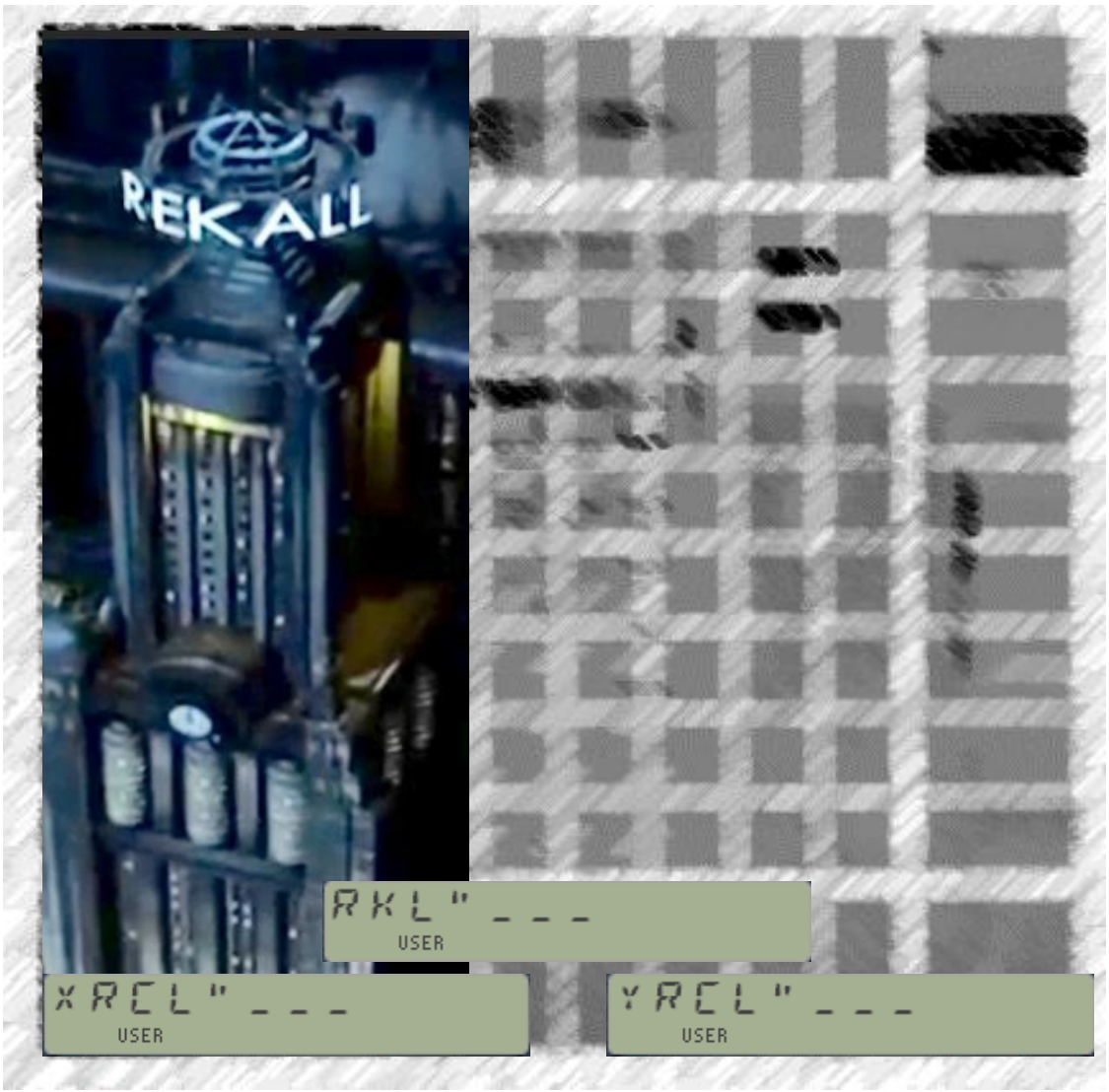
THE TOTAL_REKALL+ 2024
HP-41 MODULE

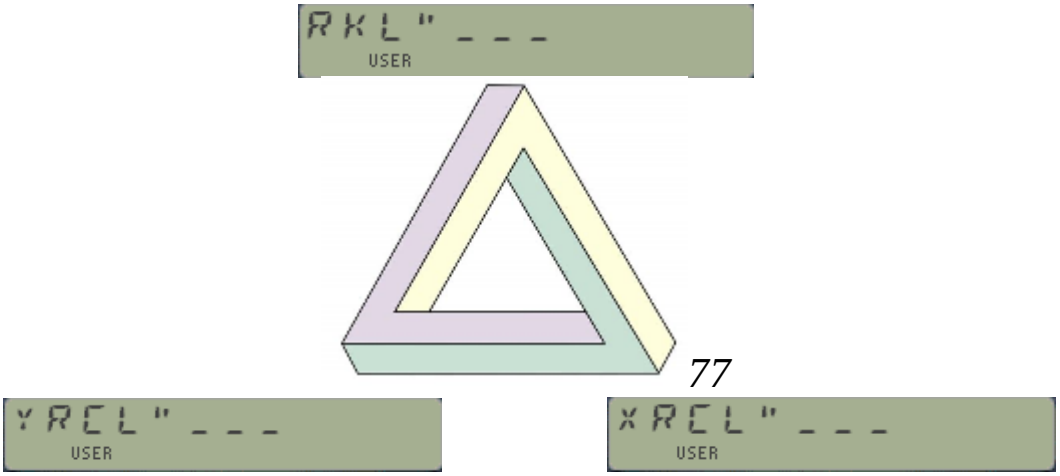
RCL Math and X-Mem Access for the HP-41

Plus Expanded Mem RCL for SY-41CL and MAXX Users.



Written and programmed by Ángel Martin
February 2024

This compilation revision 1.1.1



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

Acknowledgments.- This module is a derivative of several previous project like the XMEM_TWIN and the "CL Expanded Registers" It also overlaps with the WARP_Core module in some areas.

