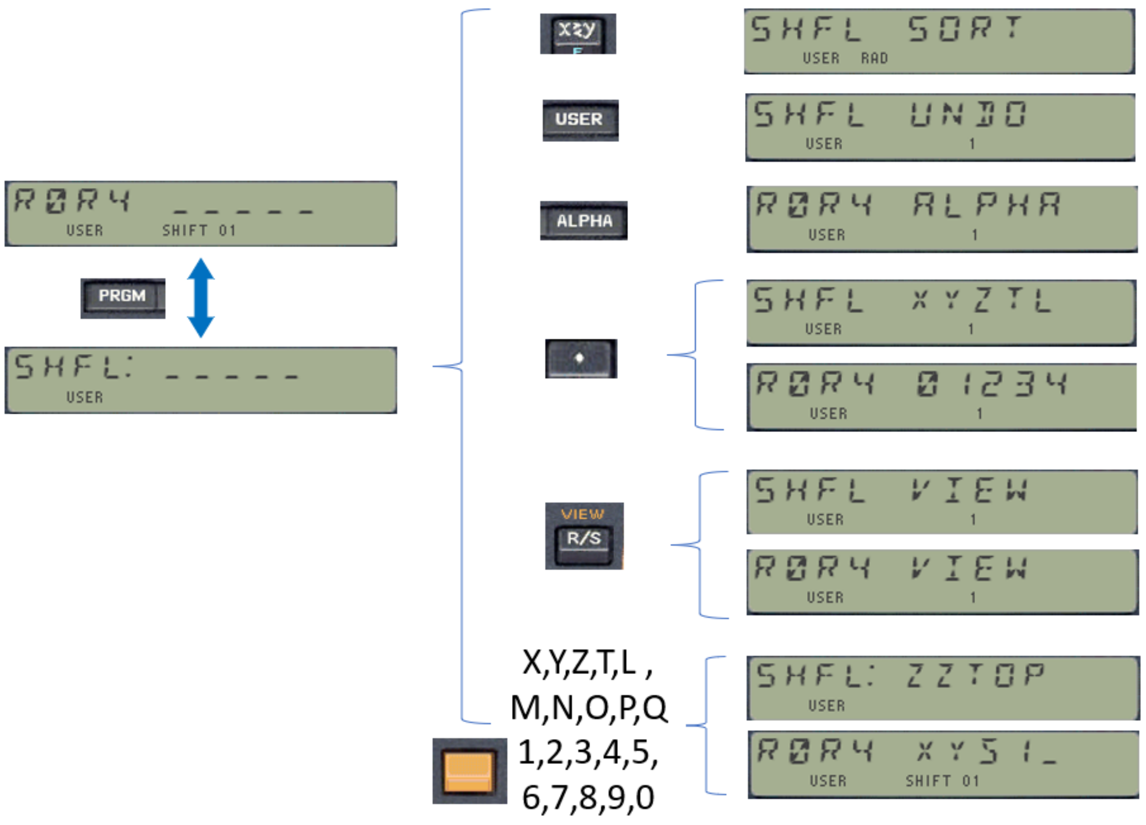
Revision K13 adds the SORT option to the set. Use it to do an *ascending sort of the involved registers*: either the stack XYZT or registers {R00-R03}. This option is accessed by pressing the X<>Y key in manual mode, or via the sub-functions **SSORT** and **RSORT** in the auxiliary FAT.



The original contents of the stack or data registers is saved in the buffer, so you can always use the UNDO option to revert to the unsorted configuration.

SHFL and R0R4 U/I functionality

The picture below summarizes all options offered by the SHFL / R0R4 functions. Note how they're interrelated and complementary of each other using the different hot keys.



- For **SHFL**, the default selection prompts for *Stack/Alpha register names*. Use [SHFT] to enter ***numeric values*** (from 0 to 9) into the target stack registers.

- For **R0R4** the default selection prompts for *Stack/Alpha register names*. Use [SHIFT] to enter ***Data Register numbers*** (from 0 to 9) into the target Data Regosters

The "Shadow Stack" concept.

The underpinnings of **SHFL** and **R0R4** take full advantage of the "emergency storage" buffer – whereby the stack registers are first copied to the buffer registers in the sequence defined by the master string, and then they're swapped with the stack in the "default" natural sequence X-Y-Z-T-L. This is the most effective way (code-wise) to perform the shuffle, and speed-wise it adds no significant penalty speed wise.

As a lateral thinking, you can use this design to make a copy of the stack to the buffer, not altering its contents – in case you'd want to restore all the contents after some operation (via UNDO), or simply as a safety backup. For this "blank" (no sorting) you need to enter the string "XYZTL" at the prompt/ To make this even more convenient, the **SHFL** function has a hot-key that introduces the default sequence {XYZTL} for you, no need to type it up. *Simply press the [RADIX] key at the initial prompt* (with the five fields shown) and enjoy the show.

Likewise, there is also a pre-built default combination for **R0R4** that really comes handy to save the contents of the first five data registers into the Shadow buffer without altering their content. Obviously, such combination is {01234}, and you can trigger it using the radix key at the prompt – no need to type the five numbers at the prompt.

To restore the original values after you've used the stack or R0R4 for other purposes, just call **bRCL** on the buffer registers following this arrangement:

X <–> bR5	T – bR2	;	R00 <–> bR5	R03 <–> bR2
Y <–> bR4	L – bR1	;	R01 <–> bR4	R04 <–> bR1
Z <–> bR3		;	R02 <–> bR3	

Example. Copy the contents of the stack into Data Registers R00 to R04

A trivial application of **R0R4** solves this case:

R0R4 "XYZTL" – or the reverse if preferred:

R00:	X
R01:	Y
R02:	Z
R03:	T
R04:	L

R0R4 "LTZYX" to have them saved as follows:

R00:	L
R01:	T
R02:	Z
R03:	Y
R04:	X

Example: Enter the integer sequence 4, 5, 6, 7, 8 in data Registers R00 to R04

Here we use both SHFL and R0R4 to get this quickly done as follows:

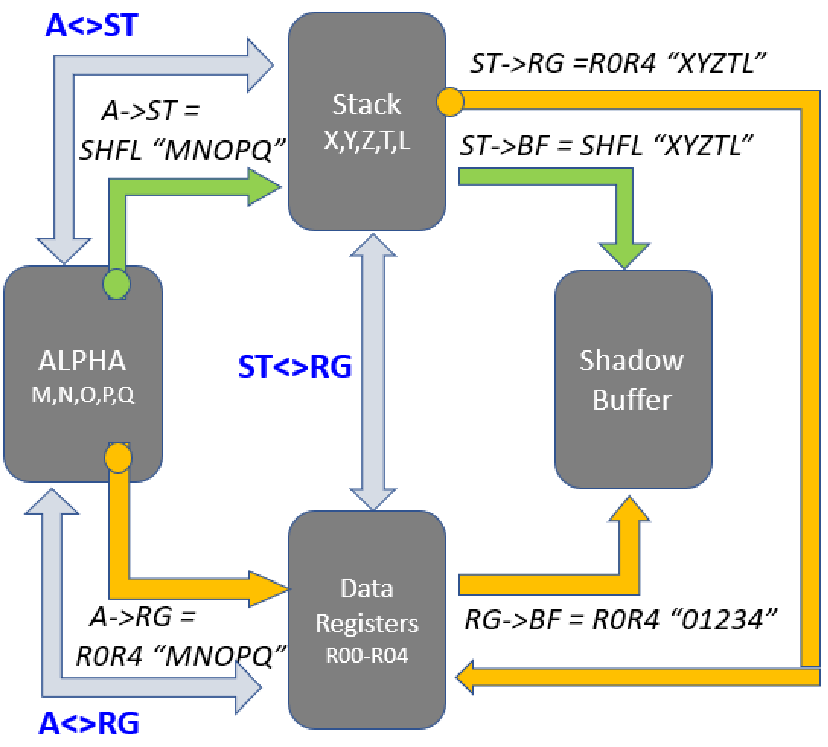
SHFL 45678	enters the desired sequence in the stack
R0R4 XYZTL	copies it to the data regs (and the shadow buffer)

Which may not be a very efficient way to get the job done in terms of the byte count, for that you'd better stick to the classic approach "*number, STO nn, number, STO nn….* " – but on the other hand it's unbeatable in leaving the stack undisturbed (except the M-register in ALPHA obviously).

```
01  STO 00          07  STO 03
02  RDN             08  RDN
03  STO 01          09  X<> L
04  RDN             10  STO 04
05  STO 02          11  X<> L
06  RDN
```

Data Transfer between Stack, Alpha and Data registers.

The picture below shows a conceptual summary of **SHFL** (green arrows) and **R0R4** (orange arrows) used as data transfer functions – i.e. not taking advantage of their sorting capability. You can see how they compare to the more direct functions (blue arrows) **A<>ST**, **A<>RG**, and **ST<>RG** also available in this module.



Examples: Saving and Restoring Stack data to/from Buffer #7 (*)

```
1   LBL "ST>B7              12  RDN
2   X<> L                   13  RTN
3   bSTO 1                  14  LBL "B7>ST
4   X<> L                   15  bRCL 1
5   bSTO 5                  16  SIGN
6   RDN                     17  bRCL 2
7   bSTO 4                  18  bRCL 3
8   RDN                     19  bRCL 4
9   bSTO 3                  20  bRCL 5
10  RDN                     21  END
11  bSTO 2
```

*(*) Note that LBL "ST>B7" is equivalent to **SHFL** "XYZTL", and that "B7>ST" does the same as **SHFL** UNDO. Thus these user routines are for illustrative example only.*

Example. The following example was provided by Didier Lachieze. A subroutine using only the stack to calculate the sum of the proper divisors of the number in X, it returns this sum in X and the initial number in Y.

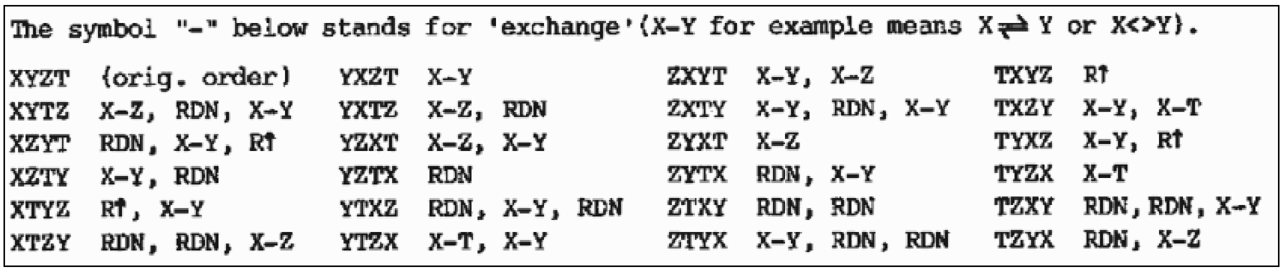
```
                                X     Y     Z     T
01 *LBL "DVSM                   n
02  1                           1     n
03  "XYXX"                      1     n     1     1
04  SHFL
05 *LBL 05
06  ISG T                      -     n     s     d
07  NOP
08  "YYZT"                      n     n     s     d
09  SHFL
10  RC/ T                       n/d   n     s     d
11  ?X< T
12  GTO 10
13  FRC?                        n/d   n     s     d
14  GTO 10
15  ?X= T                       n/d   n     s     d
16  GTO 00
17  RC+ T
18 *LBL 00
19  ST+ Z
20  GTO 05
21 *LBL 10
22  X<> Z                       s     n     n/d   d
23  END
```

The first occurrence at steps 03/04 is replacing the two instructions STO Z, STO T, and the second occurrence at steps 07/08 is also replacing two instructions: CLX, RCL Y. Note that for step 12 you'd need the function **FRC?**, available in the SandMath module - or an equivalent function from your own sources.

Equivalences with standard functions.

The table below shows the sequence of standard instructions equivalent to the different combinations of the stack registers. Obviously, this doesn't include support for the ALPHA registers and nor does it have the option to enter integer values directly either, but it's a good reference to have.

Table 3 – Stack manipulation examples from "Calculator Tips & routines", pg 26 – by John E. Dearing.



The symbol "-" below stands for 'exchange'(X-Y for example means X⇄Y or X<>Y).

XYZT	(orig. order)	YXZT	X-Y	ZXYT	X-Y, X-Z	TXYZ	R↑
XYTZ	X-Z, RDN, X-Y	YXTZ	X-Z, RDN	ZXTY	X-Y, RDN, X-Y	TXZY	X-Y, X-T
XZYT	RDN, X-Y, R↑	YZXT	X-Z, X-Y	ZYXT	X-Z	TYXZ	X-Y, R↑
XZTY	X-Y, RDN	YZTX	RDN	ZYTX	RDN, X-Y	TYZX	X-T
XTYZ	R↑, X-Y	YTXZ	RDN, X-Y, RDN	ZTXY	RDN, RDN	TZXY	RDN, RDN, X-Y
XTZY	RDN, RDN, X-Z	YTZX	X-T, X-Y	ZTYX	X-Y, RDN, RDN	TZYX	RDN, X-Z

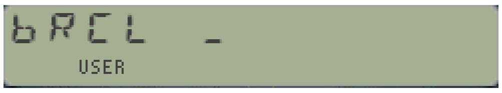
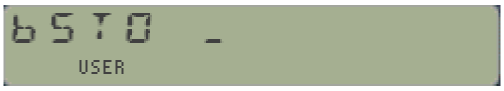
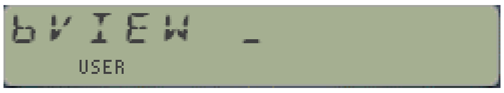
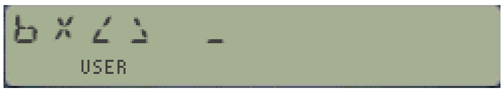
Shadow Buffer Registers Storage.

