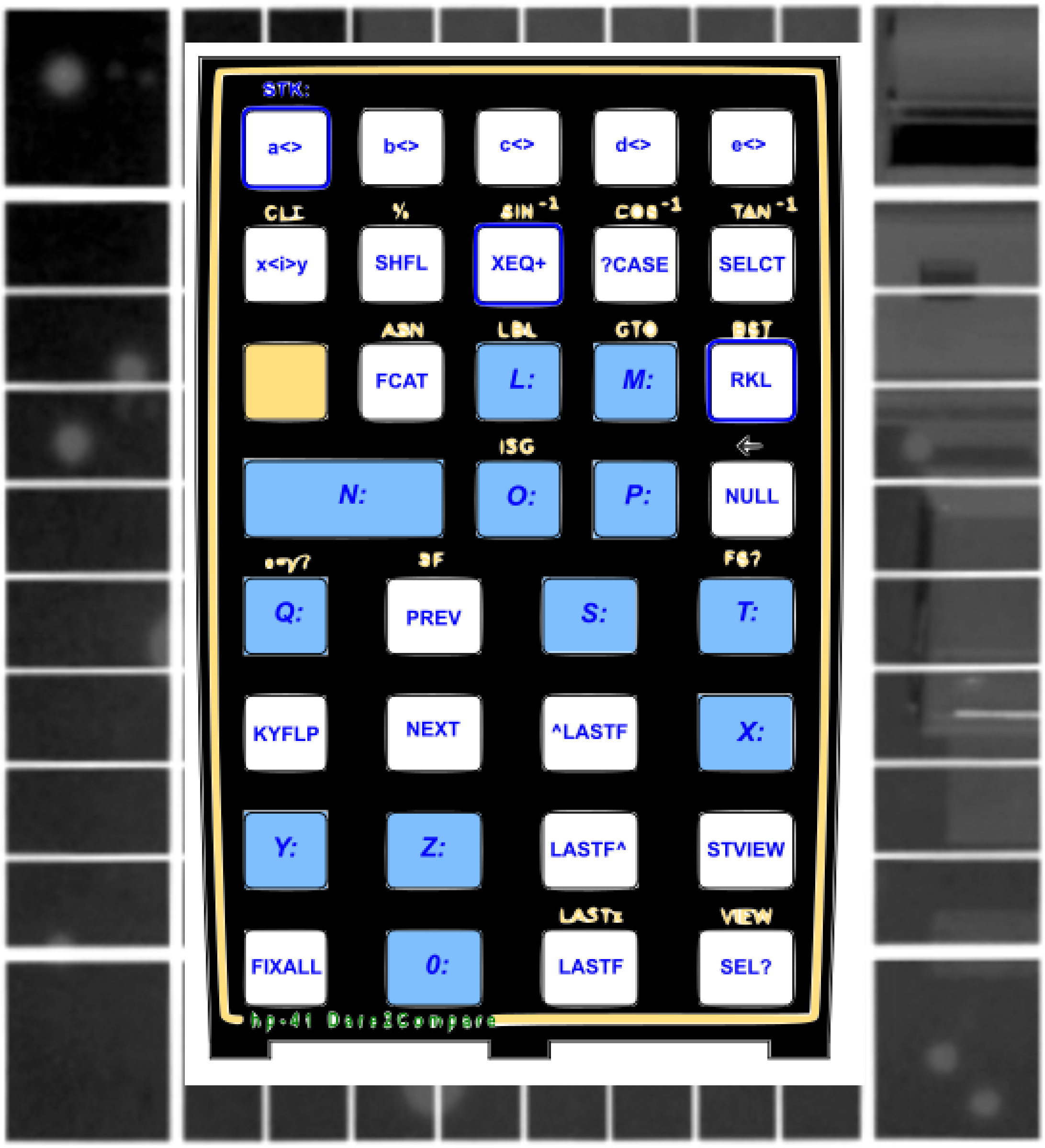
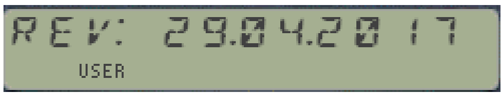
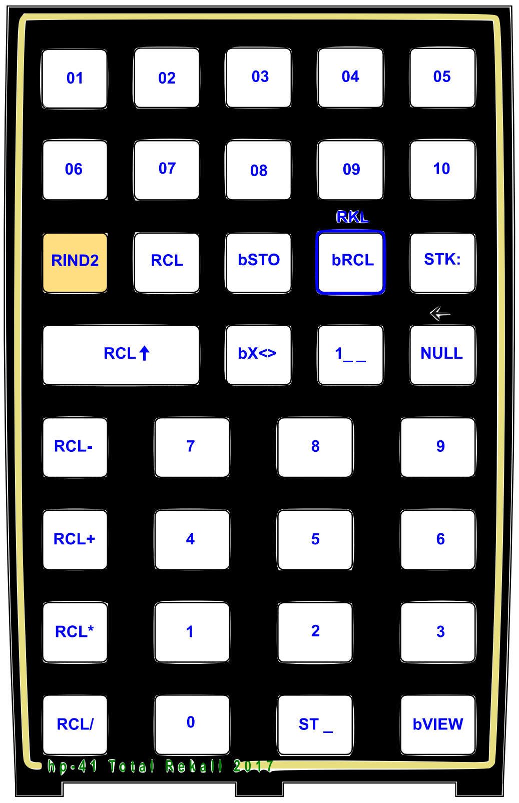
The Total_Rekall 2017 Module
"Dare to Compare" Revision

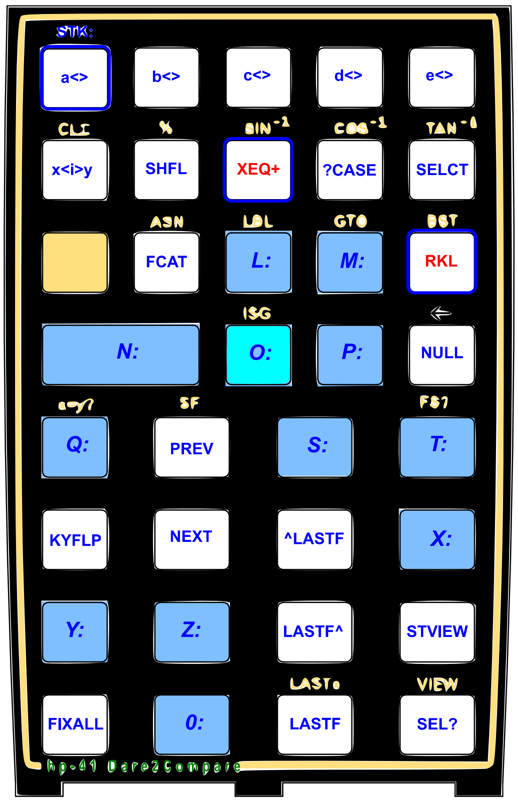
RCL Math and Full Stack Tests for the HP-41
Including FixALL mode for accurate number display
& Auto-Complete Advanced XEQ Mode





Written and programmed by Ángel Martin
April 29, 2017





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.

Summary Function Table.