Everlasting thanks to the original developers of the HEPAX and CCD Modules – real landmarks 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	**AIRCL _ _ _**	Alpha integer Recall	RG# in prompt / Next Line	Lib#4	MCODE	*Ángel Martin*
2	**RKL _ _ _**	*Enhanced RCL function*	Prompts for RG#.	Lib#4	MCODE	*Ángel Martin*
3	**RC- _ _ +**	RCL Subtraction	RG# in prompt / Next Line	Lib#4	MCODE	*Ángel Martin*
4	**RC+ _ _ _**	RCL Addition	RG# in prompt / Next Line	Lib#4	MCODE	*Ángel Martin*
5	**RC* _ _ _**	RCL Multiply	RG# in prompt / Next Line	Lib#4	MCODE	*Ángel Martin*
6	**RC/ _ _ _**	RCL Divide	RG# in prompt / Next Line	Lib#4	MCODE	*Ángel Martin*
7	**RC^ _ _ _**	RCL Power	RG# in prompt / Next Line	Lib#4	MCODE	*Ángel Martin*
8	**RCI^ _ _ _**	RCL Root	RG# in prompt / Next Line	Lib#4	MCODE	*Ángel Martin*
9	**RIND2 _ _ _**	RCL IND IND	RG# in prompt / Next Line	Lib#4	MCODE	*Ángel Martin*
10	**SIND2 _ _ _**	STO IND IND	RG# in prompt / Next Line	Lib#4	MCODE	*Ángel Martin*
11	**ST^ _ _ _**	STO Power	RG# in prompt / Next Line	Lib#4	MCODE	*Ángel Martin*
12	**STI^ _ _ _**	STO Root	RG# in prompt / Next Line	Lib#4	MCODE	*Ángel Martin*
13	**XIND2 _ _ _**	X<> IND IND	RG# in prompt / Next Line	Lib#4	MCODE	*Ángel Martin*
14	**STKSWP _**	Stack Swaps Launcher	Prompts for Stack Reg letter	Lib#4	MCODE	*Ángel Martin*
15	**RK# _ _**	Sub-function Launcher	Function index in prompt	Lib#4	MCODE	*Ángel Martin*
16	**-X-REGS FNS**	*Section Header*	n/a	Lib#4	MCODE	*Ángel Martin*
17	**A<>XRG _ _ _**	Swap ALPHA and XRGs	RG# in prompt / Next Line	Lib#4	MCODE	*Ángel Martin*
18	**CLXRG**	Clear ALL X-Regs	none	Lib#4	MCODE	*Ángel Martin*
19	**CLXRGX**	Clear X-Regs RANGE	RG# range in X	Lib#4	MCODE	*Ángel Martin*
20	**ST<>XRG _ _ _**	Swap Stack ans XRGs	RG# in prompt / Next Line	Lib#4	MCODE	*Ángel Martin*
21	**XARC _ _ _**	X-Reg ARCL	RG# in prompt / Next Line	Lib#4	MCODE	*Ángel Martin*
22	**XAST _ _ _**	X-Reg ASTO	RG# in prompt / Next Line	Lib#4	MCODE	*Ángel Martin*
23	**XDSE _ _ _**	X-Reg DSE	RG# in prompt / Next Line	Lib#4	MCODE	*Ángel Martin*
24	**XFINDX**	Find X in X-Mem	Value in X	Lib#4	MCODE	*Ángel Martin*
25	**XISG _ _ _**	X-Reg ISG	RG# in prompt / Next Line	Lib#4	MCODE	*Ángel Martin*
26	**XRCL _ _ _**	X-Reg RCL	RG# in prompt / Next Line	Lib#4	MCODE	*Ángel Martin*
27	**XRC+ _ _ _**	X-Reg. RC+	RG# in prompt / Next Line	Lib#4	MCODE	*Ángel Martin*
28	**XRC- _ _ _**	X-Reg RC-	RG# in prompt / Next Line	Lib#4	MCODE	*Ángel Martin*
29	**XRC* _ _ _**	X-Reg RC-	RG# in prompt / Next Line	Lib#4	MCODE	*Ángel Martin*
30	**XRC/ _ _ _**	X-Reg RC/	RG# in prompt / Next Line	Lib#4	MCODE	*Ángel Martin*
31	**XRGMOV**	Move X-Regs	RG# irange in X	Lib#4	MCODE	*Ángel Martin*
32	**XRGSWP**	Swap X-Regs	RG# range in X	Lib#4	MCODE	*Ángel Martin*
33	**XSTO _ _ _**	X-Reg. STO	RG# in prompt / Next Line	Lib#4	MCODE	*Ángel Martin*
34	**XST+ _ _ _**	X-Reg ST+	RG# in prompt / Next Line	Lib#4	MCODE	*Ángel Martin*
35	**XST- _ _ _**	X-Reg ST-	RG# in prompt / Next Line	Lib#4	MCODE	*Ángel Martin*
36	**XST* _ _ _**	X-Reg ST*	RG# in prompt / Next Line	Lib#4	MCODE	*Ángel Martin*
37	**XST/ _ _ _**	X-Reg ST/	RG# in prompt / Next Line	Lib#4	MCODE	*Ángel Martin*
38	**XVEW _ _ _**	View X-Reg	RG# in prompt / Next Line	Lib#4	MCODE	*Ángel Martin*
39	**XX<> _ _ _**	X-Reg X<>	RG# in prompt / Next Line	Lib#4	MCODE	*Ángel Martin*
40	**-Y-REGS FNS**	*Section Header*	n/a	n/a	MCODE	*Ángel Martin*
41	**A<>XRG _ _ _**	Swap ALPHA and Y-RGs	RG# in prompt / Next Line	Lib#4	MCODE	*Ángel Martin*
42	**CLXRG**	Clear ALL Y-Regs	none	Lib#4	MCODE	*Ángel Martin*
43	**CLXRGX**	Clear Y-Regs RANGE	RG# range in X	Lib#4	MCODE	*Ángel Martin*
44	**ST<>YRG _ _ _**	Swap Stack and Y-RGs	RG# in prompt / Next Line	Lib#4	MCODE	*Ángel Martin*
45	**YARC _ _ _**	Y-Reg ARCL	RG# in prompt / Next Line	Lib#4	MCODE	*Ángel Martin*
46	**YAST _ _ _**	Y-Reg ASTO	RG# in prompt / Next Line	Lib#4	MCODE	*Ángel Martin*
47	**YDSE _ _ _**	Y-Reg DSE	RG# in prompt / Next Line	Lib#4	MCODE	*Ángel Martin*
48	**YFINDX**	Find X in Y-Mem	Value in X	Lib#4	MCODE	*Ángel Martin*
49	**YISG _ _ _**	Y-Reg ISG	RG# in prompt / Next Line	Lib#4	MCODE	*Ángel Martin*
50	**YRCL _ _ _**	Y-Reg RCL	RG# in prompt / Next Line	Lib#4	MCODE	*Ángel Martin*

#	Function	Description	Input	Dependency	Type	Author
51	YRC+ _ _ _	Y-Reg. RC+	RG# in prompt / Next Line	Lib#4	MCODE	*Ángel Martin*
52	YRC- _ _ _	Y-Reg RC-	RG# in prompt / Next Line	Lib#4	MCODE	*Ángel Martin*
53	YRC* _ _ _	Y-Reg RC-	RG# in prompt / Next Line	Lib#4	MCODE	*Ángel Martin*
54	YRC/ _ _ _	Y-Reg RC/	RG# in prompt / Next Line	Lib#4	MCODE	*Ángel Martin*
55	YRGMOV	Move Y-Regs	RG# irange in X	Lib#4	MCODE	*Ángel Martin*
56	YRGSWP	Swap Y-Regs	RG# range in X	Lib#4	MCODE	*Ángel Martin*
57	YSTO _ _ _	Y-Reg. STO	RG# in prompt / Next Line	Lib#4	MCODE	*Ángel Martin*
58	YST+ _ _ _	Y-Reg ST+	RG# in prompt / Next Line	Lib#4	MCODE	*Ángel Martin*
59	YST- _ _ _	Y-Reg ST-	RG# in prompt / Next Line	Lib#4	MCODE	*Ángel Martin*
60	YST* _ _ _	Y-Reg ST*	RG# in prompt / Next Line	Lib#4	MCODE	*Ángel Martin*
61	YST/ _ _ _	Y-Reg ST/	RG# in prompt / Next Line	Lib#4	MCODE	*Ángel Martin*
62	YVEW _ _ _	View Y-Reg	RG# in prompt / Next Line	Lib#4	MCODE	*Ángel Martin*
63	YX<> _ _ _	Y-Reg X<>	RG# in prompt / Next Line	Lib#4	MCODE	*Ángel Martin*
0	-STKSWAP	Section Header	n/a	n/a	MCODE	*\Angel 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	Y<> _ _	Swap Y and Register	RG# in prompt / Next Line	Lib#4	MCODE	*Greg McClure*
8	Z<> _ _	Swap Z and register	RG# in prompt / Next Line	Lib#4	MCODE	*Ángel Martin*
9	T<> _ _	Swap T and register	RG# in prompt / Next Line	Lib#4	MCODE	*Ángel Martin*
10	L<> _ _	Swap L and register	RG# in prompt / Next Line	Lib#4	MCODE	*Ángel Martin*
11	M<> _ _	Swap M and register	RG# in prompt / Next Line	Lib#4	MCODE	*Ángel Martin*
12	N<> _ _	Swap N and register	RG# in prompt / Next Line	Lib#4	MCODE	*Ángel Martin*
13	O<> _ _	Swap O and register	RG# in prompt / Next Line	Lib#4	MCODE	*Ángel Martin*
14	P<> _ _	Swap P and register	RG# in prompt / Next Line	Lib#4	MCODE	*Ángel Martin*
15	Q<> _ _	Swap Q and register	RG# in prompt / Next Line	Lib#4	MCODE	*Ángel Martin*
16	-XTRA FNS	Section header	Shows Splash screen	Lib#4	MCODE	*Nelson Crowle*
17	?YFNX	Checks for YFNX	None	Lib#4	MCODE	*Ángel Martin*
18	?LIB4	Checks for Library #4	none	Lib#4	MCODE	*Ángel Martin*
19	A<>ST	Swaps ALPHA and Stack	None	Lib#4	MCODE	*Ángel Martin*
20	ASWP>	Swaps Text around ">"	String in ALPHA	Lib#4	MCODE	*Ángel Martin*
21	D>H	Decimal to Hex	Dec value in X	Lib#4	MCODE	*William Graham*
22	H>D	Hex to Decimal	Hex value in ALPHA	Lib#4	MCODE	*William Graham*
23	DROP	Puts Y in Reg# in X	Value in Y, RG# in X	Lib#4	MCODE	*Ángel Martin*
24	FETCH	Fetches register contents	Absolute address in X	Lib#4	MCODE	*Ángel Martin*
25	FINDX	Finds value in Data Regs	Value in X	Lib#4	MCODE	*Ángel Martin*
26	RK$ _	Sub-function Launcher	Function NAME in prompt	Lib#4	MCODE	*Ángel Martin*
27	RCLΣ	Recalls Stat Regs to Stack	Data in Stat Regs	Lib#4	MCODE	*Mark Power*
28	STOΣ	Stores Stack in Stack Regs	Data in Stack	Lib#4	MCODE	*Mark Power*
29	ST<>Σ	Swaps Stack with Stat Regs	Data in memory	Lib#4	MCODE	*Nelson Crowle*
30	ST>B7	Copies Stack to Shadow Buff.	Data in Stack	Lib#4	MCODE	*Ángel Martin*
31	STVIEW	Shows Stack contents	Data in Stack	Lib#4	MCODE	*Ángel Martin*
32	CAT+	Sub-function CATalog	None	Lib#4	MCODE	*Ángel Martin*

Sub-functions are accessed using the **RK#** launcher. Use either the sub-function INDEX number at the prompt, or press ALPHA to type the sub-function NAME.