If you've ever run out of data registers and wished there was a "back-door" mechanism to use in emergencies, then you should find this section interesting. These functions operate on a I/O buffer (with id#7) located below the .END. and above the Key assignment area.

The buffer holds *seven extra registers* for data storage, labeled bR1 to bR7 (therefore there's no bR0 to speak of). Just enter the index for the extended register in the prompt and the data will be stored, recalled, or exchanged with the stack X-register – as if they were standard data registers.

- **bRCL _** recalls to the X register the content of the extended reg. whose index is provided in the prompt, or in the next program line if used in a running program.
- **bSTO _** stores the X-register in the extended reg. given in the prompt, or in the next program line if used in a running program.
- **bVIEW _** shows the contents of the buffer register with index given in the prompt.
- **bX<> _** exchanges the contents of the X-register and the buffer reg. which index is provided in the prompt, or in the next program line if used in a running program.

It you try to enter a non-valid index number (basically anything larger than 7) the prompt will be maintained (without an error condition) until you either cancel the function or enter a valid value. In program mode this would show a NONEXISTENT message and the execution will halt – so be careful when you enter the parameter- which must be done manually for all sub-functions, and therefore should always be within valid range.


```
bRCL _
USER
```

```
bSTO _
USER
```

```
bVIEW _
USER
```

```
bX<> _
USER
```
You can navigate amongst these four functions using the RCL, STO, CHS and R/S keys

A Triple-duty buffer.

Besides the emergency storage registers, this buffer is also used for other two important purposes within this module, and a third one in the Formula_Eval ROM - as described below:

1. Buffer registers bR1 and bR2 are shared by the RTN stack functions **PUSHRTN** and **POPRTN**, so be careful not to override their content if both features need to be used together.
2. The first five buffer registers are used as temporary storage place by the stack shuffle function **SHFL –** as the most efficient way to re-arrange the stack registers on-the-fly (the "shadow stack" as it's been referred to sometimes).
3. All six buffer registers are used by the variable assignment in the Formula_Evaluation Module done with functions **LET=**, **GET=**, and **SWAP=**. Very much like the emergency buffer but with alphabetical labels "a" to "e", and "F".

Buffer Header: warping around SELECT.

In a daring move, here's where the emergency buffer and the Selected variable merge. As mentioned before, the buffer header contains in digits <5:3> the information of the currently selected variable, i.e. the data or stack register index marking such selection.

It was said in the previous section that the only valid input parameters for the buffer storage functions were 1 to 7; but even if that's conceptually correct it isn't entirely true: extending the definition to also include the value zero in the prompts, we can use the four functions described before to work *on the selected register* as well.

It's not the contents of the buffer header register which gets invoked, but the data register currently under the selection setup – as pointed to by the marker in the header. It is as if the register bR0 was an automatic INDirect operator for the four basic action: STO, RCL, VIEW and Exchange.

Therefore:

- **bRCL 0** recalls the *value of the selected register* to the X register in the stack. (**SRCL**)
- **bSTO 0** stores the value in the X register in the selected register, same as **SSTO**
- **bVIEW 0** shows the content of the selected register, i.e. is equivalent to **SVIEW**,
- **bX<> 0** exchanges the selected value contents with that in X, therefore it's equivalent to **S<>** ST_X - but coming the other way around – and saving bytes.

In case you didn't notice it, the value zero for any sub-function parameter doesn't need to get explicitly entered in the program – thus it's sufficient to just enter the sub-function without a non-merged second line. The only restriction is that the program step that follows it cannot be a number – which would be interpreted as its parameter otherwise.

So there you have it, yet another way to skin this cat – an interesting twist to the scheme, in case you wondered how much interconnectivity can we get between the different functionality areas of the module.

Remember that the buffer will be created when you switch the calculator ON the first time after plugging in the WARP module.

Register #	Storage	RTN Stack	Shadow Stack	Eval$ Vars
Seventh:	bR7	-	-	G
Sixth:	bR6	-	-	F
fifth:	bR5	-	Shadow-X	e
fourth:	bR4	-	Shadow-Y	d
third:	bR3	-	Shadow-Z	c
second:	bR3	reg 10(a)	Shadow-T	b
first:	bR1	reg 11(b)	Shadow-L	a
header:	SEL# pointer	-	-	

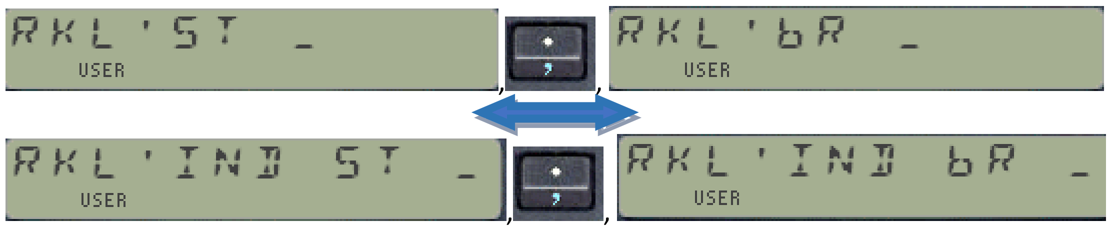
Warning: This buffer is automatically created by the module on start-up, so the data contained in it will survive a power-on/off cycle. This also applies to the selected variable used by SLCT.

Bringing the Shadow Stack registers into the RKL fold.

Revision K11 of the WARP_Core module has added a seamless integration of the buffer registers {a-F} as valid parameters for the extended comparison tests and exchange functions. Therefore, the buffer registers can be used in the data comparison and exchange functions, as well as in the Recall in-place Math operations such as RC+, RC-, RC* and RC/

In manual mode, pressing RADIX acts like a toggle between the STACK and BUFFER register selection; thus, to access the buffer registers you need to press the RADIX key while the display shows the STACK registers selection (i.e. twice in total). For example, using RKL and RKL IND:

,



RKL 'ST _
USER

RKL 'bR _
USER

RKL 'IND ST _
USER

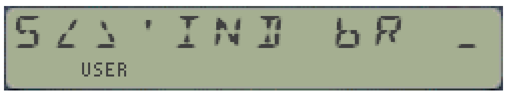
RKL 'IND bR _
USER

The available choices are {A, B, C, D, E, F, and G}, entered by pressing the [A] – [E] keys in the top row plus [F] and [G] in the second row. Note that the selected letter is briefly shown in the display as a capital letter to distinguish it from the lower-case Stack registers {a-e}.

Note that though similar in scope, this vastly supersedes the **bRCL** sub-function described earlier, which for starters didn't have INDirect or Math capabilities at all.

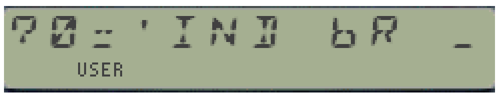
When you're editing a program, this action builds an argument index that will be saved as a program step right after the main function. This index has all the information required for the function to do the appropriate register selection at run time, as well as checking for its existence.

Not limited to the RKL situation at all, this functionality also applies to all testing and exchange functions included in the module, for instance:



S<> 'IND bR _
USER

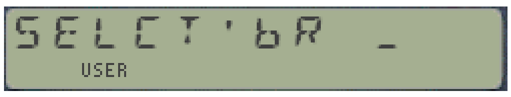
;



?0= 'IND bR _
USER

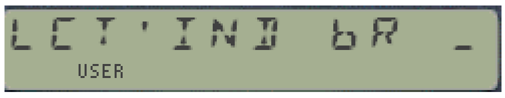
; etc.

It's also possible to use a buffer register as SELCT'ed variable "S" as well, either directly or indirectly:



SELCT 'bR _
USER

;



LCT 'IND bR _
USER

For INDirect addressing it's understood that the pointer register can be in the stack or the shadow buffer – but the value contained in it will always refer to a standard data register Rnn. By extension, only the first pointer register used by **RIND2**, **SIND2**, and **XIND2** can be a stack or shadow buffer register.

This brings even more convenience and capability to the already powerful functionality provided by the module. More choices at your disposal for a more flexible programming!

CODA: Finding the X-needle in the REG-haystack.

For those times when you'd like to know if a certain value is stored in the data register, the sub-function **FINDX** (a.k.a. **XF# 62**) is available to do a cursory comparison looking for a match with the value in the X-register. All data registers are checked, starting with R00 until the last one depending on the current SIZE. The error message NONEXISTENT will be shown if the calculator SIZE is zero.

The function returns the number of the first data register found that contains the same value as the X-Register. If none is found, the function puts -1 in X to signify a no-match situation. The stack is lifted so the sought for value will be pushed to the Y-register upon completion.

Listed below are two FOCAL routines that do the same job as **FINDX** – albeit slower and using auxiliary stack registers. It's interesting to compare the standard approach (on the right) with the alternate one (on the left) using the SELCT variable for indirect comparisons.

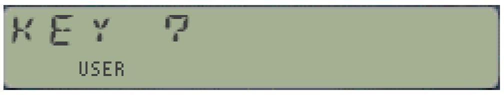
```
01  *LBL "XFND"
02   SIZE?
03   E
04   –
05   E3
06   /
07   SELCT  (IND X)
08   243
09  *LBL 00
10   WF#     (?S= Y)
11   39
12   114
13   GTO 02
14   ISG X
15   GTO 00
16   CLX
17    -1
18   RTN
19  *LBL 02
20   INT
21   END
```

```
01  *LBL "FNDX"
02   SIZE?
03   E
04   –
05   E3
06   /
07  *LBL 00
08   ?Y=  (IND X)
09   243
10   GTO 02
11   ISG X
12   GTO 00
13   CLX
14   -1
15   RTN
16  *LBL 02
17   INT
18   END
```

Again, note how the instruction **?S= IND Y** requires *a three-line non-merged instruction*, a good exponent of the versatility of this implementation indeed. The only thing to be aware of is that if you're SSTíng the program it will halt the execution – since the O/S interprets that ?S= is not programmable.

Playing with Key Assignments.

This module includes a couple of brand-new KA-related routines that you may find interesting. Their mission is to flip the key assignments on a given key or for the complete keyboard – so that the shifted and un-shifted assignments are mutually toggled.

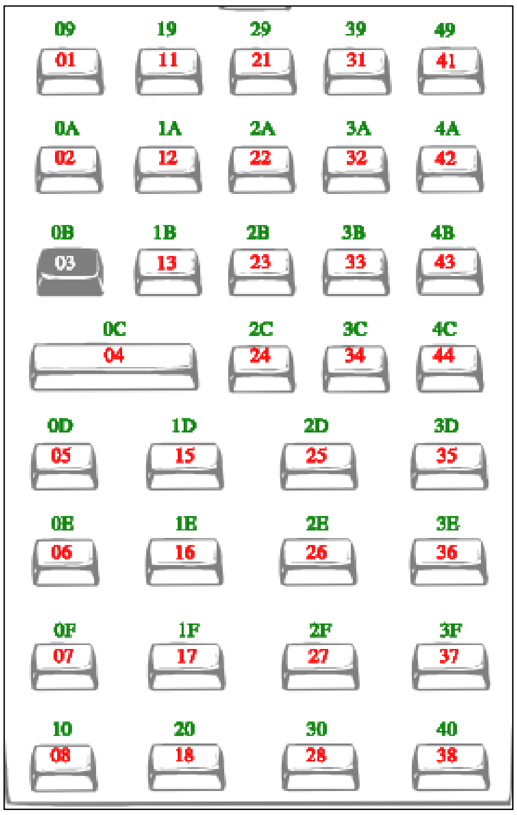


- **KAFLP** toggles all key assignments – turning shifted ones into non-shifted, and vice-versa. This will only leave unassigned keys unchanged, but will reverse the assignments if only one assignment exists for the keys.

- **KYFLP_** prompts for a key to perform the same task on an individual key basis. The prompt includes the back-arrow key but will ignore the toggle keys (ON/USER & PRGM/ALPHA)

In case you wonder why bother with this functionality, having the ability to toggle a key's USER key assignments becomes very handy if you have two function launchers assigned to that key.

A good example is with the SandMath, SandMatrix and 41Z modules – the three of them "competing" for prime time on the [Σ+] key. Flipping the assignments will save you a lot of [SHIFT] key pressings to access the functions within those launchers.



Saving Status Registers in X-Memory.