#	Function	Description	Input	Dependency	Type	Author
0	**-TTL REKALL**	*Lib#4 Check & Splash*	none	Lib#4	MCODE	*Ángel Martin*
1	**A<>RG _ _**	Alpha Exchange	RG# in prompt / Next Line	Lib#4	MCODE	*Ken Emery*
2	**XEQ+**	**Auto-Complete Model**	**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	**S<> _ _**	Swap Selected & Target Regs	Target Reg in prompt	Lib#4	MCODE	*Ángel Martin*
12	**SELCT _**	selects variable	RG# in prompt / Next Line	Lib#4	MCODE	*Ángel Martin*
13	**SHFL _ _ _ _ _**	Stack Shuffle	five stack regs in prompt	Lib#4	MCODE	*Ángel Martin*
14	**SIND2 _ _**	STO IND IND (IND ...)	RG# in prompt / Next Line	Lib#4	MCODE	*Ángel Martin*
15	**ST<>RG _ _**	Stack Exchange	RG# in prompt / Next Line	Lib#4	MCODE	*Ángel Martin*
16	**XF# _ _ _**	*Sub-function Launcher by index*	Index at the prompt	Lib#4	MCODE	*Ángel Martin*
17	**XF$ _**	*Sub-function Launcher by Name*	Name in prompt	Lib#4	MCODE	*Ángel Martin*
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	**Q<> _ _**	Swap Q and register	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< _ _**	Less than Zero test	RG# in prompt / Next Line	Lib#4	MCODE	*Ángel Martin*
31	**?0<= _ _**	Less than or Equal to Zero Test	RG# in prompt / Next Line	Lib#4	MCODE	*Ángel Martin*
32	**?0> _ _**	Greater than Zero Test	RG# in prompt / Next Line	Lib#4	MCODE	*Ángel Martin*
33	**?0>= _ _**	Greater 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< _ _**	Less than X test	RG# in prompt / Next Line	Lib#4	MCODE	*Ángel Martin*
37	**?X<= _ _**	Less than or equal to X test	RG# in prompt / Next Line	Lib#4	MCODE	*Ángel Martin*
38	**?X> _ _**	Greater than X Test	RG# in prompt / Next Line	Lib#4	MCODE	*Ángel Martin*
39	**?X>= _ _**	Greater 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< _ _**	Less than Y test	RG# in prompt / Next Line	Lib#4	MCODE	*Ángel Martin*
43	**?Y<= _ _**	Less than or equal to Y test	RG# in prompt / Next Line	Lib#4	MCODE	*Ángel Martin*
44	**?Y> _ _**	Greater than Y Test	RG# in prompt / Next Line	Lib#4	MCODE	*Ángel Martin*
45	**?Y>= _ _**	Greater 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< _ _**	Less than Z test	RG# in prompt / Next Line	Lib#4	MCODE	*Ángel Martin*
49	**?Z<= _ _**	Less than or equal to Z test	RG# in prompt / Next Line	Lib#4	MCODE	*Ángel Martin*
50	**?Z> _ _**	Greater than Z Test	RG# in prompt / Next Line	Lib#4	MCODE	*Ángel Martin*
51	**?Z>= _ _**	*Greater 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*
53	**?T# _ _**	Different from T test	RG# in prompt / Next Line	Lib#4	MCODE	*Ángel Martin*

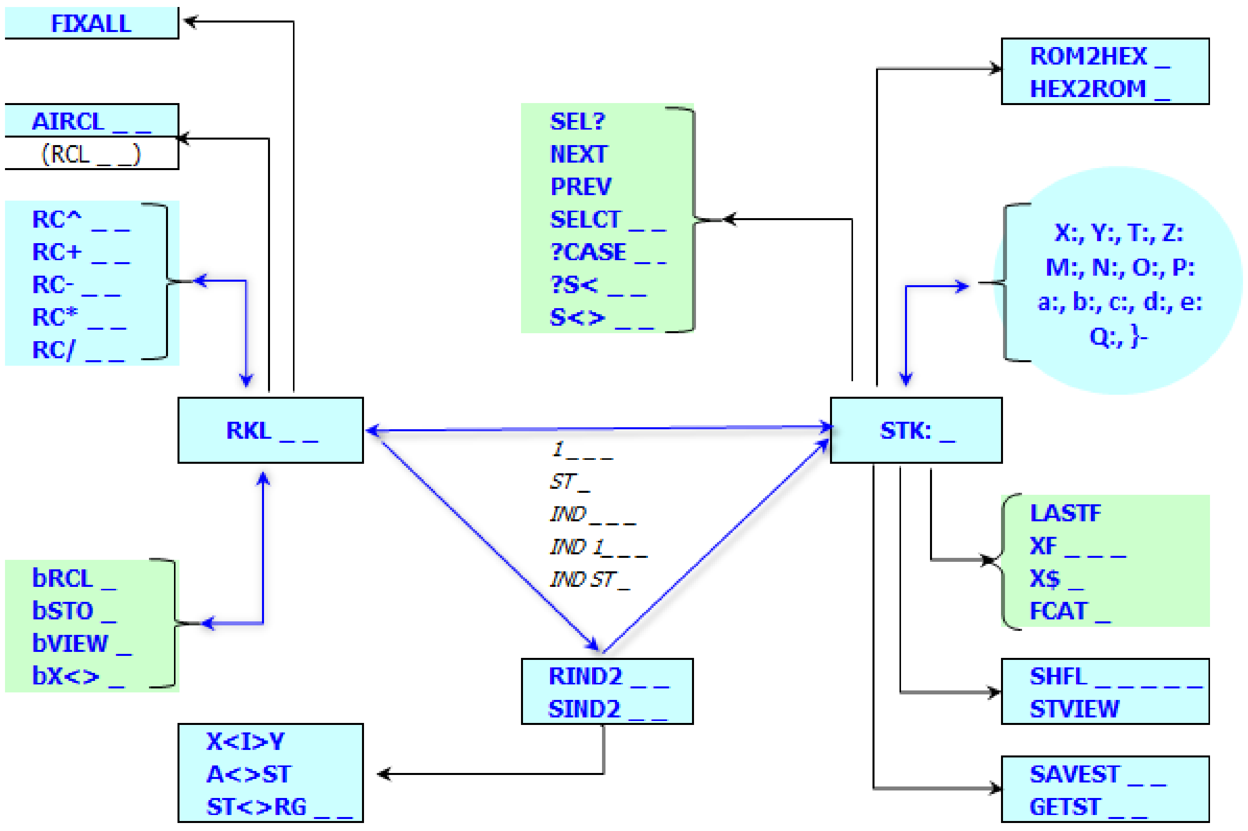
#	Function	Description	Input	Dependency	Type	Author
54	?T< _ _	Less than T test	RG# in prompt / Next Line	Lib#4	MCODE	Ángel Martin
55	?T<= _ _	Less than or equal to T test	RG# in prompt / Next Line	Lib#4	MCODE	Ángel Martin
56	?T> _ _	Greater than T Test	RG# in prompt / Next Line	Lib#4	MCODE	Ángel Martin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