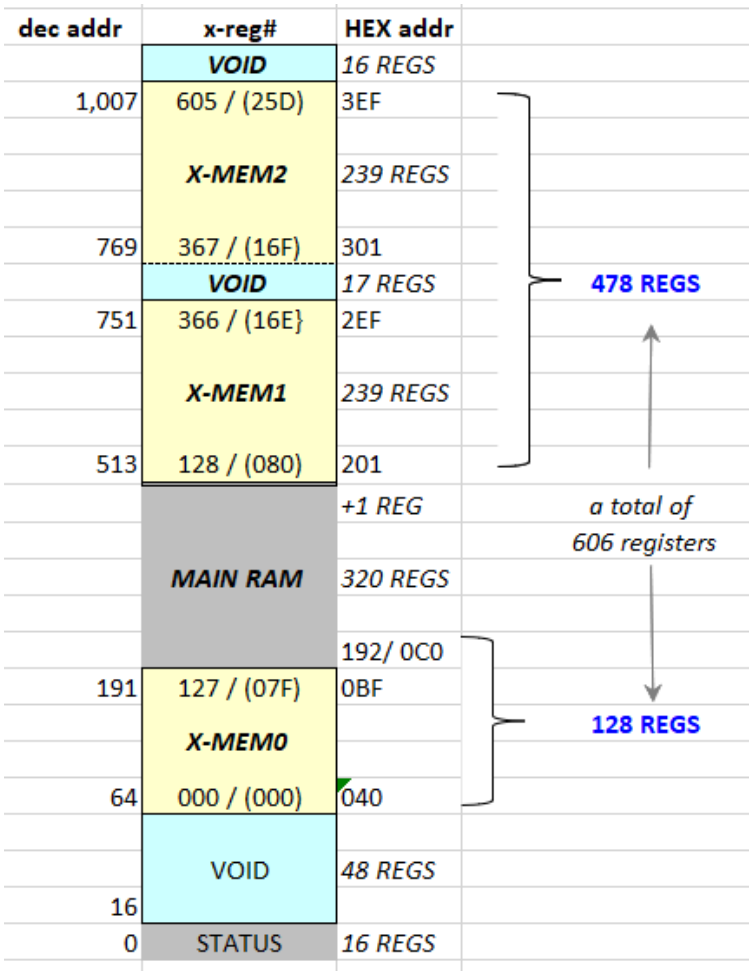
Introduction: Three Types of Memory

Implicit to the functionality provided by this module is the concept of Main, Extended and Expanded memory types.

- Main Memory holds the stack registers, data registers, I/O buffers and program memory area. The data registers capacity is controlled by the **SIZE** function. The maximum capacity on an HP-41 CV/CX is 319 registers, which requires that there are no programs, buffers, or key assignments present in the system.

- Extended memory is provided by the X-Functions and X-Memory Modules on a plain HP-41C/CV machine, and on an HP-41CX it comprises 606 data registers. The extended memory is usually structured in files but this module accesses the registers individually with the X-REGS functions – and therefore is not compatible with the standard arrangement. Refer to the **X-MEM_TWIN** module for a comprehensive description and alternate ways to avoid incompatibilities – certainly out of the scope of this module.

- Expanded Memory on a SY-41 CL or a MAXX module comprises 1,024 registers for the purpose of this module. It can be used as a bulk backup location, or its contents can also be accessed individually using the Y-REGS functions. Refer to the **XPREGS** module for a detailed description and applications.

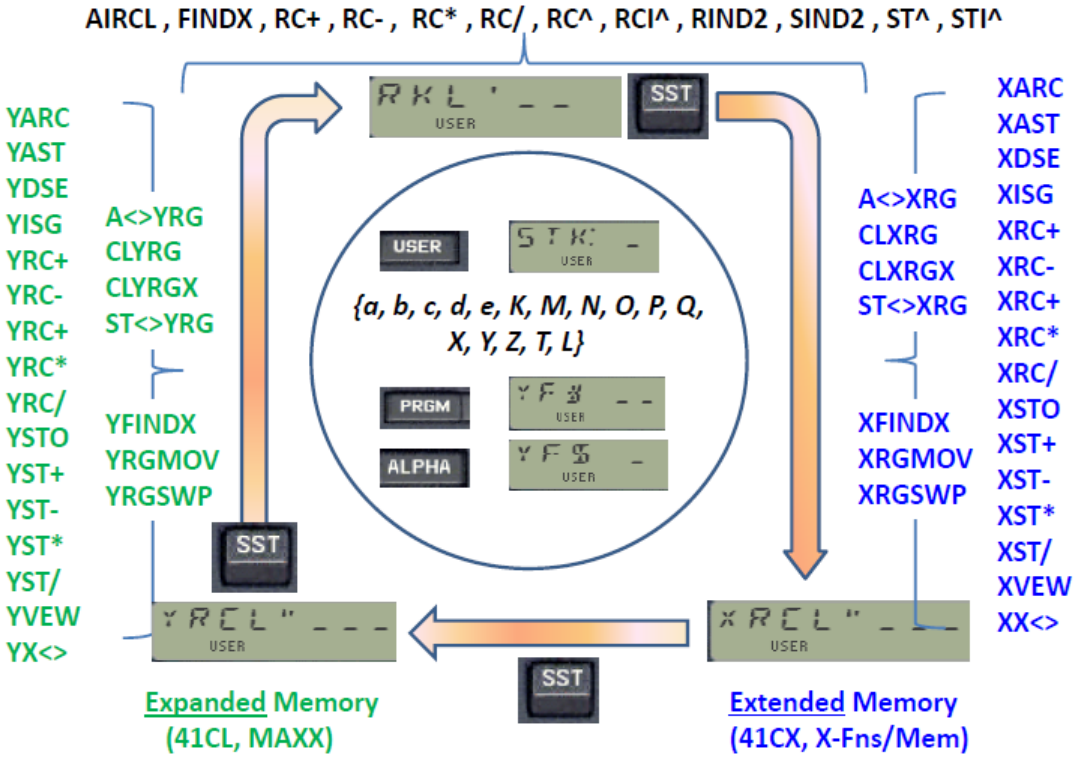


Three Launchers for the Three Kinds of Memory { RKL , XRCL , YRCL }

Implicit to the previous description is the concept of Main, Extended and Expanded memory types. Each one of the three sections in the module is dedicated to each of these types.

- The main memory section is an extension of the native capabilities, mostly (but not exclusively) related to the recall operations. A few other additions are included
- The Extended and Expanded sections are complete new additions and therefore include functions to handle the storage, recall and exchange of data in the corresponding memory area. The function names in these two sections are very similar, only differentiated by the initial letter being "X" or "Y" – a consistent naming scheme that facilitates their usage.
- Going across memory zones is also possible using the stack registers as common anchor available in the three sections. For instance from Main memory to X-Mem, or from Y-Mem to X-Mem will need to move the data through the stack register X.

The picture below summarizes all module functions grouped by the memory zone. Note that the three RCL launchers are interconnected, you can cycle through them using the SST key.



The Total_Rekall Dilemma.

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 subject of previous implementation attempts to close that gap.

The first component is naturally the individual RCL math functions, like **RC+, RC-**, **RC*** and **RC/**. These can be written without much difficulty, even supporting INDirect 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 trickery, basically writing extra M-code to replace the OS handling of the prompting in these functions – based on the PTEMP bits of the function name. The 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 those managing the direct access to **Extended Memory** (X-Mem) registers individually, or even those dedicated to the same task for **Expanded Memor**y (Y-Mem) registers – available on the SY-41 CL and MAXX modules.

That's how the idea of a Recall launcher came into being, following the same model available in the native OS for the STO math functions – grouping them under a logical structure. Subsequently the RKL launcher added further capabilities within the module, and became a new and enhanced RCL on its own beyond the grouping features.

Side note: The Sub-function Launcher **RK#**

RK# is available as a main function in the FAT, therefore it can be assigned to any key – or can it? The issue of course is the non-keyable character "#", posing a small challenge that requires either synthetic programming skills or using the extended ASN facility available in advanced modules like the AMC_OS/X.

But say that no advanced modules are at hand – then our friendly **RKL** launcher comes to the rescue: pressing the PRGM key at its prompt activates RC# as well :\

Module Dependencies.

The XMEM Twin is a Library4-aware module; therefore, it expects the Library#4 revision R4 to be present on the system. The module will check for it upon the calculator ON event, showing an error message if not found. This will abort the polling points sequence for all other modules plugged at higher position in the bus. *Do not attempt to run the programs or functions within the module without the Library#4 plugged in.*

Sub-function Catalog.