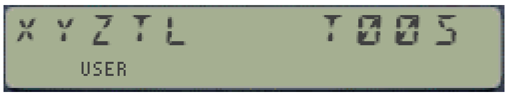
You can use sub-functions **SAVEST** and **GETST** to make backup copies of the status registers into X-Memory files, and to restore their contents back to the status area. The functions *prompt for the number of status registers to include in those back-up files*, which must be at least one and not more than 16. In manual operation the function won't allow you to enter values above 16 (first prompt must be 0/1; second prompt 0-6). If you use "00" then the complete 16 registers will be used instead.

For example if you just want to save the stack registers {T,Z,Y,X, and L} then you'd enter "05" in the prompt (since the count always starts with register T as the first one). The file name is expected to be in ALPHA - thus register M (and possibly N) would be partially used by the function itself.

Exercise caution when the upper stack registers are included, which will have dramatic effect in your program pointer and RTN stack in register a(11) and b(12); or stack assignments in registers |-(10) and e(15). Also don't underestimate the ability of a bad cold start in register c(12) to cause a MEMORY LOST condition when treated roughly.

These functions are programmable. In a running program the file name is expected in ALPHA, and the number of status registers is taken from the program line after the sub-function's index (must be added manually) – which won't be entered into the X register but as the prompt value instead. Yes, that's right: a triple non-merged lines case!

Note: The Status files have a dedicated file type in X-Memory. If you're using the AMC_OS/X Module, then their entries will be marked with the "**T**" prefix during the enumeration:


X Y Z T L T 005
USER

See the figure on the right showing the Stack register allocation within the X-Mem Data file. This particular example only goes up to 8(P), but in general you can save all the status registers, until 15(e) inclusive.

File End Marker
Register P(8)
Register O(7)
Register N(6)
Register M(5)
Register L(4)
Register X(3)
Register Y(2)
Register Z(1)
Register T(0)
FL Header Reg
FL Name Reg

Appendix. Duplicates in other Modules.

Some functions are also available in other advanced modules, as shown below:

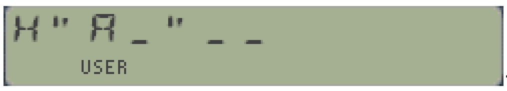
Function	Available in:	And also in:
GETST _ _	RAMPage ROM	PowerCL
SAVEST _ _	RAMPage ROM	PowerCL
KAFLP _	RAMPage ROM	XROM ROM
PUSHRTN	XROM ROM	RECURSE Module
POPRTN	XROM ROM	RECURSE Module
ROM2HEX _ _,_ _	XROM ROM	GJM ROM
HEX2ROM "A_"_ _	XROM ROM	GJM ROM
AIRCL _ _	ALPHA ROM	SandMath
PGCAT _	AMC_OS/X	Power_CL
BFVIEW _	RAMPage ROM	Power_CL
CSST	ToolBox	Power_CL

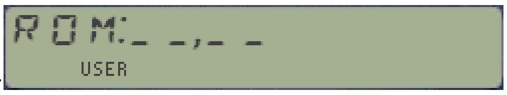
XROM to-and-from HEX bytes. (by Greg McClure)

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

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

If at any time during entry for any of these functions the opposite function is desired, pressing the "H" key will switch to the opposite routine (**ROM2HEX**<>**HEX2ROM**) – going back to the beginning of the data entry sequence.



<-->


Note that these functions are intelligent enough to discard illegal combinations of input values during the parameter entry – so you can't enter non-existing choices. This is of course non-withstanding the synthetic two-byte OS functions, but that's an entirely different subject.

Note that the result string is not placed in ALPHA – but you may use the function **DTOA** to move it there. Once the resulting string is in ALPHA it can be further used for register storage or any other string manipulation you require.

The table below shows the correspondences between the XROM id# and the HEX codes. Note that the first 64 entries are used by some synthetic multi-byte mainframe functions.

XROM id#	Hex Code	XROM id#	Hex Code	XROM id#	Hex Code	XROM id#	Hex Code
XROM 00	A0:00-:3F	XROM 08	A2:00-:3F	XROM 16	A4:00-:3F	XROM 24	A6:00-:3F
XROM 01	A0:40-:7F	XROM 09	A2:40-:7F	XROM 17	A4:40-:7F	XROM 25	A6:40-:7F
XROM 02	A0:80-:BF	XROM 10	A2:80-:BF	XROM 18	A4:80-:BF	XROM 26	A6:80-:BF
XROM 03	A0:C0-:FF	XROM 11	A2:C0-:FF	XROM 19	A4:C0-:FF	XROM 27	A6:C0-:FF
XROM 04	A1:00-:3F	XROM 12	A3:00-:3F	XROM 20	A5:00-:3F	XROM 28	A7:00-:3F
XROM 05	A1:40-:7F	XROM 13	A3:40-:7F	XROM 21	A5:40-:7F	XROM 29	A7:40-:7F
XROM 06	A1:80-:BF	XROM 14	A3:80-:BF	XROM 22	A5:80-:BF	XROM 30	A7:80-:BF
XROM 07	A1:C0-:FF	XROM 15	A3:C0-:FF	XROM 23	A5:C0- :FF	XROM 31	A7:C0-:FF

Saving and Restoring the RTN Stack.   (by Poul Kaarup)

The return stack can hold up to six addresses for subroutines, which is adequate for the vast majority of user code programs. Should that not suffice, the pair of functions described below can be used to extend that limit up to 12 addresses, effectively doubling he return capacity of the OS.

- **PUSHRTN** saves the current RTN stack into a memory buffer (with id#=7). Once saved, the current RTN stack is cleared (reset anew) so you have six more levels for your program.
- **POPRTN** restores from the buffer the RTN stack saved previously, effectively overwriting the current one at the moment of calling this call.

The program pointer (PC) and the first two pending return addresses are stored in status registers b(12), the third is stored as two halves on each register, and the remaining three in status register a(11). Note that these functions *will not save the Program Pointer information*.

This is shown in the figure below:

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	nibble

b(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	nibble

Obviously these two functions are meant to be used as a pair, in combination. Note also that because buffer#7 is used for the Stack shuffling too, you should refrain from calling **SHFL** and the direct buffer access while the extended return addresses are held in bR1 and bR2.

Because these functions use the first two registers in the "emergency buffer", you can always use the buffer recall function **bRCL** to inspect the contents of the *stored* RTN stack – and compare it with the *current* one, for example:

bRCL 1	**bRCL** 2
RCL b	RCL a
X=Y?	X=Y?

Two other functions dealing with the RTN stack are also available in the secondary FAT, as follows:

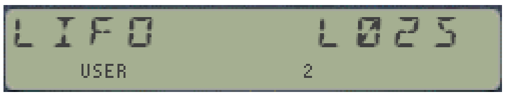
- **RTN?** Is a test function that checks whether there are pending returns in the stack. The result is YES/NO, skipping the next line in a program when false.
- **RTNS** recalls the number of pending subroutine levels to the X register, which by definition is an integer between 0 to six.

LIFO X-Functions. (by Doug Wilder)

The LIFO (Last In First Out) functions require extended functions memory to operate. The LIFO is located only in the first file in extended memory and must have a minimum size of one register and a maximum size of 120 registers. This structure allows maximum transfer speed, even faster than main memory, and does not require register numbers.

LIFO initialization: Create a first file in extended memory (recommended size is 16 to 32 registers) or if the first file currently in extended memory is of a suitable size, it may be used for the LIFO. Use a sequence similar to: "BUFFER" 28 CRFLD (the name is arbitrary). The function **LIFOINI** converts the first file in extended memory to the LIFO file type, any data in the file is unrecoverable.

If you're using the AMC_OS/X Module (always highly recommended), this is shown in a CAT#4 listing with an "**L**" character in the file type, i.e.:


LIFO L025

LIFOINI:

Converts the first file in extended memory to LIFO structure and initialize pointers.

After **LIFOINI** has been successfully executed without error, the stack is ready for use. LIFOINI may be executed again to reset the pointers. Ideally, **LIFOINI** would be only executed from the keyboard, however it may also be used in a main program, the uppermost or top driver program.

LIFO functions:

Z: is X and Y (complex data) , T: is Stack (XYZT), F: is Flags, A: is ALPHA, and R: is the RTN stack

↑X	↑Z	↑ST	↑F	↑A
POPX	POPZ	POPST	POPF	POPA

If the stack lift is disabled, **POPX** and **POPZ** do not cause a lift, eg, CLX, **POPZ** does not modify the Z and T registers. For multi-register push and pop functions, a "LIFO LIMIT" error leaves the stack in an unknown state and the LIFO pointer is left in an unknown state. For **POPA** or **POPF**, if a "DATA ERROR" occurs the Alpha/Flag register has not been modified yet the LIFO pointer is left in an unknown state.

Alpha data and Flag data are typed data, that is: one cannot pop numeric or Flag data into Alpha. Stack data is not typed: any type of data may be poped into the XYZT stack.

With an LIFO it is possible to write user code subroutines which simulate monadic functions, for example; do a push stack at entry, put the result in LASTX, then **POPST** and X<>L RTN.

It is also possible to write interrupting alarms which actually do something, they can push the stack/LASTX/Alpha/Flags at entry and recover them at exit. Thank's to HP for the forthought to not interrupt a running program when the stack lift is disabled.

POP of data into the stack is very fast unless a printer is attached, in which case the POP can be greatly slowed due to printer interface. For example a POPST in trace mode will do a full stack printout which can consume up to two seconds. In a running program, clearing F55 will greatly speed things up although trace capability will be lost.

These functions will report "NO XFM LIFO" if a lifo file does not exist. In that case you'll need to create it first and then try again.

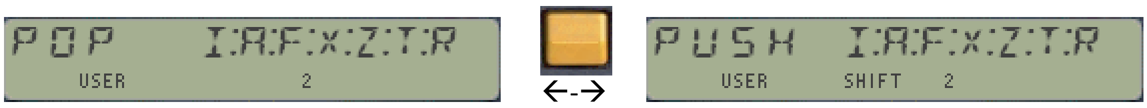
Finally, one must remember the basic rule for LIFO stack usage: whatever gets pushed MUST be poped and in reverse order! Otherwise we get what is known as a "memory leak" and eventual LIFO LIMIT error.

Launcher implementation

These functions are implemented in a LIFO launcher, with two components depending of the POP or PUSH actions. Each action is invoked by the corresponding sub-function POP and PUSH, in the auxiliary FAT. This means they are accessed via the WF$ and WF# sub-function launchers, as usual. The sub-function index is automatically entered by the function in program mode.

POP = WF# 97
PUSH = WF# 98

You can use the [SHIFT] key to toggle between them in run mode.



```
POP     I:A:F:X:Z:T:R
   USER          2
```

```
PUSH    I:A:F:X:Z:T:R
   USER     SHIFT   2
```

Function	I	A	F	X	Z	T	R
PUSH*	LIFOINI	PUSHA	PUSHF	PUSHX	PUSHZ	PUSHST	PUSHRTN
POP*	LIFOINI	POPA	POPF	POPF	POPZ	POPST	POPRTN

Notice that the option "R" stands for the **POPRTN** and **PUSHRTN** sub-functions, which use the I/O Buffer #7 and not the LIFO X-File

In a program each of the options in the launcher prompt needs to be manually added as a third program line, following the WF# and index# program steps. For instance, the code snippet below saves the contents of the ALPHA register using **PUSHA:**

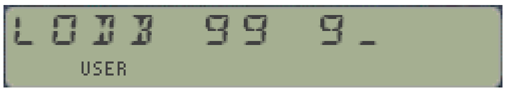
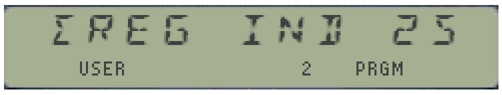
```
nn          WF#
nn+1        98
nn+2        2
```

Loading Bytes in RAM. (Nelson F. Crowle)

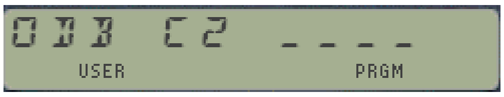
If you lived through the days of byte jumpers and load bytes' challenges, you'll no doubt have fond memories of what it was like to work with synthetics and PPC ROM routines. A few functions come from the NFC ROM and the ProtoCoder_1A. Consider them "modern day" versions (if such can be said of 30-year old code!) of some of those vintage routines, with a usability and convenience twist added by the MCODE implementation; as well as speed.

LODB _ _((_ _), _ _) is a very ingenious approach to solving the multi-byte loader problem. This function will prompt additional fields depending on the previous inputs – to complete the sequence required for 2-byte and three-byte instructions. The inputs are expected in Hexadecimal format, from 00 up to FF. See byte table in next page for details.