This module also includes a 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	?M= _ _	Equal to M test	RG# in prompt / Next Line	Lib#4	MCODE	Ángel Martin
8	?M# _ _	Different from M test	RG# in prompt / Next Line	Lib#4	MCODE	Ángel Martin
9	?M< _ _	Less than M test	RG# in prompt / Next Line	Lib#4	MCODE	Ángel Martin
10	?M<= _ _	Less than or equal to M test	RG# in prompt / Next Line	Lib#4	MCODE	Ángel Martin
11	?M> _ _	Greater than M Test	RG# in prompt / Next Line	Lib#4	MCODE	Ángel Martin
12	?M>= _ _	Greater than/equal to M test	RG# in prompt / Next Line	Lib#4	MCODE	Ángel Martin
13	?N= _ _	Equal to N test	RG# in prompt / Next Line	Lib#4	MCODE	Ángel Martin
14	?N# _ _	Different from N test	RG# in prompt / Next Line	Lib#4	MCODE	Ángel Martin
15	?N< _ _	Less than N test	RG# in prompt / Next Line	Lib#4	MCODE	Ángel Martin
16	?N<= _ _	Less than or equal to N test	RG# in prompt / Next Line	Lib#4	MCODE	Ángel Martin
17	?N> _ _	Greater than N Test	RG# in prompt / Next Line	Lib#4	MCODE	Ángel Martin
18	?N>= _ _	Greater than or equal to N test	RG# in prompt / Next Line	Lib#4	MCODE	Ángel Martin
19	?O= _ _	Equal to O test	RG# in prompt / Next Line	Lib#4	MCODE	Ángel Martin
20	?O# _ _	Different from O test	RG# in prompt / Next Line	Lib#4	MCODE	Ángel Martin
21	?O< _ _	Less than 0 test	RG# in prompt / Next Line	Lib#4	MCODE	Ángel Martin
22	?O<= _ _	Less than or equal to 0 test	RG# in prompt / Next Line	Lib#4	MCODE	Ángel Martin
23	?O> _ _	Greater than 0 Test	RG# in prompt / Next Line	Lib#4	MCODE	Ángel Martin
24	?O>= _ _	Greater than or equal to 0 test	RG# in prompt / Next Line	Lib#4	MCODE	Ángel Martin
25	?P= _ _	Equal to P test	RG# in prompt / Next Line	Lib#4	MCODE	Ángel Martin
26	?P# _ _	Different from P test	RG# in prompt / Next Line	Lib#4	MCODE	Ángel Martin
27	?P< _ _	Less than P test	RG# in prompt / Next Line	Lib#4	MCODE	Ángel Martin
28	?P<= _ _	Less than or equal to P test	RG# in prompt / Next Line	Lib#4	MCODE	Ángel Martin
29	?P> _ _	Greater thanP Test	RG# in prompt / Next Line	Lib#4	MCODE	Ángel Martin
30	?P>= _ _	Greater than or equal to P test	RG# in prompt / Next Line	Lib#4	MCODE	Ángel Martin
31	?Q= _ _	Equal to Q test	RG# in prompt / Next Line	Lib#4	MCODE	Ángel Martin
32	?Q# _ _	Different from Q test	RG# in prompt / Next Line	Lib#4	MCODE	Ángel Martin
33	?Q< _ _	Less than Q test	RG# in prompt / Next Line	Lib#4	MCODE	Ángel Martin
34	?Q<= _ _	Less than or equal to Q test	RG# in prompt / Next Line	Lib#4	MCODE	Ángel Martin
35	?Q> _ _	Greater than Q Test	RG# in prompt / Next Line	Lib#4	MCODE	Ángel Martin
36	?Q>= _ _	Greater than or equal to Q test	RG# in prompt / Next Line	Lib#4	MCODE	Ángel Martin
37	-XTRA FNS	Shows Revision date	none	Lib#4	MCODE	Ángel Martin
38	A<>ST	Exchange Alpha & Stack	Values in ALPHA and stack	Lib#4	MCODE	Ángel Martin
39	bRCL _	Buffer reg recall	buffer reg# (1-5)	Lib#4	MCODE	Ángel Martin
40	bSTO _	Buffer reg Storage	buffer reg# (1-5)	Lib#4	MCODE	Ángel Martin

41	**bVIEW**	Buffer Reg View	Buffer reg# (1-5)	Lib#4	MCODE	*Ángel Martin*
42	**bX<> _**	Buffer Reg Exchange	buffer reg# (1-5)	Lib#4	MCODE	*Ángel Martin*
43	**FIXALL**	Activates Fix ALL mode	none	Lib#4	MCODE	*Ángel Martin*
44	**KYFLP _**	Flips Key assignments	Pressed key	Lib#4	MCODE	*Ángel Martin*
45	**^LASTF _**	**Prompts for FName to add**	**Buffer #9**	Lib#4	MCODE	*Ángel Martin*
46	**LASTF^**	**Starts LastF review**	**Hot keys, Buffer #9**	Lib#4	MCODE	*Ángel Martin*
47	**RTN?**	Tests for pending RTNs	YES/NO, skips if False	Lib#4	MCODE	*Doug Wilder*
48	**RTNS**	Number of pending RTNs	Pust in X, Lifts Stack	Lib#4	MCODE	*Ángel Martin*
49	**SVIEW**	Shows S register contents	None	Lib#4	MCODE	*Ángel Martin*
50	**STVIEW**	Full Stack View	None	Lib#4	MCODE	*Ángel Martin*
51	**X<I>Y**	Exchange IND(X) & IND(Y)	Values in X, Y	Lib#4	MCODE	*Nelson F. Crowle*
52	**DSNEX**	Decrement and Skip if not Equal	Value in X	Lib#4	MCODE	*Ángel Martin*
53	**ISLEX**	Increment and Skip if Equal	Value in X	Lib#4	MCODE	*Ángel Martin*
54	**FINDX**	Find register containing X	Value in X	Lib#4	MCODE	*Ángel Martin*
55	**NEXT**	increment selection	SEL variable	Lib#4	MCODE	*Ángel Martin*
56	**PREV**	decrement selection	SEL variable	Lib#4	MCODE	*Ángel Martin*
57	**SEL?**	Shows the selected variable	SEL variable	Lib#4	MCODE	*Ángel Martin*
58	**SVIEW**	Shows Selected var contents	Value in sel var.	Lib#4	MCODE	*Ángel Martin*
59	**?S=**	Equal to S test	Data in sel and target	Lib#4	MCODE	*Ángel Martin*
60	**?S#**	Different from S test	Data in sel and target	Lib#4	MCODE	*Ángel Martin*
61	**?S<**	Less than S test	Data in sel and target	Lib#4	MCODE	*Ángel Martin*
62	**?S< _ _**	Less than or equal to S test	Data in sel and target	Lib#4	MCODE	*Ángel Martin*
63	**?S> _ _**	Greater than S Test	Data in sel and target	Lib#4	MCODE	*Ángel Martin*
64	**?S>= _ _**	Greater than or equal to S test	Data in sel and target	Lib#4	MCODE	*Ángel Martin*
65	**FCAT _**	Sub-function CATALOG	has HOT keys	Lib#4	MCODE	*Ángel Martin*

Figure 0: Interaction between the different function launchers.



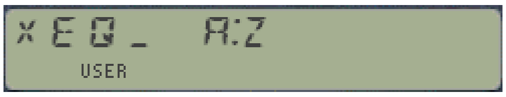
What’s new in the 2017 edition? The Auto-complete mode.

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.

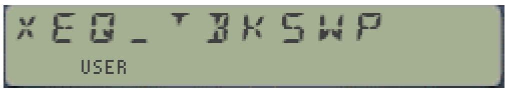
When you call the **XEQ+** function a new mode of execution opens up to the user; one where instead of spelling the complete function name at the alpha prompt, just the first initial letter is entered and calculator does the rest for you – with a few control hot-keys to navigate the complete bus, from page #3 up to the top in page #F.

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 criteria to keep the code size and execution times within reasonable parameters.