CAT+	**Sub-functions Catalog**	Ángel Martin	Source: Library #4
XEQ `	**Direct Execution**		
LASTF	**Last Function**		

The module includes an auxiliary FAT with a set of 22 sub-functions. Because these are not in the main FAT the OS knows nothing about them, so they cannot be called using XEQ, nor can they be assigned to keys using ASN. Therefore, a dedicate way to access them must exist. Two functions are available to call sub-functions either alphabetically (by spelling their name in ALPHA) or numerically, by entering their corresponding index# in the prompt:

Like all other modules with sub-functions, there is a way to enumerate them using **CAT+** – itself a sub-function included in the auxiliary FAT. We already saw that **CAT+** can be triggered from the **CAT+** prompt, pressing [ENTER^], spelling its name with **RK$**, or using index #000 with **RK#**.

A few hot-keys allow you to control and navigate the catalog during the enumeration:

- [R/S] stops and resumes the listing
- [SST] manually advances to the next sub-function
- [SHIFT] reverses the direction of the enumeration
- [ENTER^] jumps to the next/previous section header
- [XEQ] will execute the sub-function shown if the enumeration is suspended.
- Back-arrow will exit the catalog

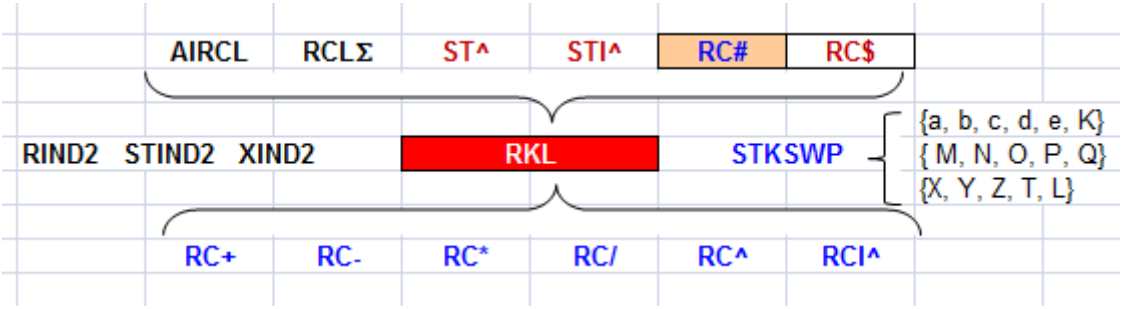
When you execute a sub-function using the launchers or by means of the **CAT+** shortcut, its index# is saved in the LastFunction buffer automatically. This allows subsequent re-execution through the LASTF facility, pressing **RK#** followed by the [RADIX] key.

All sub-functions are programmable. When they're entered into a program its name is briefly shown in the display and two program steps are added by the launchers – one with the numeric launcher RC**#** and another one following it with the corresponding index#. This is done automatically for you, no need to enter it manually.

CAT+ will print the sub-functions names if a printer is connected in NORM or TRACE modes.
The complete list of sub-functions is provided at the beginning of the manual.

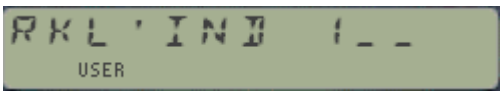
RCL Math on steroids: The Extended RKL.

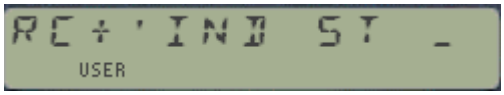
In addition to the four "standard" arithmetic operations available for the **STO** case, this module includes **RC^** and **RCI^** for the Recall Power and root functions – which will calculate the REG-th. power or root of the value in X, (i.e. X= X^RG# and X=X^1/RG#). These two functions are inverse of one another, similar to {RC* and RC/} or to {RC+ and RC-} so one of them undoes the other's effect.



The RKL launcher also includes provision to invoke other recall-related functions such as **AIRCL** and **RCLΣ** – plus it doubles as a sub-function launcher to access the sub-functions included in the module – either by their index number (using **RC#**) or by their names (using **RC$**). The last-function facility is also implemented of course, triggered by the radix key at the RC# prompt.

The stack registers and indirect addressing are of course still available using the same hot-keys, RADIX and SHIFT, same as the standard RCL implementation.





In terms of usability, note that you can switch amongst the six RCL math functions pressing the corresponding arithmetic key at their prompt. Besides, you can also revert back to the native RCL pressing the [RCL] key again at its prompt.

Direct access to the upper data registers

All RKL functions feature a 3-digit *prompt field* to directly access registers in the 100-319 range. Note that contrary to the standard native RCL function, using **RKL** in the range from 112 and up **you'll be accessing these upper main memory registers directly**, no need to use indirect addressing anymore - Thus simplifying the user interface and bringing more consistency to the machine operation.

In manual mode (interactive use) the **RKL** function code is always used, and not the standard RCL - not even for the simplest cases such as RCL 00, etc.

During program editing however the **RKL entries are automatically replaced by the native counterparts** in the OS *whenever possible*. Using the standard OS achieves maximum byte efficiency in line entering for the standard cases, i.e. RCL 27 or RCL IND ST X in a single program step as opposed to using the non-merged approach.

This replacement won't happen when the data register lies in the upper region, i.e. above R111 – and in those instances the **RKL** function itself will be entered in the program, followed by a numeric line coding the corresponding register number – as well as the INDirect addressing case of course. This is reflected in the summary table shown in next page – including the Shadow Stack buffer area.

Note that even if the U/I doesn't provide access for registers above R199 nothing is stopping you from editing the parameter line manually with values up to 319 - assuming they are configured via SIZE.

Registers	Direct Addressing	INDirect Addressing
Lower (R00-R111)	RCL nn, merged	RCL IND nn
Upper (R112-R319)	**RKL** nn , non-merged	**RKL** nn+0x180, 2 lines
Stack (T, Z, Y, X, L….)	RCL ST nn +0x170	RCL IND ST nn+0x2F0
Shadow Buffer (A-G)	**RKL** nn+0x300	R**KL** nn+0x480

Table 1. Data Registers Zones

Shadow Stack Buffer

The Shadow Stack area is a 7-register buffer in main memory (with buffer id#=7) that can be used as stack backup, or as emergency data storage for those cases when we run out of data registers (been there before?). Accessing the buffer suffix "bR" is simple: just press the RADIX key twice at the RCL function prompt, followed by the buffer register number (only from 1 to 7 are allowed). In fact repeat pressings of the Radix key will toggle between the ST and BF suffixes, as shown below.

This module includes the sub-function **ST>B7** to save data from the stack into the Shadow Stack buffer – refer to the **WARP_Core** module for further information on this subject.

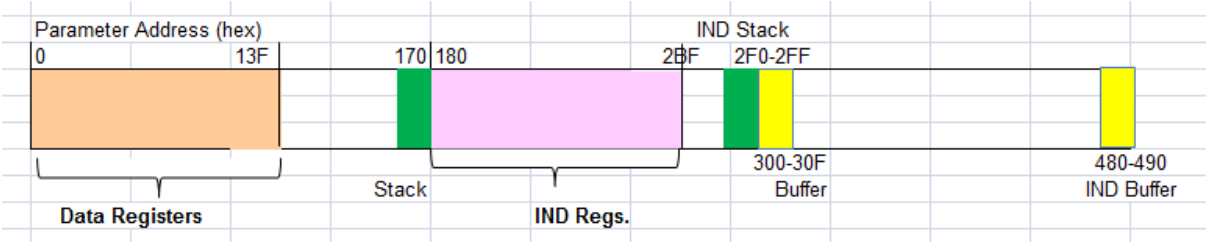
The table below shows the stack registers mapping within the shadow buffer when using the **ST>B7** function. Note that buffer registers F and G are not used in this case.

Stack	T	Z	Y	X	L	-	-
Buffer	A	B	C	D	E	F	G

Diagrams of data suffixes.

The following picture shows the suffix ranges used by the different functions to flag register zones in Main memory. They are provided as a FYI only, as there's no need to concern yourself with the internal structure of the implementation to use the functions.

Stack adds 0x170 ; IND adds 0x180 ; both combined adds 0x2F0
Buffer adds 0x300 ; IND buffer adds 0x480



Once again, with this implementation ALL registers can be accessed directly and ALL registers can be used as INDirect addressing location. i.e. cases like **RCL 199**, and **RCL IND 125** are now a perfectly valid syntax.