Example: to enter ΣREG IND 25 you can use the prompt values "99" and "99" at the initial and subsequent prompts (see the display below at the point of the last digit input):


```
LODB 99 9_
USER
```
→

```
ΣREG IND 25
USER      2  PRGM
```

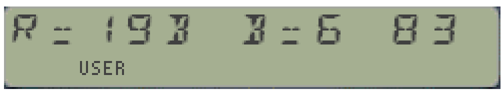
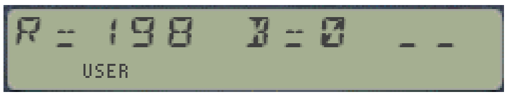
Example: Use this function to compile a 3-byte GOTO; note the prompt field will have 4 fields to input the next two bytes (for the jump address and the LBL number):


```
ODB C2 _ _ _ _
USER       PRGM
```

Make sure you have your copy of the Byte Table handy to provide the input parameters for the prompts.

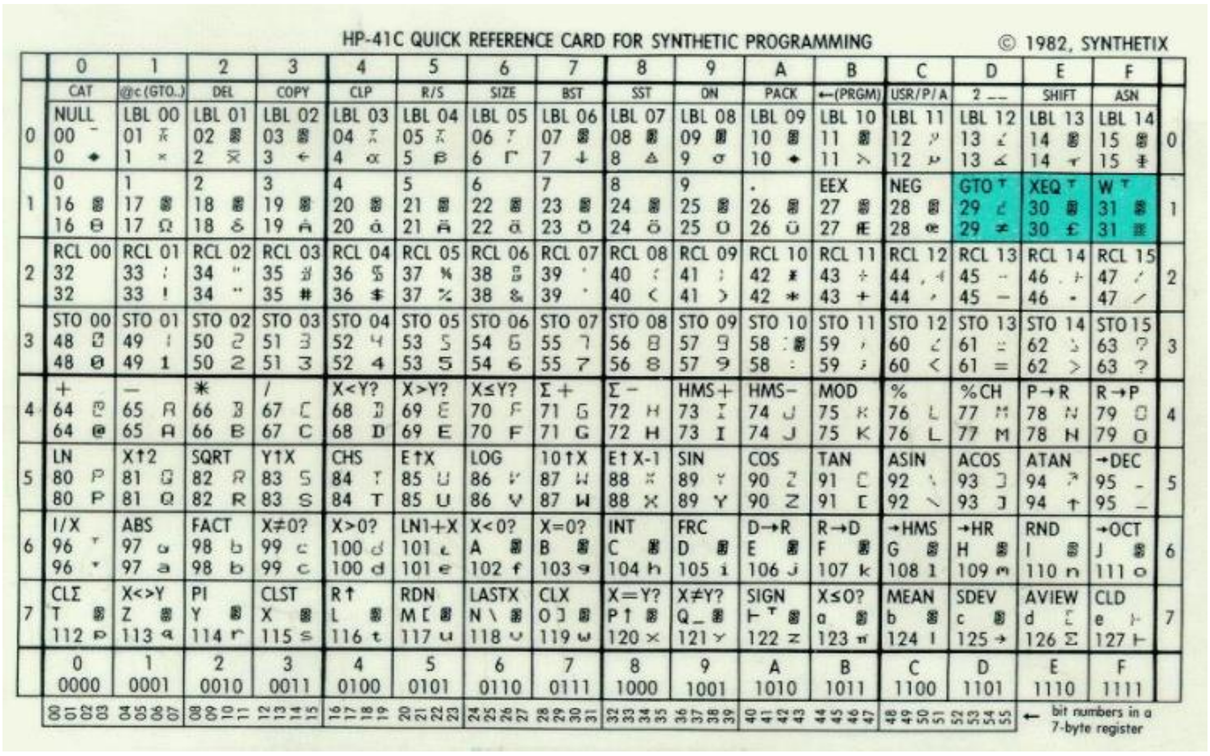
Note: This function is not finished – but it works for the majority of 2- and 3-byte combinations as-is. It was later superseded by a more systematic RAM-editor version in the Proto-Coder ROM, which is described below, but I think its ingenious approach deserves a place on this module.

LOADB _ _ is a more capable RAM Editor (this one is from the Proto Coder_1A ROM) that can be used to review and edit the contents at the byte level. It takes the starting position from the current PC location, and presents a prompt that shows the current register and byte number, as well as the current word value in hex at that address:


```
R: 193 B:6 83
USER
```

```
R: 198 B:0 _ _
USER
```

At this point you can use the [SHIFT]/[SST]keys to move up and down in memory, the [ENTER^] key to null the byte at that location, the [RCL] key to input a new RAM address, the [R/S] key to terminate the function and return to the OS, or the back-arrow key enables the value field to edit the byte with a new value.

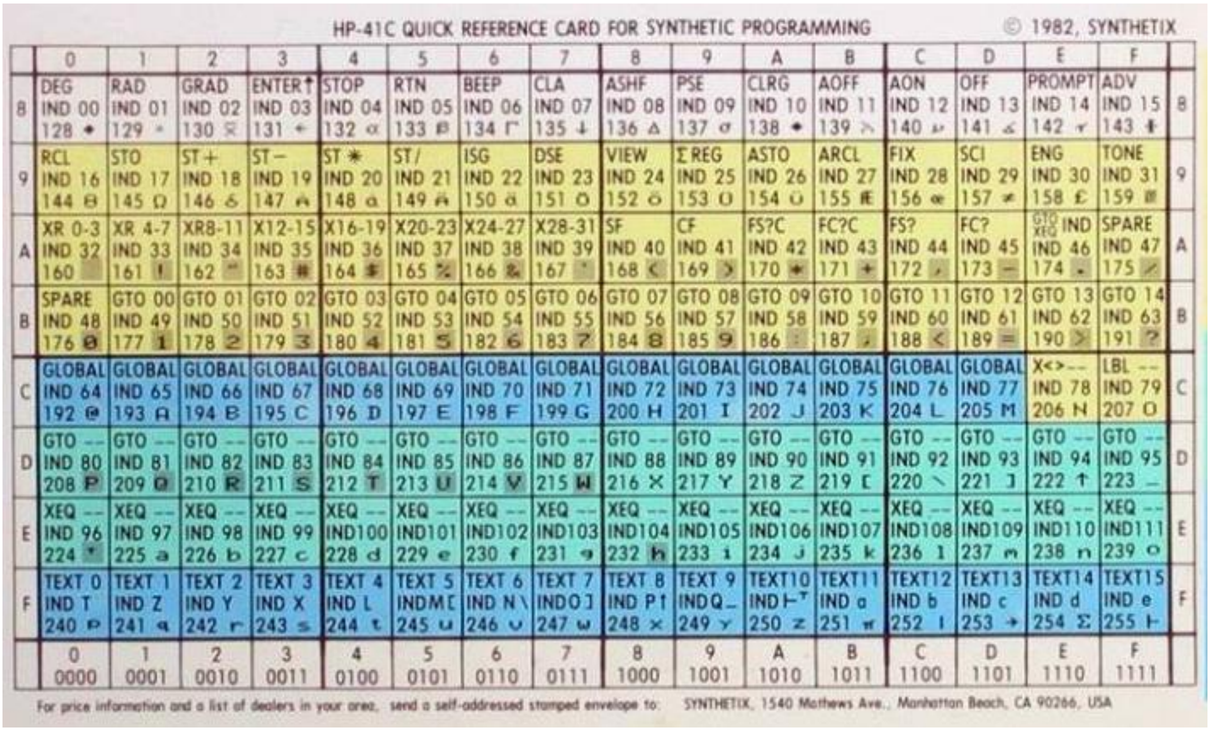
Be careful with the changes you make and be aware that pressing back arrow will require editing the byte value (ior pressing R/S). The byte address will not automatically increase after editing, so you're can see the result.



HP-41C QUICK REFERENCE CARD FOR SYNTHETIC PROGRAMMING © 1982, SYNTHETIX

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	
	CAT	@c(GTO..)	DEL	COPY	CLP	R/S	SIZE	BST	SST	ON	PACK	←(PRGM)	USR/P/A	2 __	SHIFT	ASN	
0	NULL 00 ¯ 0 ◆	LBL 00 01 x̄ 1 ×	LBL 01 02 ▒ 2 x̄	LBL 02 03 ▒ 3 ←	LBL 03 04 x̄ 4 α	LBL 04 05 x̄ 5 β	LBL 05 06 Γ 6 Γ	LBL 06 07 ▒ 7 ↓	LBL 07 08 ▒ 8 Δ	LBL 08 09 ▒ 9 σ	LBL 09 10 ▒ 10 ◆	LBL 10 11 ▒ 11 λ	LBL 11 12 μ 12 μ	LBL 12 13 ∠ 13 ∠	LBL 13 14 ▒ 14 τ	LBL 14 15 ▒ 15 Φ	0
1	0 16 ▒ 16 θ	1 17 ▒ 17 Ω	2 18 ▒ 18 δ	3 19 ▒ 19 Ȧ	4 20 ▒ 20 ȧ	5 21 ▒ 21 Ä	6 22 ▒ 22 ä	7 23 ▒ 23 Ö	8 24 ▒ 24 ö	9 25 ▒ 25 Ü	. 26 ▒ 26 ü	EEX 27 ▒ 27 Æ	NEG 28 ▒ 28 æ	GTO ᵀ 29 ≠ 29 ≠	XEQ ᵀ 30 ▒ 30 £	W ᵀ 31 ▒ 31 ▒	1
2	RCL 00 32 32	RCL 01 33 ! 33 !	RCL 02 34 " 34 "	RCL 03 35 # 35 #	RCL 04 36 $ 36 $	RCL 05 37 % 37 %	RCL 06 38 & 38 &	RCL 07 39 ' 39 '	RCL 08 40 (40 (	RCL 09 41) 41)	RCL 10 42 * 42 *	RCL 11 43 + 43 +	RCL 12 44 , 44 ,	RCL 13 45 - 45 −	RCL 14 46 . 46 .	RCL 15 47 / 47 /	2
3	STO 00 48 0 48 0	STO 01 49 1 49 1	STO 02 50 2 50 2	STO 03 51 3 51 3	STO 04 52 4 52 4	STO 05 53 5 53 5	STO 06 54 6 54 6	STO 07 55 7 55 7	STO 08 56 8 56 8	STO 09 57 9 57 9	STO 10 58 : ▒ 58 :	STO 11 59 ; 59 ;	STO 12 60 < 60 <	STO 13 61 = 61 =	STO 14 62 > 62 >	STO15 63 ? 63 ?	3
4	+ 64 @ 64 @	− 65 A 65 A	* 66 B 66 B	/ 67 C 67 C	X<Y? 68 D 68 D	X>Y? 69 E 69 E	X≤Y? 70 F 70 F	Σ+ 71 G 71 G	Σ− 72 H 72 H	HMS+ 73 I 73 I	HMS− 74 J 74 J	MOD 75 K 75 K	% 76 L 76 L	%CH 77 M 77 M	P→R 78 N 78 N	R→P 79 O 79 O	4
5	LN 80 P 80 P	X↑2 81 Q 81 Q	SQRT 82 R 82 R	Y↑X 83 S 83 S	CHS 84 T 84 T	E↑X 85 U 85 U	LOG 86 V 86 V	10↑X 87 W 87 W	E↑X-1 88 X 88 X	SIN 89 Y 89 Y	COS 90 Z 90 Z	TAN 91 [91 [	ASIN 92 \ 92 \	ACOS 93] 93]	ATAN 94 ↑ 94 ↑	→DEC 95 _ 95 _	5
6	1/X 96 ᐩ 96 ᐩ	ABS 97 a 97 a	FACT 98 b 98 b	X≠0? 99 c 99 c	X>0? 100 d 100 d	LN1+X 101 e 101 e	X<0? A ▒ 102 f	X=0? B ▒ 103 g	INT C ▒ 104 h	FRC D ▒ 105 i	D→R E ▒ 106 j	R→D F ▒ 107 k	→HMS G ▒ 108 l	→HR H ▒ 109 m	RND I ▒ 110 n	→OCT J ▒ 111 o	6
7	CLΣ T ▒ 112 p	X<>Y Z ▒ 113 q	PI Y ▒ 114 r	CLST X ▒ 115 s	R↑ L ▒ 116 t	RDN M [▒ 117 u	LASTX N \ ▒ 118 v	CLX O] ▒ 119 w	X=Y? P ↑ ▒ 120 x	X≠Y? Q _ ▒ 121 y	SIGN ⊢ᵀ ▒ 122 z	X≤0? a ▒ 123 π	MEAN b ▒ 124 |	SDEV c ▒ 125 →	AVIEW d ⊏ 126 Σ	CLD e ⊦ 127 ⊢	7
	0 0000	1 0001	2 0010	3 0011	4 0100	5 0101	6 0110	7 0111	8 1000	9 1001	A 1010	B 1011	C 1100	D 1101	E 1110	F 1111	

00 01 02 03 04 05 06 07 08 09 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 ← bit numbers in a 7-byte register