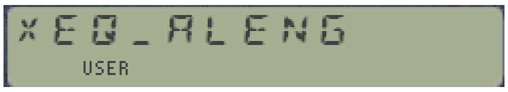
In short: the function **XEQ+** starts a new mode by prompting for an initial character letter or number, A-Z, a-e, 0-9. After that selection is made, after a short search time (negligible on the CL for sure) it presents all functions currently available in the bus that begin with that letter - commencing 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. Both MCODE functions and FOCAL programs will be shown:



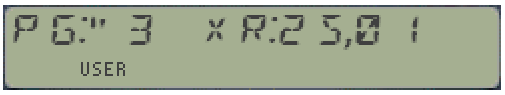
;



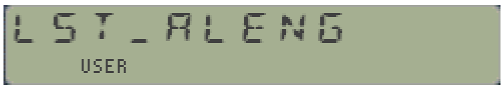
Once you've locked up your target function simply press XEQ to execute it, or ASN to assign it to the key of your choice. Or if you’re not sure this is your choice (say duplicates or similarly spelled ones), pressing RCL will show you some vital signs of the function, such page# and XROM id#

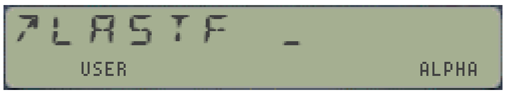


->

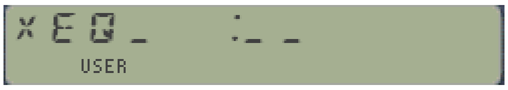


The operation will also add the executed function automatically to the enhanced **LASTF** facility, which now will hold up to five entries (say, LastF now stands for “last-five”? ;-). Two new functions allow the user to review and execute these **(LASTF^)**, plus a manual mode to enter functions into the list if so desired (**^LASTF)**.

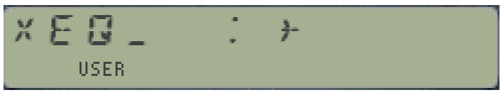




Lower case characters (a-e), numbers and all other key-able special chars are accessed using the shifted keys in the standard ALPHA keyboard. Another option is provided to use special characters – even if not key-able but allowed in function names. This makes it possible to search for function names staring with “μ”, the forwards or backwards goose, or all the little men just to name a few.



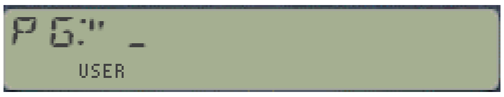
‘2E” =>



Note that the functions are not listed in alphabetical order, but in sequential order as they're found in the respective FAT's of the modules plugged in the calculator. The only restriction is that they all start with the letter chosen at the initial prompt.

If you'd rather see an automatic 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.

If no function starting with the selected letter exists the initial prompt will be shown again for a news selection. The same 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:



In case you wonder, page#4 is simply skipped over, while pages 0-1-2 when chosen default to page#3 – so this functionality does *not* include the standard functions of the calculator (such as BEEP, SDEV, etc.)

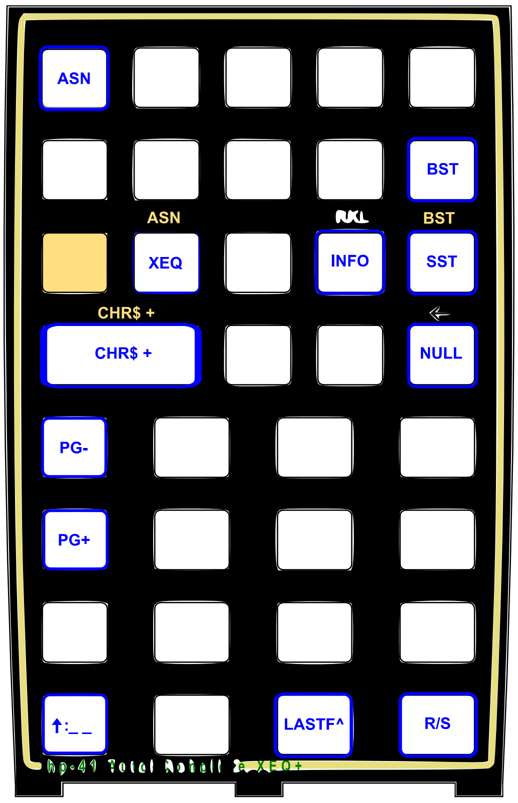
Overlays and Underlays.

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

The picture on the right 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 also active as a hot key to revert to the original XEQ function. Use it if you want to revert to the standard OQ method to spell the function name in ALPHA mode, simply press ALPHA twice and then spell the name as usual.



Seeing is believing: try it out and chances are soon it'll become one of your favorites. A real keeper!

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 page for details.
- Added a secondary FAT with 64 new sub-functions, amongst them all test functions on the stack registers {M-Q} – to complement {T to L} which were already there 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 in 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**
- Implements the "emergency storage buffer" with five 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 also included using zero as description in the strings.

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.

Notes: To make all these additions and enhancements possible it was needed to remove the UMS (Unit Management System) from the previous version 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. { **FCAT** }

FCAT 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 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 (or entering its index at the **XF#** prompt) is always required. This can become annoying if you want to repeatedly execute a given sub- function. The **LAST Function** implementation certainly minimizes this issue for repeat executions of the last sub-function called, without a dedicated key assignment required.

Launchers and Last Function functionality. { **XF#** , **XF$** }

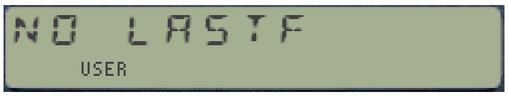
This module includes full support for the "LASTF" functionality. This is a handy choice for repeat executions of the same function (i.e. to execute again the last-executed function), without having to type its name or navigate the different launchers to access it. The implementation is not universal – 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:

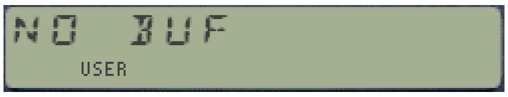
Module	**Launchers**	**LASTF Method**
Dare2Compare	**-STKT _**	Captures (sub)fnc id#
	RKL _ _	Captures (sub)fnc id#
	XF$ _	Captures fnc NAME
	XF# _ _ _	Captures (sub)fnc id#
	FCAT (XEQ')	Captures (sub)fnc id#

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

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*** and **RC/**. 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.

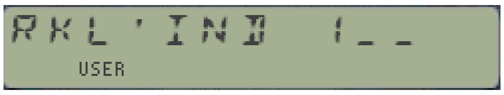
To solve the stack addressing one needs to resort to heavier wizardry, basically writing extra code to replace the OS handling of the prompting in these functions – which is based on the PTEMP bits of the function name. The custom prompting is therefore completely under the control of the function, and 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.

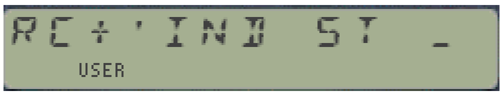
Once the technique was developed it was relatively easy to apply to other functions, like the stack exchange and comparison tests – if you can you envision instructions like: “Y<> IND M”, or “Z<=N?” to give just two examples. Unfortunately, the Library#4 was already full, so the subroutines are only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

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).