The path not taken

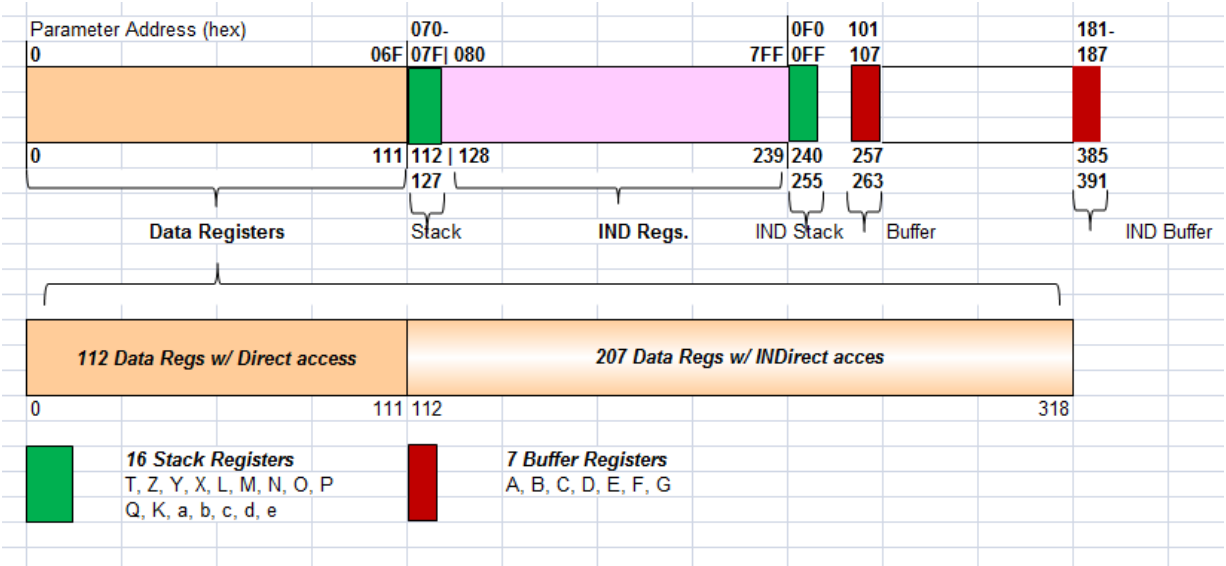
Note the distinctly different approach followed by this module compared to the standard convention used by the HP-41 operating system. Indeed we're departing the tried and tested ways in search for a more logical implementation - this requires a new convention to flag the stack registers and the indirect addressing, different from the standard one used by the OS. With the new technique ALL data registers can be accessed by the RKL function directly, thus it get rids of the overlapping zones consequence of the suffixes 0x070 and 0x080 used for the stack registers and indirect addressing - see the table below for details.

	Argument	Shown:	Argument	Shown:	Argument	Shown:
	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 2: Register index mapping used by the OS

Graphically the Data Suffix diagram for the "Classic" approach is represented below:

Stack adds 0x070; IND adds 0x080 ; both combined adds 0x0F0
Buffer adds 0x100 ; IND buffer adds 0x180



All other modules using extended RCL implementations (notably the **SandMath** and the **WARP_Core**) utilize the "Classic" model..

An exception to the rule.

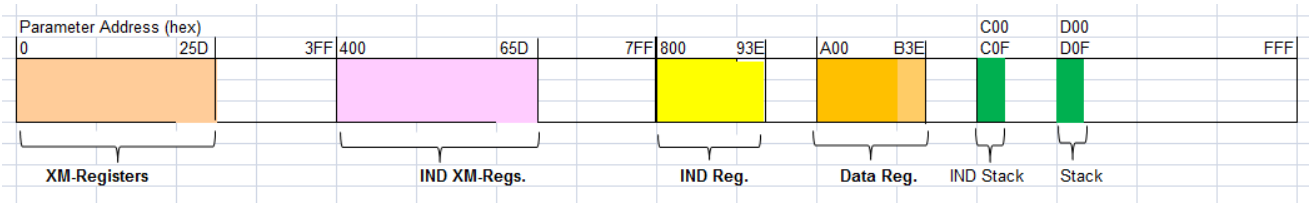
A few prompting functions in the module do not allow stack arguments in their prompts when entered in a program. These are the sub-functions for register swapping. Note however that in manual mode you can use any register number, Stack registers, Buffer registers and INDirect addressing as destination

Be aware that when editing a program the non-merged line 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.* Even more confusing perhaps, note that these exceptions use the "Classic" data register addressing scheme, thus only data registers R00 to R111 can be accessed directly, and registers R112 and above cannot be selected as INDirect addresses. Refer to the table 2 ahown in previous page.

Other data suffix diagrams

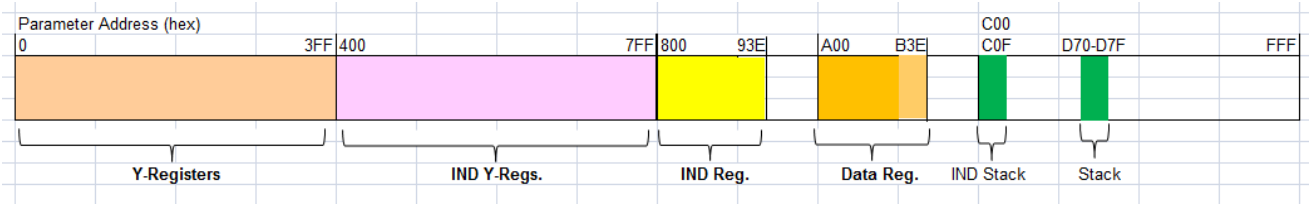
Just for completion sake, see below the corresponding data suffix diagrams for the Extended and Expanded memory cases.

Extended Memory - As usual, it assumes a full X-Mem configuration



In words:
IND adds 0x400 ; Stack adds 0xD00 . Both combined adds 0x 0xC0F
Regular data registers adds 0xA00, IND data registers adds 0x800

Expanded Memory



In words:
IND adds 0x800 ; Stack adds 0xD70 . Both combined adds 0x 0x7F
Regular data registers adds 0xA00, IND data registers adds 0x800

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 indirection for STO / RCL / X<> actions .

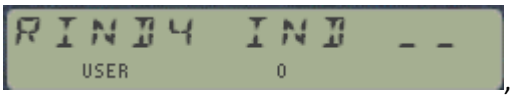
Enough to make your head spin a little? – Then you should try the TRIPLE indirection, available when you hit the shift key at that stage, ie:

SIND2 IND _ _ = STO IND IND IND _ _
RIND2 IND _ _ = RCL IND IND IND _ _
XIND2 IND _ _ = X<> IND IND IND _ _

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

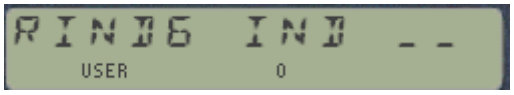
Going over the top: Multiple Indirection

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:


, and then:


,

Followed by the screens shown below in a continuous sequence:


, and then:


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

R10 = 0;
R00 = 3;
R03 = 5;
R05 = 7;
R07 = π

Then we have:

RCL 10 = 0
RCL IND 10 = 3
RIND2 10 = 5
RIND2 IND 10 = 7
RIND4 10 = π
RIND4 IND 10 = 5
RIND6 10 = 7 , etc…

Note that this functionality is restricted to RUN 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
```

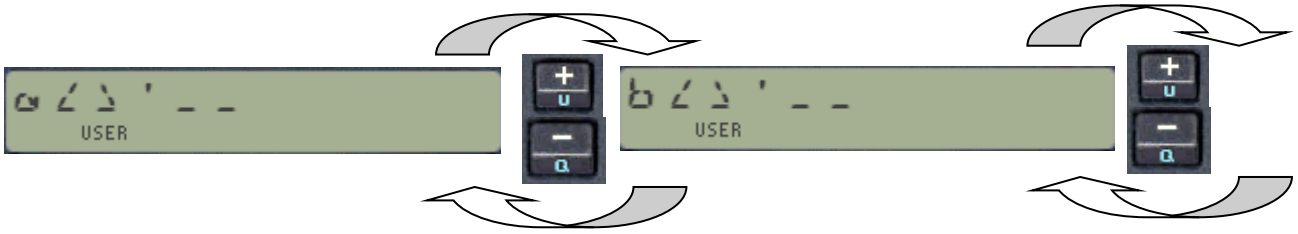
Stack Swapping functions and **STKSWP**

The OS includes the **X<>** function to exchange the contents of the X-register with other stack or data registers – but if fails to extend this to any other stack register. To this point, a set of stack-swapping sub-functions is also included in this module. You can use them to exchange the contents of any of the 16 status registers in the machine, including ALPHA, the stack, and the other reserved status registers (as always, use these with caution!).

In addition to the standard method with the sub-function launchers **RC#** and **RC$**, these functions can be accessed using a dedicated launcher: the function **STKSWP** in the main FAT.

_{a, b, c, d, e, K} ; {M. N. O. P} ; {X. Y. Z. T. L}

This is a very convenient method that offers further usability as it allows the "navigation" across the status registers using the PLUS/MINUS keys. For example, between the "a" and "b" regs:



As expected, Stack registers and Indirect addressing are fully supported by these functions – just like the native **X<>** offers in the standard machine.

;


Caveat emptor.- Because they are sub-functions, entering them in a program requires the user to *manually add the parameter as a THIRD program line* – using the "classic" convention for suffixes, as discussed previously and shown in the table below. Add 0x070 for stack, 0x080 for IND and 0x0F for both. A few examples are included:

M<> N
{RC#, 11, 118}

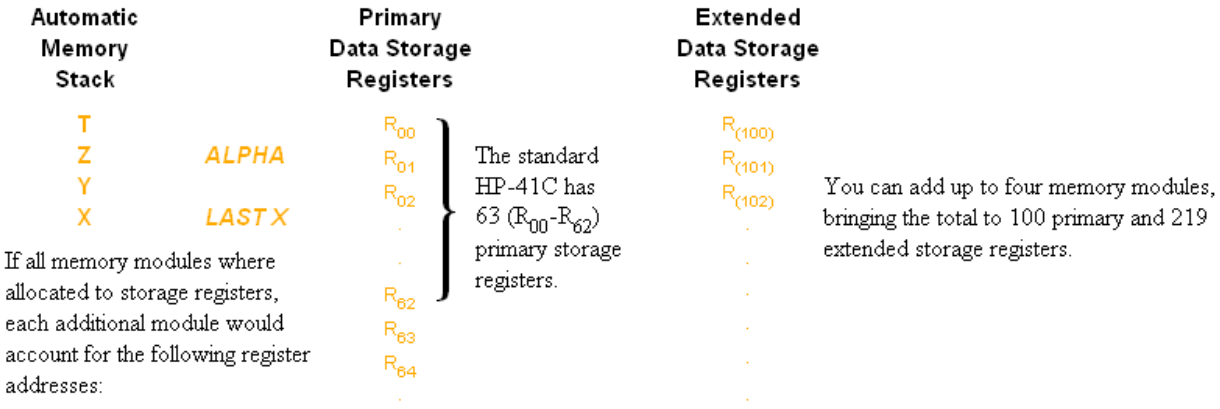
Z<> IND 05
{RC#, 8, 133}

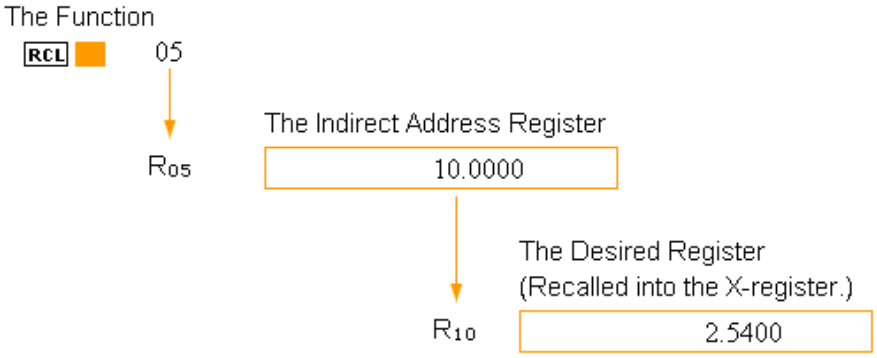
Y<> IND T
{RC#, 7, 240}

8

	Argument	Shown:	Argument	Shown:	Argument	Shown:
	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

Appendix.- A trip down to Memory Lane. From the HP-41 User's Handbook.-





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÷.

Utilities in the Sub-functions area.

Besides the Stack registers swap functions, a few other utilities are included in the auxiliary FAT. They're mostly self-explanatory so we'll include just a short description here.

FINDX is a very fast neat routine that looks for the value in the stack X-register in the data registers area. If a match is found the register number is entered in X , and if not found the value -1 is used instead. The stack is lifted unless of course the stack lift flag is disabled, like when executed right after CLX or ENTER^. So it works much like like **POSA** and **POSFL** in the X-Functions module.

STVIEW does a sequential enumeration of the stack registers, labeling the display with the stack letter for easier traceability.

FETCH recalls the content of the register whose parameter index (following the "new" convention) is in the X register. For example FETCH 25 recalls R25 to X, and FETCH 117 recalls R117 to X.

DROP will put the value in the Y-register into the data register whose number is in X

RCLΣ , **STOΣ**, and **ST<>Σ** manage exchanges between the stack and the statistical registers - a block of five registers defined by the native function **ΣREG**. Refer to the HP-41 user manual for details. The table below shows the register correspondence assuming a ΣREG 11 setting:

Stack	T	Z	Y	X	L
Statistical	R11	R12	R13	R14	R15

A<>ST exchanges the Stack registers {X, Y, Z,, T, L} and the individual ALPHA+ registers (that is the standard ALHA plus the Q-Reg), {M, N, O, P, Q }. It has the corresponding mapping as well:

Stack	T	Z	Y	X	L
ALPHA+	P	O	N	M	Q

ASWP> swaps the two sides of the string in ALPHA around the ">" character – and does nothing if " >" doesn't exist. For instance the string "CATS>DOGS" changes to "DOGS>CATS".

D>H and **H>D** are little hexadecimal to decimal conversion routines. The maximum number allowed is 0x2540BE3FF or 9,99999999 E9 decimal. These functions were written by William Graham and published in PPCJ V12N6 p19, enhancing in turn the initial versions first published by Derek Amos in PPCCJ V12N1 p3.

The input values are expected to be in the X-register (decimal value) for **D>H** and in ALPHA for **H>D** (Hex string) – and vice versa for the output results. A couple of examples: "ABCD", **H>D** => 43,981.000 , **D>H** => "ABCD"

ST>B7 loads the contents of the stack in the Shadow Stack buffer. It was mentioned before in the manual.

?LIB4 and **?YFNX** simply check for the presence of the Library#4 and YFNX modules plugged in the calculator. They put a warning message if not found and halt the program execution if entered in a program.

Part II -. Managing the Global X-Mem Extended registers

This section covers the individual access of the extended registers included in the three X-MEM blocks. You'll be able to store, recall, view, exchange, and perform ISG/DSE operations on 606 of those registers as if they were standard data registers within main memory. Note the presence of the arithmetic operations as well.

Extended Regs	Store	Recall	Other	X-Blocks
X-Register	**XSTO _ _ _**	**XRCL _ _ _**	**XX<> _ _ _**	**CLXRG**
From 0 to 605.	**XST+ _ _ _**	**XRC+ _ _ _**	**XVEW _ _ _**	**CLXRGX**
	XST- _ _ _	**XRC- _ _ _**	**XDSE _ _ _**	**XRGMOV**
	XST* _ _ _	**XRC* _ _ _**	**XISG _ _ _**	**XRGSWP**
	XST/ _ _ _	**XRC/ _ _ _**	**XFINDX**	**ST<>XRG _ _ _**
ALPHA	**XAST _ _ _**	**XARC _ _ _**		**A<>XRG _ _ _**

Besides the direct access, you also have the INDirect addressing capabilities implemented on the expanded registers; the sixteen Stack registers (including synthetic regs {M-e}); and all the standard-Registers - a hybrid mode, unique to this implementation.

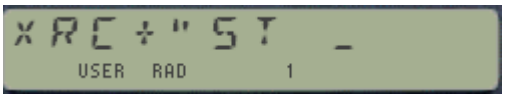
Most of the functions will prompt for the parameters to use. The initial prompt is a three-field underscore for the X-register indeed. Pressing [SHIFT] changes it to IND three-digit fields for another X-register to be used as indirect. Pressing the [RADIX] key changes to the IND ST _ prompt, where you'll enter the register mnemonic, from T to e (all sixteen are available). Pressing the radix key again changes to a IND RG_ _ prompt where you can enter a standard register number to use as indirect address. Repeat pressings of the radix key act as a toggle between those two. There's also provision for direct stack and standard register arguments – even if those can be redundant in practice, being exactly the same as the original ones.