HP-41C QUICK REFERENCE CARD FOR SYNTHETIC PROGRAMMING © 1982, SYNTHETIX

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	
8	DEG IND 00 128 ◆	RAD IND 01 129 ×	GRAD IND 02 130 x̄	ENTER↑ IND 03 131 ←	STOP IND 04 132 α	RTN IND 05 133 β	BEEP IND 06 134 Γ	CLA IND 07 135 ↓	ASHF IND 08 136 Δ	PSE IND 09 137 σ	CLRG IND 10 138 ◆	AOFF IND 11 139 λ	AON IND 12 140 μ	OFF IND 13 141 ∠	PROMPT IND 14 142 τ	ADV IND 15 143 Φ	8
9	RCL IND 16 144 θ	STO IND 17 145 Ω	ST+ IND 18 146 δ	ST− IND 19 147 Ȧ	ST * IND 20 148 ȧ	ST/ IND 21 149 Ä	ISG IND 22 150 ä	DSE IND 23 151 Ö	VIEW IND 24 152 ö	ΣREG IND 25 153 Ü	ASTO IND 26 154 ü	ARCL IND 27 155 Æ	FIX IND 28 156 æ	SCI IND 29 157 ≠	ENG IND 30 158 £	TONE IND 31 159 ▒	9
A	XR 0-3 IND 32 160	XR 4-7 IND 33 161 !	XR8-11 IND 34 162 "	X12-15 IND 35 163 #	X16-19 IND 36 164 $	X20-23 IND 37 165 %	X24-27 IND 38 166 &	X28-31 IND 39 167 '	SF IND 40 168 (	CF IND 41 169)	FS?C IND 42 170 *	FC?C IND 43 171 +	FS? IND 44 172 ,	FC? IND 45 173 −	GTO XEQ IND IND 46 174 .	SPARE IND 47 175 /	A
B	SPARE IND 48 176 0	GTO 00 IND 49 177 1	GTO 01 IND 50 178 2	GTO 02 IND 51 179 3	GTO 03 IND 52 180 4	GTO 04 IND 53 181 5	GTO 05 IND 54 182 6	GTO 06 IND 55 183 7	GTO 07 IND 56 184 8	GTO 08 IND 57 185 9	GTO 09 IND 58 186 :	GTO 10 IND 59 187 ;	GTO 11 IND 60 188 <	GTO 12 IND 61 189 =	GTO 13 IND 62 190 >	GTO 14 IND 63 191 ?	B
C	GLOBAL IND 64 192 @	GLOBAL IND 65 193 A	GLOBAL IND 66 194 B	GLOBAL IND 67 195 C	GLOBAL IND 68 196 D	GLOBAL IND 69 197 E	GLOBAL IND 70 198 F	GLOBAL IND 71 199 G	GLOBAL IND 72 200 H	GLOBAL IND 73 201 I	GLOBAL IND 74 202 J	GLOBAL IND 75 203 K	GLOBAL IND 76 204 L	GLOBAL IND 77 205 M	X<>-- IND 78 206 N	LBL -- IND 79 207 O	C
D	GTO -- IND 80 208 P	GTO -- IND 81 209 Q	GTO -- IND 82 210 R	GTO -- IND 83 211 S	GTO -- IND 84 212 T	GTO -- IND 85 213 U	GTO -- IND 86 214 V	GTO -- IND 87 215 W	GTO -- IND 88 216 X	GTO -- IND 89 217 Y	GTO -- IND 90 218 Z	GTO -- IND 91 219 [	GTO -- IND 92 220 \	GTO -- IND 93 221]	GTO -- IND 94 222 ↑	GTO -- IND 95 223 _	D
E	XEQ -- IND 96 224 ᐩ	XEQ -- IND 97 225 a	XEQ -- IND 98 226 b	XEQ -- IND 99 227 c	XEQ -- IND100 228 d	XEQ -- IND101 229 e	XEQ -- IND102 230 f	XEQ -- IND103 231 g	XEQ -- IND104 232 h	XEQ -- IND105 233 i	XEQ -- IND106 234 j	XEQ -- IND107 235 k	XEQ -- IND108 236 l	XEQ -- IND109 237 m	XEQ -- IND110 238 n	XEQ -- IND111 239 o	E
F	TEXT 0 IND T 240 p	TEXT 1 IND Z 241 q	TEXT 2 IND Y 242 r	TEXT 3 IND X 243 s	TEXT 4 IND L 244 t	TEXT 5 INDM[245 u	TEXT 6 IND N \ 246 v	TEXT 7 INDO] 247 w	TEXT 8 IND P↑ 248 x	TEXT 9 INDQ_ 249 y	TEXT10 IND ⊢ᵀ 250 z	TEXT11 IND a 251 π	TEXT12 IND b 252 |	TEXT13 IND c 253 →	TEXT14 IND d 254 Σ	TEXT15 IND e 255 ⊢	F
	0 0000	1 0001	2 0010	3 0011	4 0100	5 0101	6 0110	7 0111	8 1000	9 1001	A 1010	B 1011	C 1100	D 1101	E 1110	F 1111	

For price information and a list of dealers in your area, send a self-addressed stamped envelope to: SYNTHETIX, 1540 Mathews Ave., Manhattan Beach, CA 90266, USA

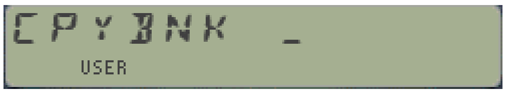
Copying code from bank-switched ROMS. { **CPYBNK** }

There are almost no tools available to extract or copy code from a bank switched ROM. When faced with that challenge I typically used ad-hoc modifications of Warren Furlow's routine **CB**, posted at: http://www.hp41.org/LibView.cfm?Command=View&ItemID=317

That routine is specific for fixed source and destination pages, as well as only useful for the second bank. Writing a more general-purpose function was always on my mind, and finally here it is at last. Obviously to be successful the destination must be a Q-RAM (MLDL or CL).

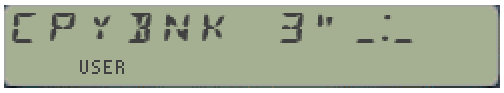
CPYBNK is a prompting function. It has a customized prompt with three distinct sections that are shown on the screen as the data entry progresses. The parameters entered are as follows:

- Bank number, an integer decimal from 1 to 4
- Source page, an hex value from 0 to F
- Destination page, same as above.

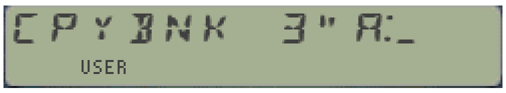


```
CPYBNK _
USER
```

The function is smart enough to know what the first prompt must be, thus it'll simply ignore non-allowed values, presenting the same prompt again. You can use the back-arrow key to cancel at any moment. Once the bank number is entered the prompt requests the "FROM:TO" pages, as denoted by the underscore characters on both sides of the colon. The screens below show this at different stages of the process:



```
CPYBNK 3" _:_
USER
```

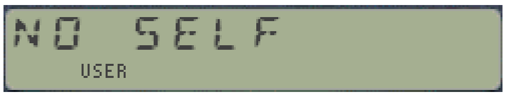


```
CPYBNK 3" A:_
USER
```

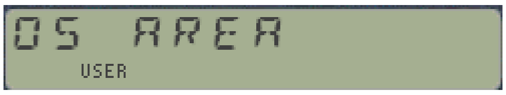
The copy is always made into the main bank of the destination page (bank-1). This is typically a Q-RAM page in an MLDL (or a RAM page on the CL) thus only supports one bank. Besides the practical usage is intended to copy elusive, hard-to-reach code buried into secondary banks – therefore it wouldn't appear very sensible to copy it into equally obscure destinations.

The main bank is the first one; therefore you can use "1" to select it. In this case the function does the same as **CPYPGE** in the PowerCL, or **COPYROM** in the HEPAX module..

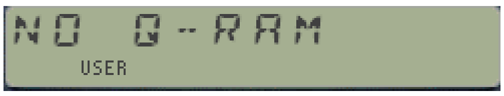
CPYBNK is also clever enough to exclude its own banks either as source or destination pages. This is needed to avoid the copy process colliding with the execution of its MCODE. Furthermore, it'll always prevent the O/S area as destination, and it will perform a Q-RAM test to ensure the destination page is write-enabled.



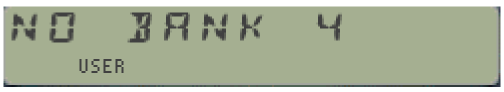
```
NO SELF
USER
```



```
OS AREA
USER
```



```
NO Q-RAM
USER
```



```
NO BANK 4
USER
```

Lastly, if the source ROM doesn't have the chosen bank an error message is shown and the execution aborts. More than just a convenient feature, this is vital to ensure that the execution doesn't activate a non-existing bank – which could create all kinds of havoc if the location of the missing bank is already occupied in RAM or FLASH by other modules. See below the error messages for these conditions:

New X-Mem File Pointer Functions.

A few new record pointer functions are included to complement the original set from the Extended Functions module. The intent was to facilitate the operation of the Equation Library FOCAL programs, saving some steps here and there and providing more flexibility in their use.

The functions are shown on the table below:-

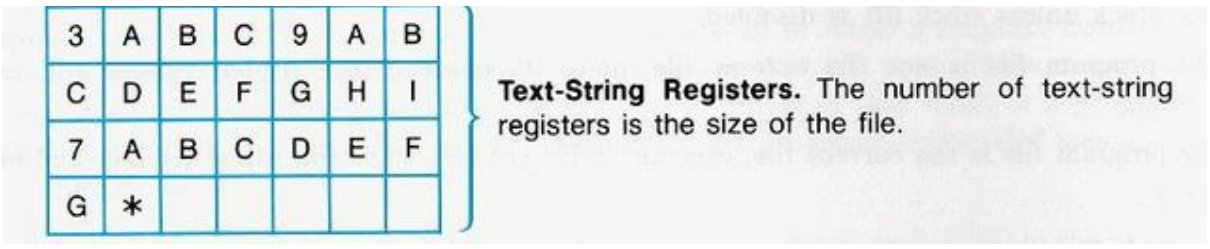
Function	Description	Input	Output
REC-	Move pointer one position down	none	Pointer moved
REC+	Move pointer one position up	None	Pointer moved
REC+X	Advance Pointer in Record	Number of positions in X	Pointer is moved

The pointer functions mostly deal with updating the file header location where the pointer position is saved. They verify that the chosen position is within the boundaries of the ASCII file and adjust it accordingly. See the File Header diagram below for details:

T	A	D	R	-	C	H	R	R	E	C	S	Z	E
13	12	11	10	9	8	7	6	5	4	3	2	1	0

An interesting challenge arises because the ASCII file records are of variable length, so there isn't a constant number of characters per record. This is handled by reading the record-length nybble, located at the beginning of each record.

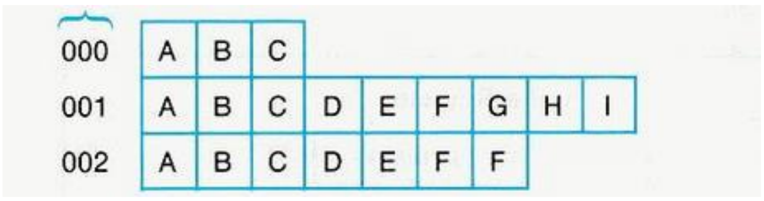
Shown below is a text file of three records: ABC, ABCDEFGHI, and ABCDEFG. At the start of each record is an extra byte indicating the length of each record, which can be up to 254 chars long. The "*" at the end of the file indicates the end of the current contents of the file.



3	A	B	C	9	A	B
C	D	E	F	G	H	I
7	A	B	C	D	E	F
G	*					

Text-String Registers. The number of text-string registers is the size of the file.

Which is shown in the Editor as this (record length and last char bytes are invisible):



000	A	B	C						
001	A	B	C	D	E	F	G	H	I
002	A	B	C	D	E	F	F		

For comparison purposes the standard approach used by the original X-Functions always requires recalling the pointer first using **RCLPT(A),** adding or subtracting the number of positions using the stack, and resetting the pointer using **SEEKPT(A).** This alters the stack registers and requires multiple steps per action – as opposed to using new pointer functions, with a more straightforward method. for example, **REC+** is functionally equivalent to (but has none of the shortcomings of):

RCLPT(A), INT , 1 , + , SEEKPT(A)

If you're interested in the details go ahead and check the MCODE listing below.