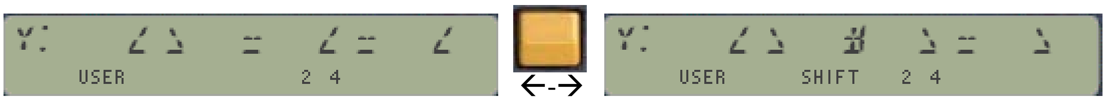
Lastly, you can revert to the native RCL pressing the [XEQ] key again at its prompt. When you do this in program mode the standard OS is used for efficient line entering of the standard cases, i.e. RCL 27 in a single program step as opposed to using the non-merged approach. More on this subject later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

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 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. <u>Just be careful with these</u>!!

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 "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.

Reversed Logic RPN?

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.

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	135	IND 07

Table 1: Register index mapping.

A few exceptions to the rule.

A few functions in the module do not allow stack arguments in their prompts. These functions are **ARCLI, 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.

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

```
01  LBL "STSRT          06  ?Z> (T)            11  Y<> (Z)
02  X>Y?                07  Z<> (T)            12  X>Y?
03  X<>Y                08  X>Y?               13  X<>Y
04  ?Y> (Z)             09  X<>Y               14  END
05  Y<> (Z)             10  ?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 the 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

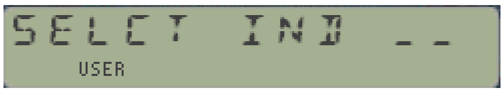
Register	Exchange	Tests
Q	Main	Sub-fcn
|-	Sub-fcn	Indirect
a	Sub-fcn	Indirect
b	Sub-fcn	Indirect
c	Sub-fcn	Indirect
d	Sub-fcn	Indirect
e	Sub-fcn	Indirect
"0"	n/a	Main
Rnn	Main	Indirect

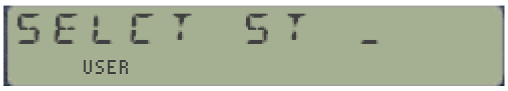
Lastly, non-stack Data Register swapping is missing from this set, but it's not forgotten - it's the subject of the next sections.

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.





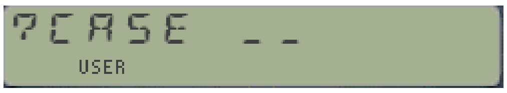
In program mode the variable for SELCT and the comparison value for ?CASE will be introduced as non-merged lines in program step following the main function – which is consistent with the other functions seen before that use the same schema. Note that comparison values are positive integers only.

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).

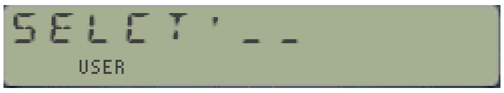
As the question mark would suggest, **?CASE** is a typical test function that will follow the "do if true / skip if false" rules when running in a program – or show the familiar "YES/NO" in manual mode.

*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!*

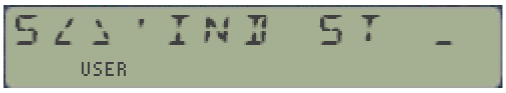
General-Purpose Exchange with SELCT / S<>

In a parallel implementation to the previous subject, you can also use the **SELCT** schema combined with the main 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 ;-)



Defines the selected variable-register



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 the selected variable.

If you're not sure of 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 number to the display (but not to the X-register).

- **SEL?** shows the value currently selected. If no selection has been mede the value shown is 4,095. 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.

Note that, like **?CASE** described before, if no register variable has been previously selecter then the exchange will use the X register as a default – and in that instance the number returned if you press [R/S] at the prompt will be 4,095.

Increasing and Decreasing the selected variable.

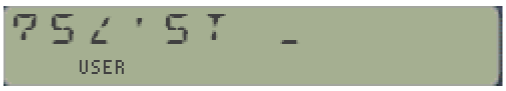
These sub-functions are related to the variable selected by SELCT, as follows:

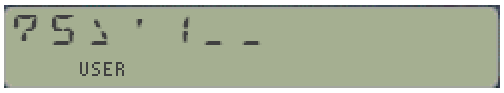
- **NEXT** and **PREV** increment and decrement the selected variable by one. No decrement will occur if the selection is R00. No changes will be made if no selection exists (which defaults to Stack "X"). These functions are very useful during program control for sequential access to different registers as selected variables.

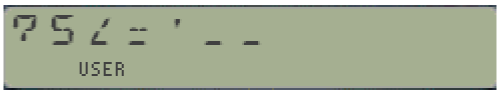
Remark that NEXT/PREV have effect on the register number stored in the buffer header (i.e. the "S" variable), but not on the register contents. Also that if an indirect or stack register is selected then the next/previous value is dictated by the "natural" register sequence, i.e. Stack_L comes after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

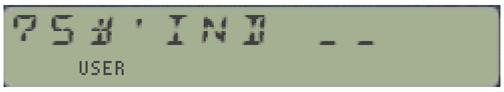
Value Comparison tests with selected variable.

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!).









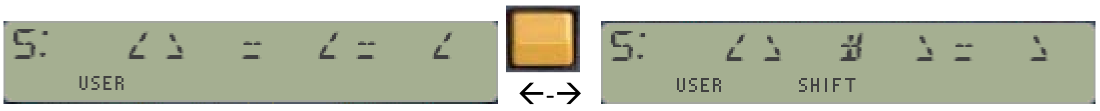
Note that the equal-to comparison **?S=** is different from **?CASE**; in both instances it is the content of the selected register what gets used as first value, but the second value differs: in the equal-to case it is the content of the target register being compared, whereas for ?CASE the comparison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

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  XF#                       03  XF#
04  58                        04  60
05  4        ?S< 04           05  4      ?S> 04
06  Yes                       06  yes
07  No                        07  no
```

The surrogate Stack Register "S".

All the variable comparison functions, as well as the exchange 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.

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  XF#     (?S>= IND X)     use the reverse test and a
17  66                       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  (0)
08  ENTER^
09  ISG X
10  GTO 00
11  RTN
12  LBL 00
13  XF#    (?S<= IND X)
14  64
15  243
16  GTO 00
17  S<>    (IND X)
18  243
19  LBL 00
20  XF#    (NEXT)
21  59
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.eeeii , 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)    select ind(bbb)
11  242
12  XF#   (ISLEX)    bbb+1
13  57
14  RTN              all done if (bbb+1) > eee
15  LBL 00
16  ...
```

And similarly in SRTALL2:

```
06  LBL 01
07  SELCT (0)
08  ENTER^
09  XF#  (ISLEX)     bbb+1
10  57
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 in the table below:

CX-Function	X=NN?	X#NN?	X<NN?	X<=NN?	X>NN?	X>=NN?
TotalRekall	?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: { nn.0kk, **REGSWAP** }
- **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	10,025	10, ENTER^
S<> 25	**REGSWAP**	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.