Once you complete the entry adding the register number the action is performed in RUN mode, or two program lines are entered in program mode – automatically selecting the appropriate parameter depending on the direct or indirect types. This is automatically done so you needn't (and shouldn't) edit the value entered in the program's second line at all – which will be properly interpreted in a running program.

You can move between the functions while the prompts are up; not only to select the math operation but also to change the main function amongst the group. So for instance during the **XRCL** _ _ _ main prompt pressing the [SST] key will trigger the **XX<>** function, or pressing [STO] will invoke the **XSTO** function instead. Also *you can revert to the original mainframe functions* pressing the corresponding key of the function in the prompt, for instance here pressing [RCL] will trigger the original RCL _ _

The functions will not allow you to enter any value greater than 605 either as direct index or indirect index – not even when entering them in a program line. Attempting to enter larger values will trigger a "NONEXISTENT" error message. However, that check is not made for IND_RG combinations, as there's no telling at that point about how many standard registers will be available at the execution stage.

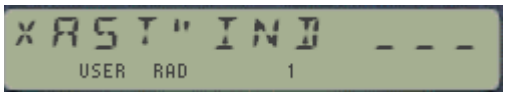
The usage of standard stack and data registers is not only more convenient from the usability standpoint, but also it enables the RCL math on these registers via the **XRCL** function:

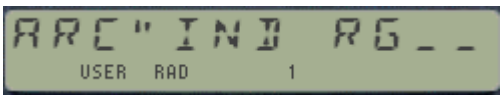


Although possible, it is however not meant to be used in a program because of the obviously higher byte count. That's why when used in an editing program the direct stack and data register X-functions *revert automatically to the native STO/RCL functions instead*, which has the additional benefit of a clearer representation by the OS as merged lines.

Storing and Recalling ALPHA Data

The extended functions **XAST** and **XARC** provide the means to store and recall ALPHA data directly in the expanded registers area. Like their numeric counterparts, they support direct, INDirect, stack and standard registers indexes for a complete palette of options at your disposal. You can access these directly from the XSTO/XRCL prompts by pressing the ALPHA key at any time.





Deleting Expanded Registers.

The function **CLXRG** will delete all the 606 expanded registers.

Additionally with **CLXRGX** you can selectively delete a defined block of extended registers as defined by its control word (in X) "bbb.eeennn", The bbb digits are the base address of the source expanded register block, in the range R0 through R605. The eee digits are the base address destination expanded register block, again in the range R0 through R605. nnn is the incremental step for the registers to delete. If nnn is zero a value of one used.

Other Block Operations.

Think of the following functions as analogous to the X-Functions extensions on the original function set of the calculator, only applied to the extended memory area instead.

- **A<>XRG** and **ST<>XRG** exchange a group of five extended registers with ALPHA (plus Q) or the Stack (T-L) respectively. The start register is to be entered at the prompt in manual mode, or expected to be in the X-Register when running a program. These functions do not allow INDirect indexing.

- **XRGMOV** and **XRGSWP** can be used to move or exchange a block of extended registers at once – either contiguous or in an increment pattern as provided by the control word sss.dddii in the X-register. Much the same as the X-Functions RGMOVE and RGSWAP - in case you wonder.

Moving around the Extended Registers Functions.

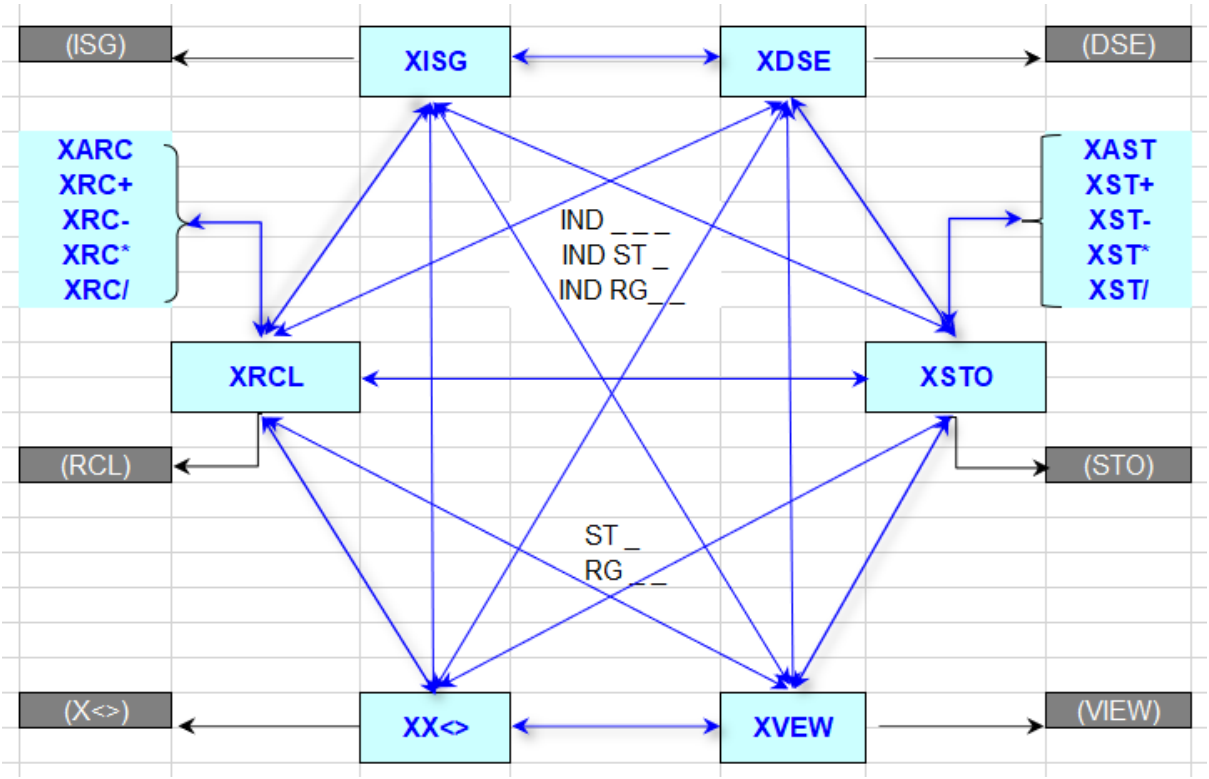


Figure 1.

Even if there isn't a dedicated launcher for these functions, navigation amongst them is as easy as intuitive. First off, assign one of the functions to its "natural" key, for instance **XSTO** to the STO key.

Then while the XSTO _ _ _ prompt is shown you can move about all the X-Reg functions by pressing the key for the corresponding action, i.e.

- RCL will toggle to XRCL _ _ _
- SST (for X<>) will launch XX<> _ _ _
- R/S (for VIEW) will launch XVEW _ _ _
- CHS (for ISG) will launch XISG _ _ _
- ALPHA will trigger XAST _ _ _
- The math keys will launch the corresponding math function, ie. XST+ _ _ _
- EEX (for DSE)will launch XDSE _ _ _
- SHIFT will add the IND _ _ prompt
- RADIX will add the ST _ prompt. All 16 status regs are selectable.
- RADIX again to toggle between ST _ and RG _ _. Choose any standard reg up to 99.
- STO again to exit to the native STO function (no way back!)
- Note that the DIRECT Stack/REG prompt is not strictly needed – that's the native function already. However, the RCL Math functions are useful and are available using this approach.

All the options above are available from within any of the 16 functions – regardless of which one you used to start the sequence. See the descriptions earlier in this manual for more details.

Extra bonus: Finding the X-needle in the X-haystack.

For those times when you'd like to know if a certain value is stored in the X-data registers, the function **XFINDX** is available to do a cursory comparison looking for a match with the value in the X-register. All X-data registers are checked, starting with XR00 until XR605 – which could take a long time depending on where the match exists.

The function returns the number of the first X-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 stack register Y upon completion.

Below there is a FOCAL routine that checks up to XR605, as well as an equivalent routine for the standard data registers - for comparison purposes. See the WARP_Corel manual for yet another routine to tackle this "where is Waldo" problem using other advanced functions.