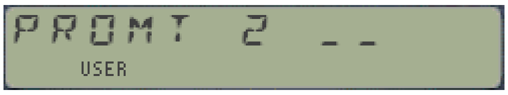
Header	A41E	0AB	"+"	PT to Next Record
Header	A41F	043	"C"	FName in ALPHA
Header	A420	045	"E"	
Header	A421	052	"R"	Ángel Martin
REC+	A422	108	SETF 8	
	A423	033	JNC +06	[MERGE]
Header	A424	0AD	"-"	PT to Previous Record
Header	A425	043	"C"	FName in ALPHA
Header	A426	045	"E"	
Header	A427	052	"R"	Ángel Martin
REC-	A428	104	CLRF 8	
MOVPT3	AA09	03E	B=0 MS	uses current file!
	AA0A	249	?NC XQ	
	AA0B	0F0	->3C92	[CURFLT]
	AA0C	0B0	C=N ALL	A(10:8) - 1 rg addr (name)
	AA0D	0FC	RCR 10	N(12:10) - add file header
	AA0E	270	RAMSLCT	select file header addr
	AA0F	038	READATA	read FLDH value
	AA10	106	A=C S&X	put file size in A.X
	AA11	1A6	A=A-1 S&X	REC# is zero-based
	AA12	03C	RCR 3	move rec# to C.X
	AA13	10C	?FSET 8	next?
	AA14	03B	JNC +07	no, skip
NEXT	AA15	366	?A#C S&X	is SZE=REC# ?
	AA16	063	JNC +12d	yes, do nothing
	AA17	306	?A<C S&X	is SZE<REC# ?
	AA18	057	JC +10d	yes, do nothing
	AA19	226	C=C+1 S&X	increase record
	AA1A	023	JNC +04	[RESTORE]
PREV	AA1B	2E6	?C#0 S&X	zero record?
	AA1C	033	JNC +06	yes, skip over
	AA1D	266	C=C-1 S&X	decrease record
RESTORE	AA1E	03C	RCR 3	put byte# in C.X
	AA1F	046	C=0 S&X	clear byte#
	AA20	13C	RCR 8	rotate back
	AA21	2F0	WRTDATA	update header
IGNORE	AA22	046	C=0 S&X	
	AA23	270	RAMSLCT	select chip0
	AA24	3AD	PORT DEP:	
	AA25	08C	GO	
	AA26	3C7	->AFC7	[RTN3]

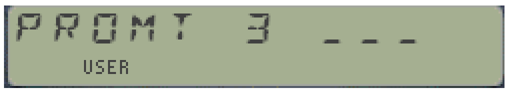
The final call to [RTN3] gives the show away: this routine is located in the third bank of the WARP_Core, and as such the code needs to switch back to the main bank before yielding to the OS.

Other Utilities

PROMT – General Prompting   X: number of prompts

PROMT _ is a general-purpose, direct HEX entry function. The number of Hex digits to enter is provided at its own prompt in manual mode, or in the X register if used in a program. The result is placed in X as a binary number – with as many valid digits as the number of fields in the prompt.


PROMT 2 _ _
USER


PROMT 3 _ _ _
USER

For example, to prepare the RAM location "60FF" using **PROMT**, you first enter "4" at the function prompt and then the four hex digits of the address directly. The result is placed in the X register ready for the byte functions to use.

Note: This function is very similar to **HPROMPT,** included both in the HEPAX and the Hepax Dis-Assembler Modules.

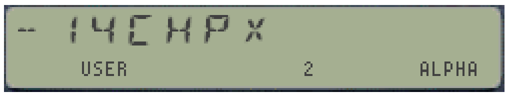
arning: you should be aware that these functions use the X register for scratch

ALPHB - Alphabetizing ALPHA   ALPHA: Text
A<>A – ALPHA Reversal   ALPHA: Text

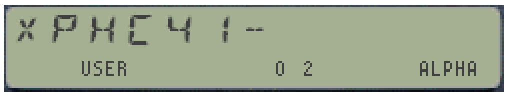
Never too late for exciting ALPHA routines - **ALPHB** and **A<>A** are utility functions written by Poul Kaarup. Use them to reverse or sort the contents of the ALPHA register alphabetically, either in descending (UF 00 clear) or ascending (UF 00 set) order. A neat little example of utilization of the standard OS routines, make sure you don't miss it!

Example: sort ALPHA alphabetically when its contents is "HP-41CX"

CF 00, WF$ "ALPHB"


- 14CHPX
USER 2 ALPHA

SF 00, WF$ "ALPHB"


XPHC41-
USER 0 2 ALPHA

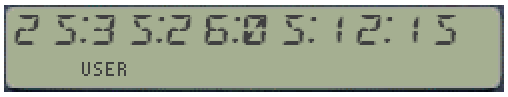
Example: Reverse the contents of ALPHA with the text "RECURSION"

WF$ "A<>A"   => "NOISERUCER"

IOBUS – Bus enumeration Prompt: 1, 2, 3

IOBUS is a shrank-down version of the function with same name available in the PowerCL_Extreme module. It scans the complete I/O bus (i.e. the ROM pages) looking for pages matching the selected criteria. In this version the choices are limited to four: (0) empty pages – a.k.a BFREE, (1) used pages – a.k.a. BUSED, (2) bank-switched pages – a.k.a BANKED, and (3) current ROM id# list (a.k.a. ROMLST). Any other input will trigger the "DATA ERROR" message.
.

- **ROMLST** produces a list in Alpha with the XROM id#'s of the plugged modules on the system, so you can check for dups. Because of the 24-char limit in the Alpha string, only the last 8 modules will be shown – sufficient in the majority of cases, especially considering that pages 3, 4, and 5 are most likely unique because of being dedicated to the X-Functions, the Library#4, and the Time Module.

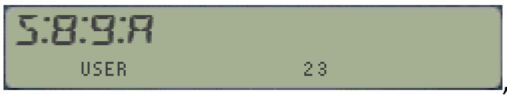


```
25:35:26:05:12:15
USER
```

Example: winning Lotto combination or ROM list?

- **BANKED** presents a colon-separated string of numbers (in hex) corresponding to those pages with a bank-switched configuration, as defined in the ROM signature characters. The official convention is not strictly followed by the (very few) authors of the few bank-switched ROMs, but the number of banks should be marked in characters 2/3/4 of the ROM signature.

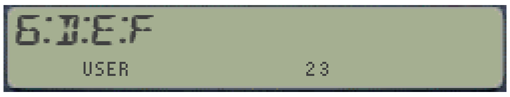
  An example with both the PowerCL and the SandMath_4x4 plugged returns the following:-
  Can you explain the presence of the "5"? Hurry, time's ticking out!



```
5:8:9:A
USER              23
```

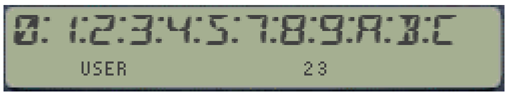
,

- **BFREE** and **BUSED** will present colon-separated strings of hex numbers corresponding to those free or used pages in the calculator. Obviously the OS will always be listed by **BUSED**, which is a nice clue to quickly tell which particular string you're looking at. See for instance the examples bellow showing a pretty decent configuration:



```
6:D:E:F
USER              23
```

for the free pages, and



```
0:1:2:3:4:5:7:8:9:A:B:C
USER              23
```

for the used pages.

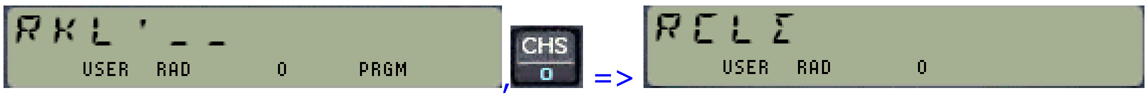
  The strings are compiled using the display and transferred to ALPHA upon completion. For full-house configurations the list of used pages will take up more characters than those allowed in the display – and the string will be scrolled to the left, dropping the first three pages in the worst case. Since those hold the OS (always there) there's no real information loss.

  The strings can have "holes", as this is totally dependent on the modules plugged. Some of them use the upper part of the port (like the Zenrom), or just simply due to the physical locations used.

ST<>Σ - Bulk ΣREG Access	Prompt: First Data Register

These three utilities (**ST<>Σ**, **STOΣ** and **RCLΣ**) come very handy to exchange or access the contents of the statistical registers and the stack all at once. The XYZTL stack registers are used as data source or destination respectively.

Note also that there's a quick shortcut to RCLΣ from the main **RKL** prompt – using the CHS key.



A<>ST – ALPHA and Stack Exchange	No Input
A<>RG - ALPHA and Data Regs Swap	Prompt: First Data Register
ST<>RG – Stack and Register Exchange	Prompt: First Data Register

Even though these functions are included in other modules they fully fit the "Total Rekall" theme and therefore are added to the WARP_Core as well. Not much to write home about but a handy and effective way to manage the data across the Stack, ALPHA, and Data Registers.

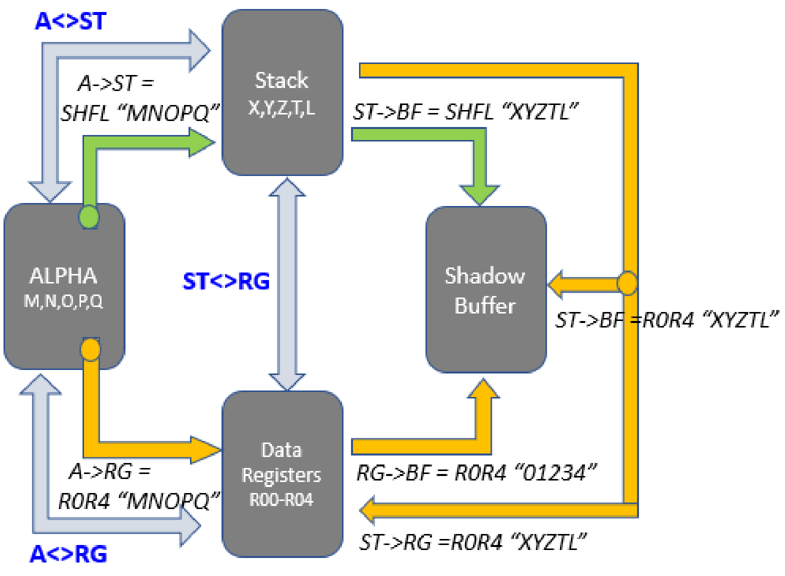
FWIW here's again the diagram showing the overlap between these direct data exchange functions compared with the Selected scheme in the Shadow buffer:

A<>RG and **ST<>RG** are prompting functions, allowing both Stack and INDirect arguments. You need to enter the first of the five data Registers block used for the exchange, as follows:

T <-> Rn
Z <-> R(n+1)
Y <-> R(n+2)
X <-> R(n+3)
L <-> R(n+4)

And:

M <-> Rn
N <-> R(n+1)
O <-> R(n+2)
P <-> R(n+3)
Q <-> R(n+4)

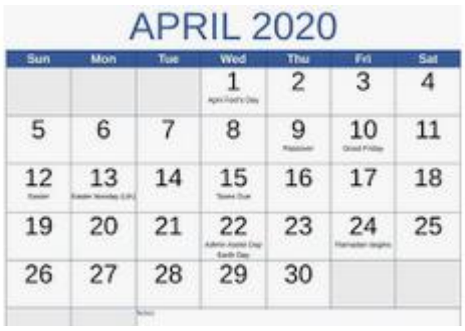


EASTER - Easter Sunday Date. X: Date in current format

A classic amongst calculator aficionados, this super-fast MCODE version was written by Kari Pasanen. Simply enter the year in X and call **EASTER** to see it replaced with the date of Easter Sunday in the current date format (either MDY or DMY). X is saved in LastX (and EASTER's sub-function code will be saved in the LastF buffer as well).

For example, for 2020

MDY => 4.122020, i.e. April 12th.
DMY => 12.042020



APRIL 2020

Sun	Mon	Tue	Wed	Thu	Fri	Sat
			1	2	3	4
5	6	7	8	9	10	11
12	13	14	15	16	17	18
19	20	21	22	23	24	25
26	27	28	29	30		

BFVIEW - Buffer Contents Viewer X: Buffer id#

Do you feel the urge to inspect the contents of the LAST-7 buffer, or any other one for that matter? Then you're in luck, using **BFVIEW** that's just a quick sub-function call away: enter the buffer id# and issue the call in your favorite way either with **WF#**, **WF$** or **XEQ$** (the Chef's recommendation of course). The Buffer registers will be shown sequentially with a small pause in-between each.

Here's a complete buffer id# table for your reference:

Buffer id#	Module / EPROM	Reason
1	David Assembler	MCODE Labels already existing
2	David Assembler	MCODE Labels referred to
3	Eramco RSU-1B	ASCII data pointers
4	Eramco RSU-1A	Data File pointers
5	**CCD Module, Advantage**	Seed, Word Size, Matrix Name
6	**Formula Evaluation**	Operands and Operators
	Extended IL (Skwid)	Accessory ID of current device
7	Extended IL (Skwid)	Printing column number & Width
	WARP Core, Formula Eval	Shadow Stack
8	**41Z Module**	Complex Stack and Mode
9	**SandMath, PowerCL, WARP …**	Last Function data, LAST-five
10	Time Module	Alarms Information
11	**HP-16C Emulator**	64-bit Data and stack
	Plotter Module	Data and Barcode parameters
12	IL-Development; CMT-200	IL Buffer and Monitoring
13	CMT-300, FORTH-41	Status Info, FORTH Code
14	Advantage, SandMath	INTEG & SOLVE scratch
15	Key Assignments	

PGCAT - Page Catalog No Input

A real must-have, for those still not using the AMC_OS/X (say what?), **PGCAT** is also included for your convenience. This is the best way to sequentially enumerate the ROMS plugged in your system only showing the ROM header function (first one in the FAT). It shows "NO ROM" when blank, and "NO FAT" for pages with a proper id# but no functions in them (such as blank HEPAX RAM pages for instance). You can press and hold any key (other than R/S) at any time to pause the enumeration.`

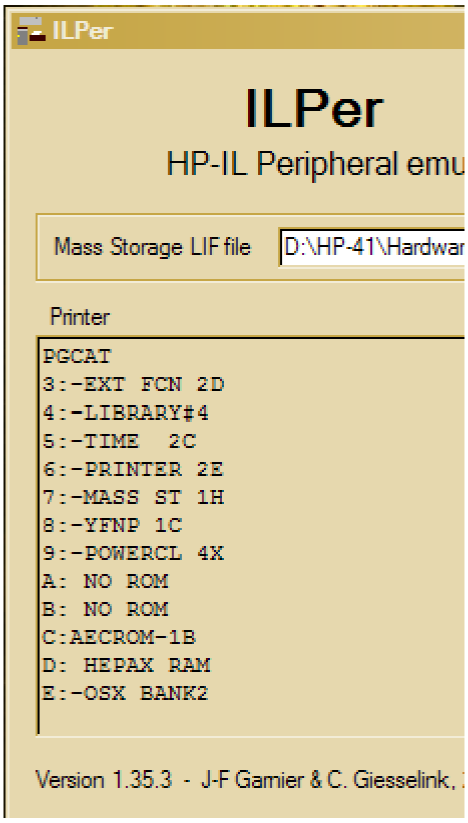
PGCAT is taken from the HEPAX Module (called **BCAT** there, within the HEPAX sub-functions group) - and written by Steen Petersen. **PGCAT** enumerates the first function of each page, starting with page 3. The enumeration can be stalled pressing any key other than R/S or ON, but the individual functions won't be listed.