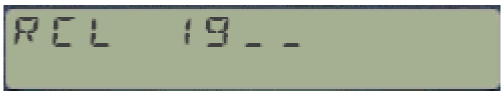
For these two implementations, the second byte of the RCL is *added to the same instruction in a program*, i.e. RCL 111 will be displayed as "RCL J", and similarly RCL 127 will show as "RCL e". This is clearly more efficient in byte usage; however *it does not support the RCL arithmetic operations* allowed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

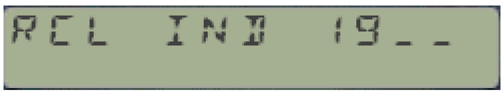
Note that the OS/X Prompt lengthener is only triggered with the standard OS-provided functions, and therefore won't appear at the custom prompt offered by "RKL _ _" or "RIND2 _ _"; nor by the ZENROM's after you have pressed the EEX key, i.e. "RCL 1_ _". Pressing the ON key in those instances will just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


RCL IND 19_ _

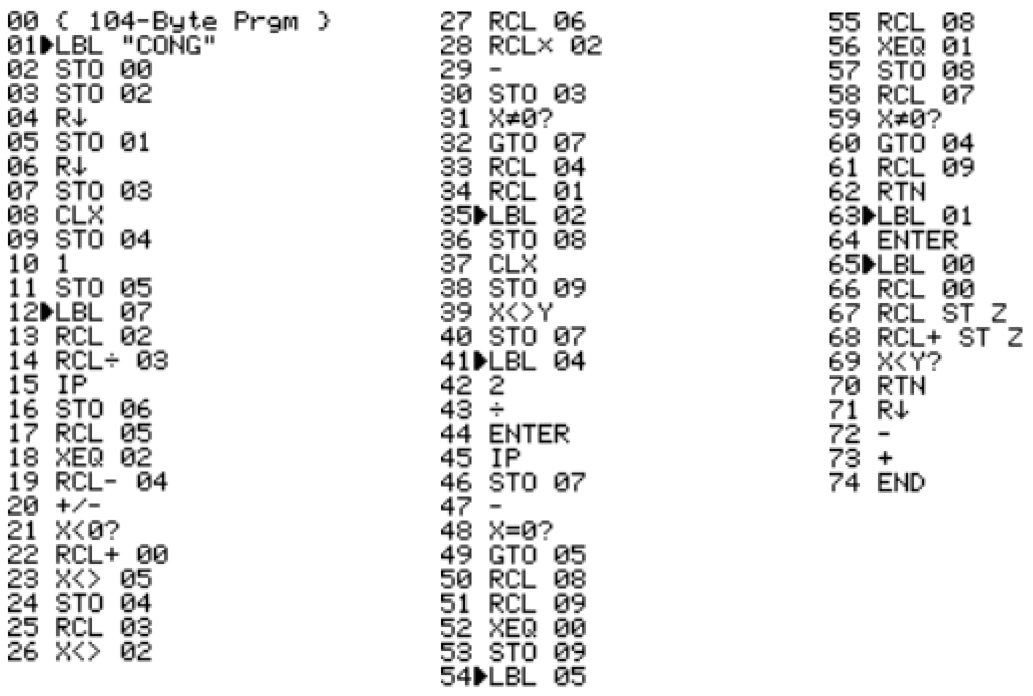
If you did that in PRGM mode, say entering 1900 in the prompt, surprisingly the end result turns out to be "RCL G" – which equals RCL 108. This can be explained by the (apparently unrelated) fact that MOD(1900, 128) = 108, i.e. we've gone full circle in data registers parlance.

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** and **RIND2**, which perform a double STO/RCL IND IND _ _

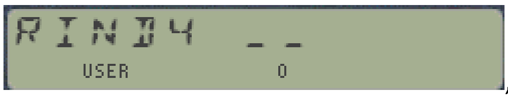
Enough to make your head spin a little? – Then you should try the TRIPLE indirection, available when you hit the shift key at that stage, ie:

SIND2 IND _ _ = STO IND IND IND _ _
RIND2 IND _ _ = RCL IND IND IND _ _

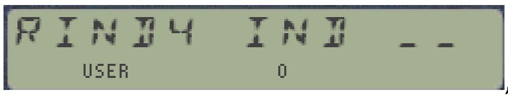
These functions use two (or three if SHIFTED) standard data registers to hold the arguments of the data register where the value is to be recalled from (RIND2) or stored into (SIND2). Better keep your register maps handy!

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:

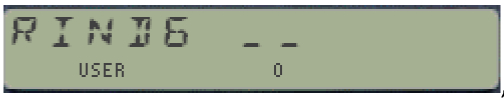

RIND4 _ _
USER 0

, and then:

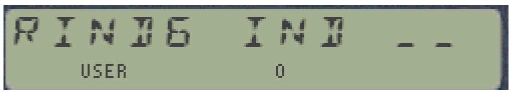

RIND4 IND _ _
USER 0

,

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).