PGCAT Lists the first function of every ROM block (i.e. Page), starting with Page 3 in the 41 CX or Page 5 in the other models (C/CV). The listing will be printed if a printer is connected and user flag 15 is enabled.

- Non-empty pages will show the first function in the FAT, or "NO FAT" if such is the case
- Empty pages will show the "NO ROM" message next to their number.
- Blank RAM pages will also show "NO FAT", indicating their RAM-in-ROM character.

No input values are necessary. This function doesn't have a "manual mode" (using [**R/S**]) but the displaying sequence will be halted while any key (other than [**R/S**] or [**ON**]) is being depressed, resuming its normal speed when it's released again.

See on the right the printout output from **PGCAT** using J-F Garnier's PIL-Box and the ILPER PC program, showing a nice traceability of the pressed keys:



```
PGCAT
3:-EXT FCN 2D
4:-LIBRARY#4
5:-TIME  2C
6:-PRINTER 2E
7:-MASS ST 1H
8:-YFNP 1C
9:-POWERCL 4X
A: NO ROM
B: NO ROM
C:AECROM-1B
D: HEPAX RAM
E:-OSX BANK2
```

XROM$ - XROM Function Decoder ALPHA: User Prog Name

Written by Klaus Huppertz this function was published in PRISMA magazine, April 1990. **XROM$_** prompts for a FOCAL program name (or gets it from ALPHA in a running program) and scans the program code looking for all the XROM calls included in it, showing either the section header for the module if it's plugged in, or the XROM function id# if the module is not present. The listing is sequential, and you need to press [SST] to see the next match (any other key will terminate the function). It's therefore very handy to find out the XROM dependencies of your FOCAL code. Note however that it will not work on FOCAL programs loaded in plug-in ROMs.

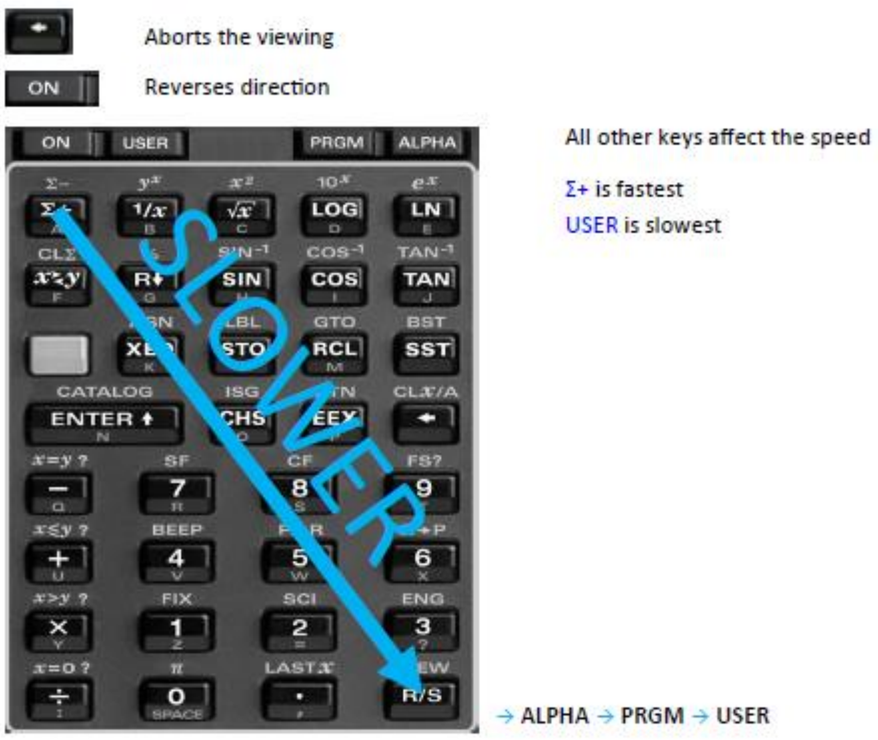
CSST -View Program   ALPHA: User Prog Name

CSST sequentially displays the program steps of the program pointed at by the Program Counter (PC). It's equivalent to using the **SST** key multiple times, and thus its name.

This function is programmable, operating in single-step mode or in back-step mode depending on whether the user flag 0 is cleared or set. The back-arrow key terminates the display of program lines, yielding to normal keyboard operation in RUN mode, or transferring control to the running FOCAL program that executed **CSST.** In the latter case, the program execution resumes with the currently displayed line (i.e. it has moved the program pointer).

The [**ON**] key toggles between single-step operation to back-step operation and vice-versa. The "**0**" annunciator is visible in the display whenever back-step operation is in effect.

The delay between lines shown can be adjusted by pressing any keyboard key, see the table below taken from the original article in PPCJ V9N7 p49 (refer there for further details). To use it, position first the PC at the target location (using GTO or similar). Note that the display time increases linearly from the top key down to the bottom key in a given key column on the keyboard, and continues to increase on the next column to the right. Furthermore (staying on the same row of keys), pressing the key one column right of the selected key will roughly double the selected display time.



Multiple Stack Register Tests. { **X=YZ?** , **X=YZT?** }

These functions are derivatives of the original X=Y? test. They provide further control to your program flow choices by allowing second and third chance options when the first condition X=Y? is false – thus they're not different from X=Y? when the such condition is true.

Their behavior is defined below – note that the testing is done sequentially, starting with X,Y, then X,Z, and finally X,T. The rule of thumb is *"one program line will be skipped each time a test is false, and until the first true one is found (it any)"*

X=YZ? X EQUAL TO Y OR Z

Tests if X is equal to Y or Z, with the following possible results:

- No program lines are skipped if X=Y
- Skips one program line if first test (X=Y) is false but the second (X=Z) is true
- Skips two program lines if second test (X=Z) is also false

X=YZT? X EQUAL TO Y, Z OR T

Tests if X is equal to Y,Z or T, with the following results:

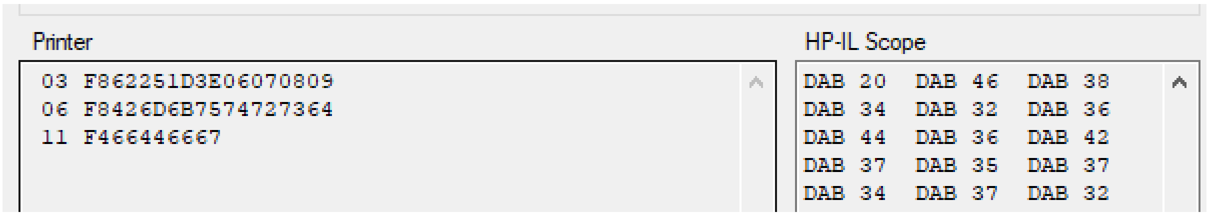
- No program lines are skipped if X=Y
- Skips one program line if first test (X=Y) is false but the second (X=Z) is true
- Skips two program lines if first & second tests (X=Y, X=Z) are false but the third is true.
- Skips three program lines if the three tests (X=Y, X=Z, and X=T) are false

DETEXT – Decoding Synthetic Text Lines

ALPHA: Program NAME

This function scans a user code program looking for synthetic text lines and prints the byte codes in the peripheral HP-IL printer when found. The Program Name must be in ALPHA. If ALPHA is empty, then the current program will be decoded.

The program does a good job discerning between "normal" text lines and those with non-keyable characters (on the original machine, that is). Note that it must reside in RAM memory, thus you'd need first to COPY it when it resides in a plug-in module.



```
03 F862251D3E06070809
06 F8426D6B7574727364
11 F466446667
```

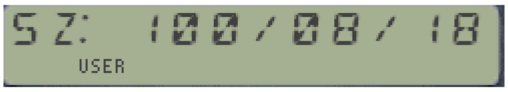
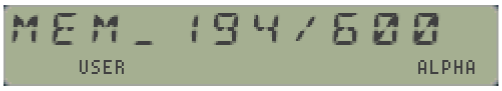
```
DAB 20   DAB 46   DAB 38
DAB 34   DAB 32   DAB 36
DAB 44   DAB 36   DAB 42
DAB 37   DAB 35   DAB 37
DAB 34   DAB 37   DAB 32
```

DETEXT was contributed by Ross Wentworth and first published in PPCCJ V12N1 p34. Unfortunately, the original article did not include the Hex codes for the MCODE instructions so it took a bit longer than needed to reconstruct…

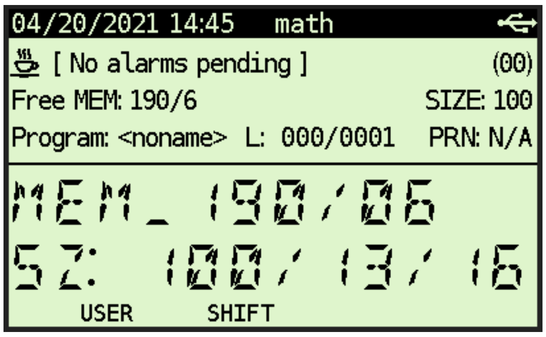
?MEM – Resource Viewer

Ever wondered how many data registers are available for programs, or how big the I/O buffers section is at a given time? Those two and a few more are easy to find out with ?MEM, which produces two screens with the following information:

- Size screen, showing the space used by Data Regisdters, I/O Buffers, and Key Assignments
- Memory screen, showing available regs in Main and X-Memory


```
SZ:  100/08/ 18
  USER
```
;

```
MEM_ 194/600
  USER                ALPHA
```

Note that the Memory screen remains in ALPHA, whereas the Size screen is displayed in the LCD. This is important as its contents will be lost when you switch ALPHA ON to access the Memory screen. Obviously on the DM-41X this limitation doesn't exist, as you can see below:


```
04/20/2021 14:45   math
[ No alarms pending ]                (00)
Free MEM: 190/6                SIZE: 100
Program: <noname>  L: 000/0001  PRN: N/A
MEM_ 190/06
SZ:  100/ 13/ 16
   USER     SHIFT
```

METRON - Keeping the Beat X: Beats per minute

Last but not least, here's a cool beat-keeping utility written by Mark Power (see DataFile V9N4 p18) to help with your musical chores. Use it as a metronome replacement for your out-of-town rehearsals, just enter the beat in X and call **METRON** .

Note that the time module is required for proper operation.

Appendix. - Internal Data Field structure for Extended Prompts. -

There are five main categories that support the extended prompting facility, as follows:
(1) Register Swaps; (2) RKL & RCL Math; (3) Comparison Tests; (4) Select/Case Support; and (5) Double INDirection. All of them share the same main core code that provides support for INDirect addressing, Stack registers or the combination of both - therefore it's important to have the input data structured in such a way that is compatible with all use cases. The different requirements for the function execution are summarized below:

- RCL Math needs descriptors for the type of arithmetic operation and source data register
- Register Swaps needs descriptors for source and destination registers
- Comparison tests need descriptors for the operator and source and destination regs
- Select/Case functions need descriptors of the #S variable and the operator
- Double Indirection needs descriptors for RLC/STO, multiplicity order, and source register

To be able to use the same core code, all these must be arranged in a common scheme that is compatible with all functions. Besides, it needs to survive calls to partial key entry sequences (that overwrite the Status bits), and cannot utilize system flags 3 and 4 – used to signal running program and PRGM mode entry conditions.

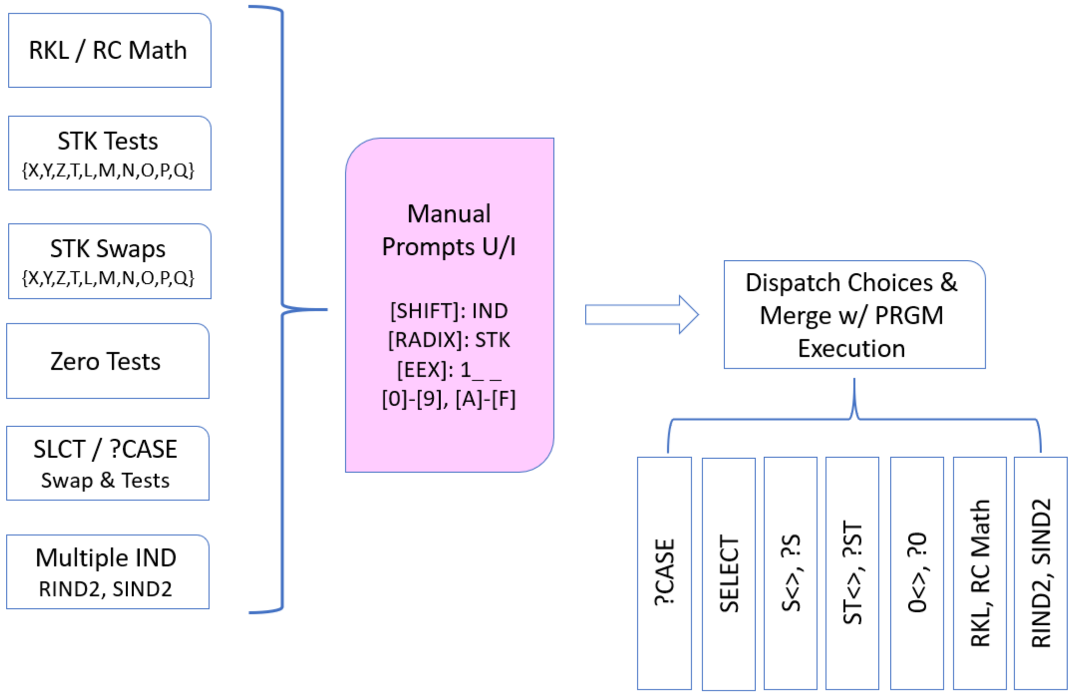
The table below describes such arrangement, which basically uses all status bits, the C.MS and C<12> digits, The Mantissa and S&X for function address and hot-key tables. The fields are stored in the "Data Configuration Register", which is populated by the routines and saved in the stack register 9(Q) as temporary repository. - *See footnote (*)*

Besides those the code also borrows F11 and F12 temporarily to single case special cases such as AIRCL, ?CASE, and SELCT – always taking good care to restore their default values upon termination.

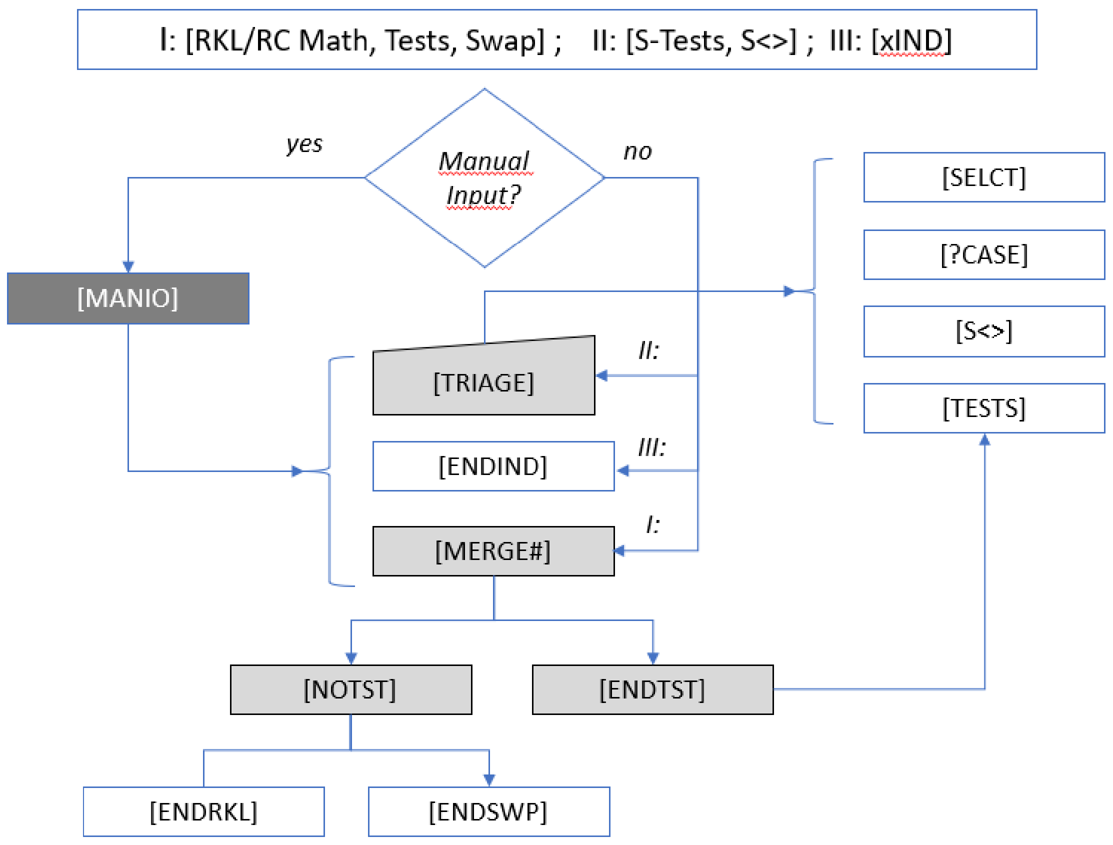
Register Swaps	RCL Math	Comparison Tests	Select/Case	Multiple INDirection
`C<12> digit is cleared			C<12> digit = "F"	C<12> digit = 2, 4, 6,8…
	Hot-key Table address in [S&X] field; Function Address in [ADR] field.			
All Flags configured in Q<7:8> field				
Clears F0		Sets F0	F0 set: <> case	n/a
n/a	F1 set: Main RKL	F1 set: "#" case	F1 set: "#" case	
F2 Clear	Sets F2	F2 set: "=" case	F2 set: "=" case	
F3 set: PRGM data entry				
F4 set: SST execution; (F13: program running)				
n/a	F5 set: RCL+	F5 set: ">" case	F5 set: ">" case	n/a
	F6 set: RCL^	F6 set: "<=" case	F6 set: "<=" case	
	F7 set: RCL*	F7 set: "<" tests	F7 set: "<" tests	F7 set: SIND2 case
F8 Used by [BCDBIN]		All Clear: ">=" case	All Clear: ">=" case	F8 Used by [FNCTXT]
Register Swaps	**RCL Math**	**Comparison Tests**	**Select/Case**	**Multiple INDirection**
C.MS holds Stack Reg# ; C<12> holds flag for category				
[MS] =0: T-Register	n/a	[MS] =0: T-Register		n/a
[MS] =1: Z-Register		[MS] =1: Z-Register		
[MS] =2: Y-Register		[MS] =2: Y-Register		
[MS] =3: X-Register		[MS] =3: X-Register		
[MS] =4: L-Register		[MS] =4: L-Register		
[MS] =5: M-Register		[MS] =5: M-Register		
[MS] =6: N=Register		[MS] =6: N=Register		
[MS] =7: O-Register		[MS] =7: O-Register		
[MS] =8: P-Register		[MS] =8: P-Register		
[MS] =9: Q-Register		[MS] =9: Q-Register		
[MS = A-F FOR {a-e}		[MS = A-F FOR {a-e}		

(*) Q-Note: The Q-register contents is overwritten during the Stack Comparison in Manual mode; therefore, they're meant to be used in running programs only.

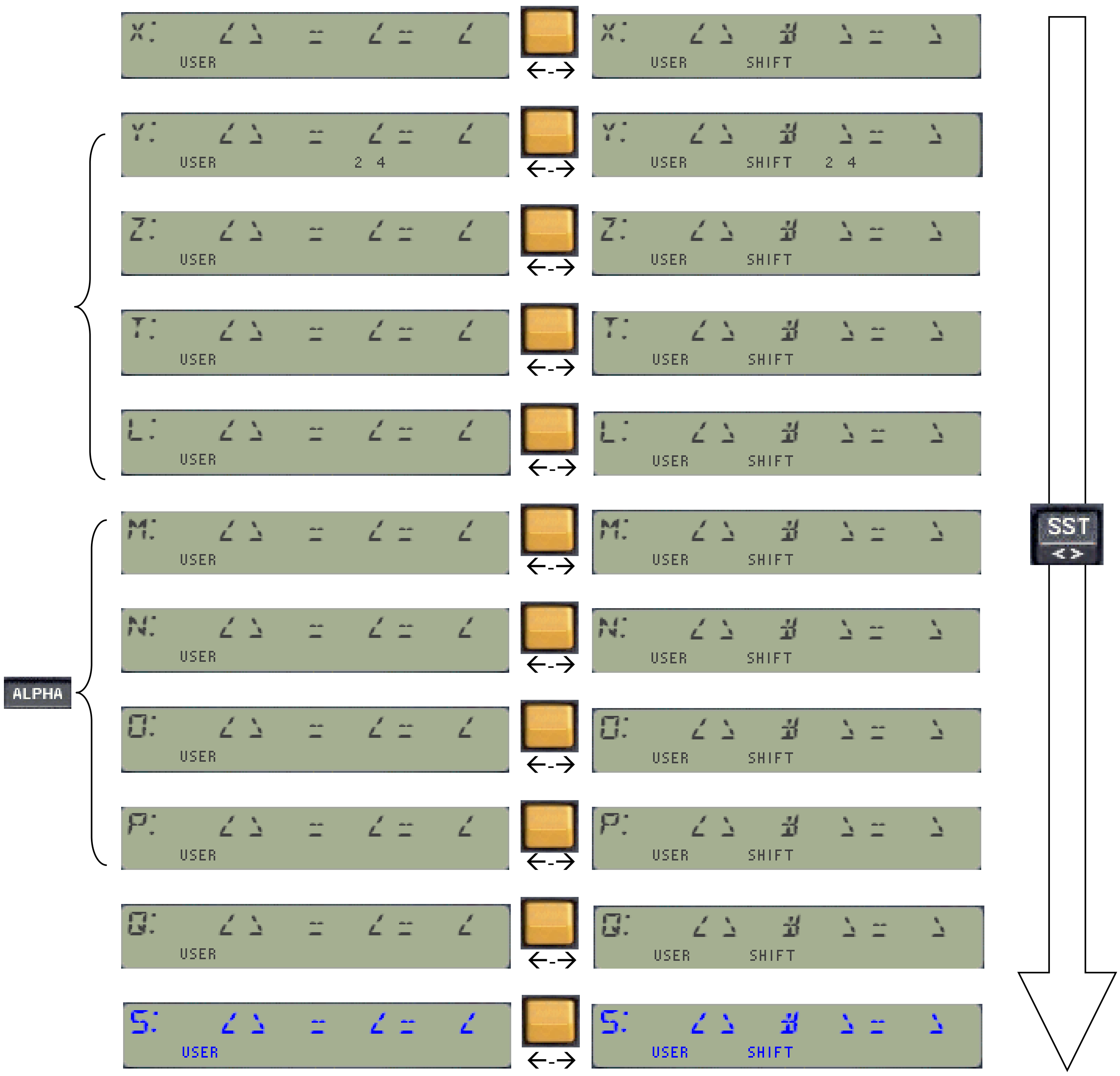
The block diagram below shows the top-level structure of the workflow:

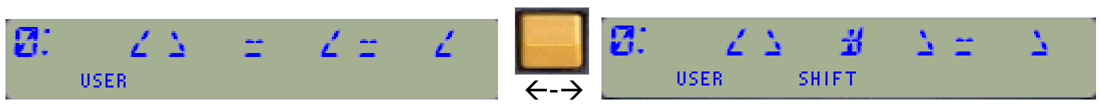

RKL / RC Math
STK Tests {X,Y,Z,T,L,M,N,O,P,Q}
STK Swaps {X,Y,Z,T,L,M,N,O,P,Q}
Zero Tests
SLCT / ?CASE Swap & Tests
Multiple IND RIND2, SIND2
Manual Prompts U/I
[SHIFT]: IND
[RADIX]: STK
[EEX]: 1_ _
[0]-[9], [A]-[F]
Dispatch Choices & Merge w/ PRGM Execution
?CASE
SELECT
S<>, ?S
ST<>, ?ST
0<>, ?0
RKL, RC Math
RIND2, SIND2

Once the manual U/I interface parses the choices the work is not done; it now needs to dispatch the execution to the appropriate function branch, which is also determined by looking at the status bits and configuration data. This is tricky but all needed information is contained in the input variables, providing reliable operation. The diagram below shows the logic between the involved subroutines:



Appendix.- Dare to Compare: 84 functions at your fingertips !





If "Zero" is the foster child, then the selected variable is the surrogate stack member!

Q-Note: The Q-register contents is overwritten during the Stack Comparison <u>in Manual mode</u>; therefore, they're meant to be used in running programs only.