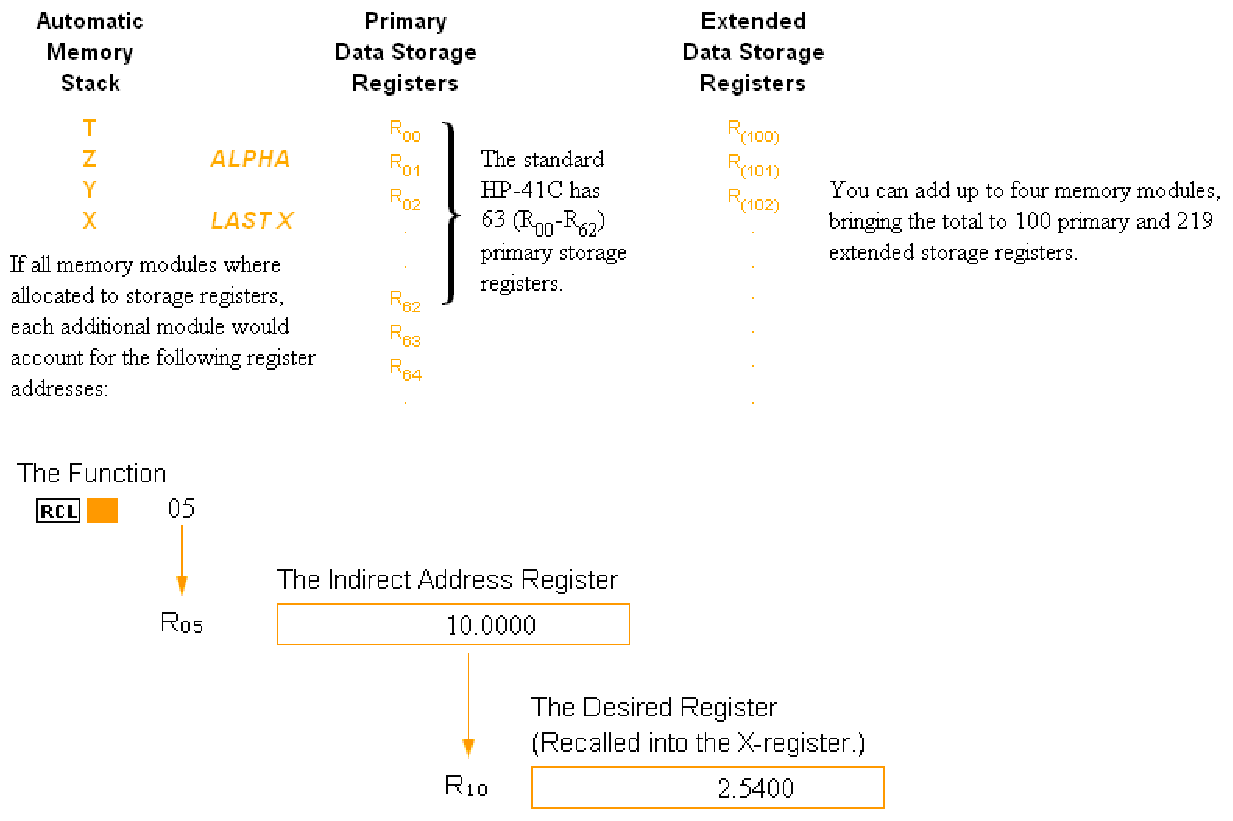
Application Example: Bubble Sort without data movement. *(By Greg McClure)*

```
;
; 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
      08 RIND2
      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.-



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

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.

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.

Numbers on the 41 platform are represented by 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 "whole" 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:

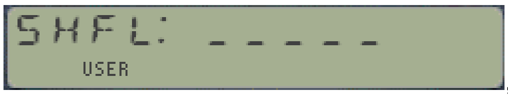
   FIX = max { 0 , [(9-z#) + XP] }

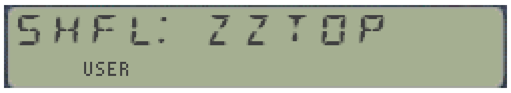
2. If number < 1 (or x="9") - Let |XP| = (100 – xyz) . Then we have:

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

Stack Shuffling and selective clearing. { **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.



, i.e:


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, but a few considerations must be made:

- Register M is always used by 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

Finally, 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

Entering this function 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. Note that a DATA ERROR message will come up (and the program execution will stop) should that string contain any invalid character – but it will ignore characters beyond the fifth one starting from the RIGHT of the ALPHA registers.

Checking the results.

For a quick check of the results you can use the sub-function **STVIEW** for an enumeration of the stack registers in L-X-Y-Z-T sequence – a nice complement to help you keep your bearings at all times. **STVIEW** is accessible pressing [R/S] at the main STK: launcher.

The “Shadow Stack” concept.

The underpinnings of **SHFL** 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 copied back to 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 feature to intently make a “shadow” copy of the stack in the buffer – in case you’d want to restore all the contents after some operation (like a UNDO instruction would perform), or simply as a safety backup. 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.

To restore the original values, just use **bRCL** on the buffer registers following this arrangement:

X – bR5	Z – bR3	L –bR1
Y – bR4	T – bR2	

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  NEXT T                 -     n     s     d
07  “YYZT”                 n     n     s     d
08  SHFL
09  RC/ T                  n/d   n     s     d
10  ?x< T
11  GTO 10
12  FRC?                   n/d   n     s     d
13  GTO 10
14  ?X# T                  n/d   n     s     d
15  RC+ T
16  ST+ Z
17  GTO 05
18* LBL 10
19  x<> Z                  s     n     n/d   d
20  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.

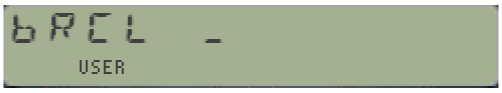
Emergency Buffer Registers Storage.

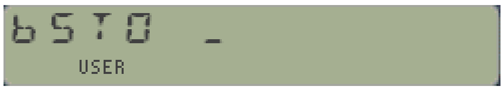
If you've ever run out of data registers and wished there was a "back-door" mechanism to use in emergencies, then you should find this section interesting. These functions operate on a I/O buffer (with id#7) located below the .END. and above the Key assignment area.

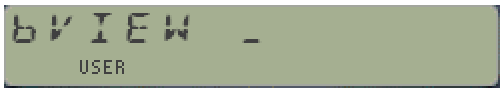
The buffer holds five extra registers for standard data storage, labeled bR1 to bR5 (therefore there's no bR0 to speak of). Just enter the index for the extended register in the prompt and the data will be stored, recalled, or exchanged with the stack X-register – as if they were standard data registers.

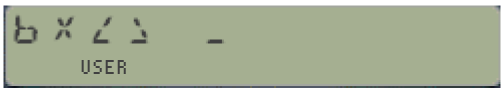
- **bRCL _** recalls to the X register the content of the extended reg. which index is provided in the prompt, or in the next program line if used in a running program.
- **bSTO _** stores the X-register in the extended reg. given in the prompt, or in the next program line if used in a running program.
- **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 except 1,2,3,4,5) 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 has to be done manually for all sub-functions, and therefore should always be within valid range.









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 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. All 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).

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 the information of the currently selected variable, i.e. the data 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 5; 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.
- **bSTO 0** stores the value in the X register in the selected register,
- **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.

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 following 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: the buffer will be created the first time you need it to save/retrieve data to/from the extended registers, or call the RTN stack backup functions, or perform a stack shuffle or choose a variable for SLCT/?CASE operation. This is the reason why you may notice a slighter longer execution time the first time this is done.

	Storage	RTN Stack	Shadow Stack
fifth	bR5	-	Shadow-X
fourth:	bR4	-	Shadow-Y
third:	bR3	-	Shadow-Z
second:	bR3	reg 10(a)	Shadow-T
first:	bR1	reg 11(b)	Shadow-L
header:	SEL# pointer	-	-

Warning: This buffer is not automatically created by the module on start-up, so the data it contains will not survive a power-on/off cycle. This also applies to the selected variable used by SLCT.

Finding the X-needle in the 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#** 58) 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.

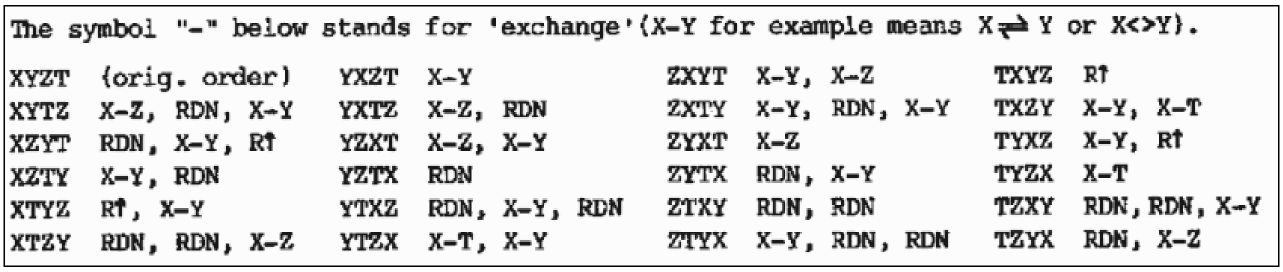
The function returns the number of the first data register found that contains the same value as the X-Register. If none is found, the function puts -1 in X to signify a no-match situation. The stack is lifted so the sought for value will be pushed to the Y-register upon completion.

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 with the alternate one 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  XF#     (?S= Y)
11  62
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
```

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

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.

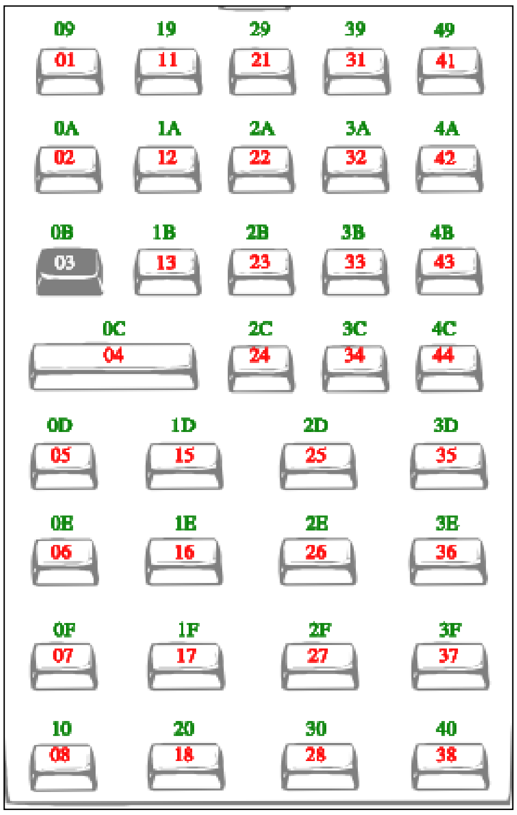
KEY ?
USER

- **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.



Appendix. Changes from previous revisions.

In order to make room for the **XEQ+** mode (about 800 bytes of code!), a few functions had to be removed from the module, as listed below. Note that these are also available in other modules, so it's not an absolute loss but a relocation instead. The sections describing these functions are kept here as an appendix of this manual for consistency and reference purposes; just be aware that they're not included anymore.

Function	Available in:	And also in:
GETST	RAMPage ROM	PowerCL
SAVEST	RAMPage ROM	PowerCL
KAFLP	RAMPage ROM	XROM ROM
PUSHRTN	XROM ROM	RECOURSE Module
POPRTN	XROM ROM	RECOURSE Module
ROM2HEX	XROM ROM	GJM ROM
HEX2ROM	XROM ROM	GJM ROM
AIRCL	ALPHA ROM	SandMath

Saving Status Registers in X-Memory.

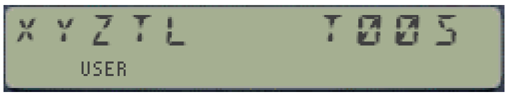
You can use sub-functions **SAVEST** and **GETST** to make backup copies of the status registers into X-Memory files, and to restore their contents back to the status area. The functions prompt for the number of status registers to include in those back-up files, which must be at least one and not more 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:



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.

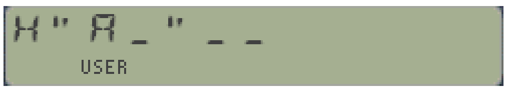
File End Marker
Register P(8)
Register O(7)
Register N(6)
Register M(5)
Register L(4)
Register X(3)
Register Y(2)
Register Z(1)
Register T(0)
FL Header Reg
FL Name Reg

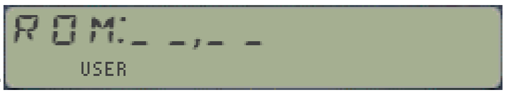
XROM to and from HEX bytes. (by Greg McClure)

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 function 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.

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

There are four main groups of functions that support the extended prompting facility, as follows: Register Swaps; RCL Math; Comparison Tests; and Double INDirection. All of them share the main core code section to provide support for INDirect addressing, Stack registers or the combination of both, thus 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
- Reg Swaps needs descriptors for source and destination registers
- Comparison tests need descriptors for the operator and source and destination regs
- Double Indirection needs descriptors for RLC/STO, multiplicity order, and source register

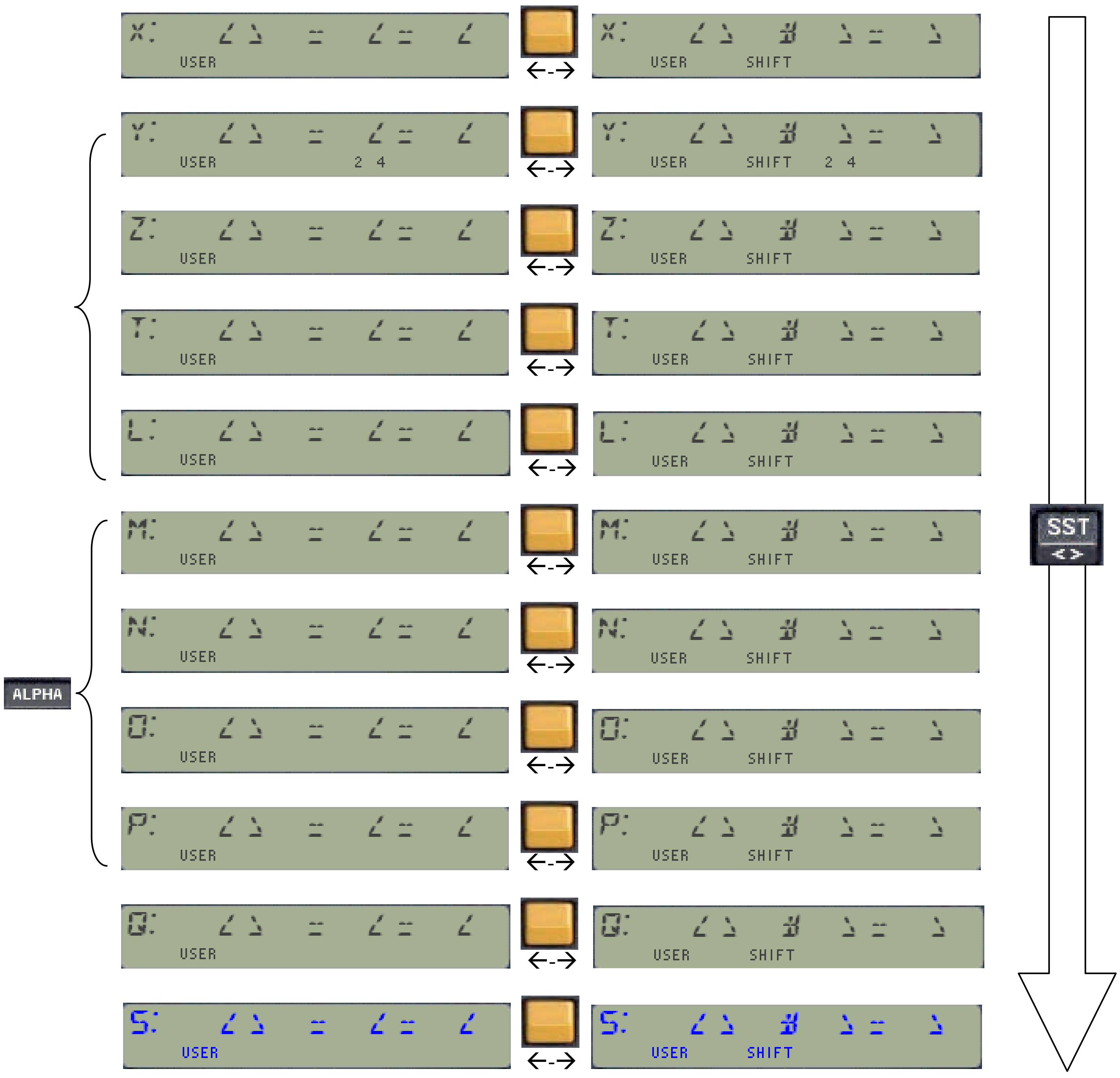
To be able to use the same core code, all these must be arranged in a common scheme that is compatible with all functions. Besides, it needs to survive calls to partial key entry sequences, and cannot overwrite system flags 3 and 4 – used to signal running program and PRGM entry conditions.

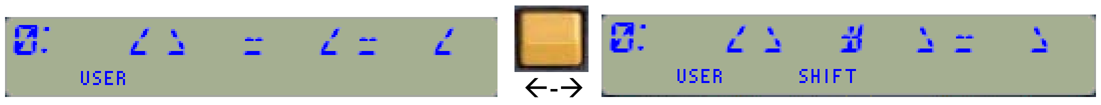
The table below describes such arrangement. The fields are stored in the "Data Configuration Register", which is populated by the routines and saved in the stack register 9(Q) as temporary repository.

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

(*) F8 status is transferred to F4 after SST condition check.

Appendix.- Dare to Compare: 96 functions at your fingertips !





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