```
01  LBL "XFNDX"
02  .605
03  X<>Y
04  LBL 00
05  XRCL IND Y (3074)
06  X=Y?
07  GTO 02
08  RDN
09  ISG X
10  GTO 00
11  CLX
12  -1
13  RTN
14  LBL 02
15  X<> Z
16  INT
17  END
```

```
01  LBL "FINDX"
02  SIZE?
03  E
04  –
05  E3
06  /
07  X<>Y
08  LBL 00
09  RCL IND Y
10  X=Y?
11  GTO 02
12  RDN
13  ISG Y
14  GTO 00
15  CLX
16  -1
17  RTN
18  LBL 02
19  X<> Z
20  INT
21  END
```

The possibilities of having an additional set of 606 registers available to your own programs are wide and deep. For starters you could permanently operate with a SIZE 000 and use all the 320 standard registers in main memory for User Code programs, key assignments and I/O buffers; so a few more bytes taken up by the parameter lines won't be a problem.

Converting Standard Programs

Having a complete function set ensures that you can convert programs very easily, simply by replacing the standard functions with their expanded version. Even the ALPHA storage functions **XAST** and **XARC** are included, which can also use the expanded register range.

Then you have the benefit of a much larger set of registers (606 vs. a maximum of 319 without any program in RAM) available for your program, a sheer advantage to manage larger size cases of the problem you're trying to solve – from matrix operations to sorting data, to mention just a couple.

For example, with a few modifications the PPC ROM programs **S2** and **S3** can be used to sort more than 600 registers in a very efficient way. – with random data populating those registers it took about 20 seconds to sort 606 registers on TURBO 50 mode!

See below two simple routines I used to populate the registers and to view them. They expect the control word bbb.eee in X before you run them.

01	**LBL "XVIEW"**
02	LBL 00
03	"XR"
04	ARCLI
05	"|-: "
06	**XRCL IND X** (3075)
08	ARCLX
09	AVIEW
10	PSE
11	RDN
12	ISG X
13	GTO 00
14	END

01	**LBL "XRAN"**
02	RCL X
03	LBL 00
04	RNDM
05	**XSTO IND Y** (3074)
07	RDN
08	ISG X
09	GTO 00
10	RDN
11	END

Functions **ARCLI** and **RNDM** are available in the AMC_OX/X Module.

Note.- In case you're interested, the parameter lines used by these functions as non-merged, second line, correspond to the following:

1. The register index for direct access, from 0 to 605 [000 – 25D]
2. The indirect register index for IND from 1,024 to 1,629 [400 – 65D]
3. The hybrid standard register IND RG, from 2,048 to 2,367 [800 – 93F]
4. The direct standard registers, from 2560 to 2879 [A00 – B3F]
5. The indirect Stack register index, from 3,072 to 3,088 [C00 – C10]
6. The direct Stack registers index, from 3328 to 3343 [D00 – D10]

Obviously, there's a few gaps of unused values, like between 2,368 and 2,559 – but you shouldn't be concerned with this at all; after all the parameters are entered automatically by the functions (totally transparent to the user), and it takes the same number of bytes to use a 4-digit number, regardless of its value.

Note that the status register Q(9) is used internally by the function's MCODE, and therefore should not be used in your FOCAL programs as synthetic register when the expanded registers functions are also used.

Part III - Managing the CL Expanded Y-registers

The CL board has three 4k-blocks of RAM memory reserved for extensions to the OS. So far these have been rather ignored by all applications, but finally they're put to a good use with the function sets included in this module.

This section covers the individual access of the expanded Y-registers included in the RAM block located at 0x801, i.e. the first of those three RAM blocks. You'll be able to store, recall, view, exchange, and perform ISG/DSE operations on 1,024 of those registers as if they were standard 41 registers within main memory. Note the presence of the arithmetic operations as well.

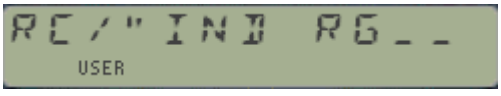
Extended Regs	Store	Recall	Other	X-Blocks
Y-Register From 0 to 1,023. *Use EEX for the Fourth digit field.*	**YSTO _ _ _**	**YRCL _ _ _**	**YX<> _ _ _**	**CLYRG**
	YST+ _ _ _	**YRC+ _ _ _**	**YVEW _ _ _**	**CLYRGX**
	YST- _ _ _	**YRC- _ _ _**	**YDSE _ _ _**	**YRGMOV**
	YST* _ _ _	**YRC* _ _ _**	**YISG _ _ _**	**YRGSWP**
	YST/ _ _ _	**YRC/ _ _ _**	**YFINDX**	**ST<>YRG _ _ _**
ALPHA	**YAST _ _ _**	**YARC _ _ _**		**A<>YRG _ _ _**

Besides the direct access, you also have the INDirect addressing capabilities implemented on the expanded registers; the sixteen Stack registers (including synthetic regs {M-e}); and all the standard-Data Registers - a hybrid mode, unique to this implementation.

Most of the functions will prompt for the parameters to use. The initial prompt is a three-field underscore for the Y-register indeed. Pressing [SHIFT] changes it to IND three-digit fields for another Y-register to be used as indirect. Pressing the [RADIX] key changes to the IND ST _ prompt, where you'll enter the register mnemonic, from T to e (all sixteen are available). Pressing the radix key again changes to a IND RG_ _ prompt where you can enter a standard register number to use as indirect address. Repeat pressings of the radix key act as a toggle between those two. There's also provision for direct stack and standard register arguments – even if those can be redundant in practice, being exactly the same as the original ones.

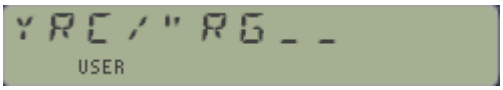
Once you complete the entry adding the register number the action is performed in RUN mode, or two program lines are entered in program mode – automatically selecting the appropriate parameter depending on the direct or indirect types. This is automatically done so you needn't (and shouldn't) edit the value entered in the program's second line at all – which will be properly interpreted in a running program.

You can move between the functions while the prompts are up; not only to select the math operation but also to change the main function amongst the group. So for instance during the YRCL _ _ _ main prompt pressing the [SST] key will trigger the **YX<>** function, or pressing [STO] will invoke the **YSTO** function instead. Also *you can revert to the original mainframe functions* pressing the corresponding key of the function in the prompt, for instance here pressing [RCL] will trigger the original RCL _ _



The functions will not allow you to enter any value greater than 1,023 either as direct index or indirect index – not even when entering them in a program line. Attempting to enter larger values will trigger a "NONEXISTENT" error message. However, that check is not made for IND_RG combinations, as there's no telling at that point about how many standard registers will be available at the execution stage.

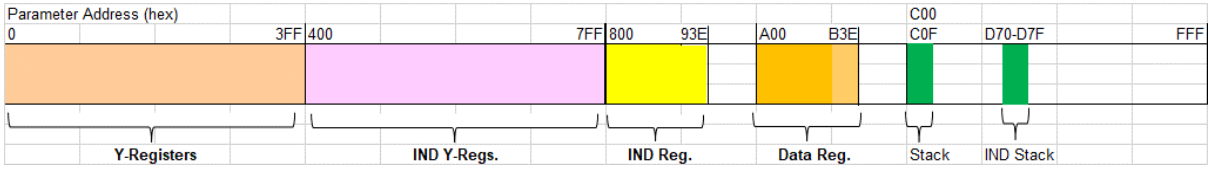
The usage of standard stack and data registers is not only more convenient from the usability standpoint, but also it enables the RCL math on these registers via the **YRCL** function:



Although possible, it is however not meant to be used in a program because of the obviously higher byte count. That's why when used in an editing program the direct stack and data register Y-functions *revert automatically to the native STO/RCL functions instead*, which has the additional benefit of a clearer representation by the OS as merged lines.

The table and chart below show all possible combinations for program editing:

Func.	Y-Reg nnn	adds 0x400 IND Y-reg nnn	adds 0xA00 Data Reg nn	adds 0x800 IND Data Reg nn	adds 0xD70 STK nn	adds 0xC00 IND STK nn
YSTO	nnn	nnn+1024	**STO nn**	nn+2048	**STO ST(nn)**	nn+3072
YST+	nnn	nnn+1024	**ST+ nn**	nn+2048	**ST+ ST(nn)**	nn+3072
YST-	nnn	nnn+1024	**ST- nn**	nn+2048	**ST- ST(nn)**	nn+3072
YST*	nnn	nnn+1024	**ST* nn**	nn+2048	**ST* ST(nn)**	nn+3072
YST/	nnn	nnn+1024	**ST/ nn**	nn+2048	**ST/ ST(nn)**	nn+3072
YRCL	nnn	nnn+1024	**RCL nn**	nn+2048	**RCL ST(nn)**	nn+3072
YRC+	nnn	nnn+1024	nn+2560	nn+2048	nn+3440	nn+3072
YRC-	nnn	nnn+1024	nn+2560	nn+2048	nn+3440	(nn+3072
YRC*	nnn	nnn+1024	nn+2560	nn+2048	nn+3440	nn+3072
YRC/	nnn	nnn+1024	nn+2560	nn+2048	nn+3440	nn+3072
YDSE	nnn	nnn+1024	**DSE nn**	nn+2048	**DSE ST(nn)**	nn+3072
YISG	nnn	nnn+1024	**ISG nn**	nn+2048	**ISG ST(nn)**	nn+3072
YX<>	nnn	nnn+1024	**X<> nn**	nn+2048	**X<> ST(nn)**	nn+3072
YVEW	nnn	nnn+1024	**VIEW nn**	nn+2048	**VIEW ST(nn)**	nn+3072
YARC	nnn	nmn+1024	**ARCL nn**	nn+2048	**ARCL ST(nn)**	nn+3072
YAST	nnn	mnn+1024	**ASTO nn**	nn+2048	**ASTO ST(nn)**	nn+3072



Note that the U/I won't allow entering Data Register indexes above 99. If values nn>99 are needed you should edit manually the second non-merged line, using the same rule as shown above. This will only apply to the YRCL_Math functions, as all others have a native OS equivalent.

Y-REG Data_Reg nn = Y-REG as first line plus (nn+2560) as second line.
Y-REG IND Data_Reg nn = Y_REG as first line plus (nn+2048) as second line.

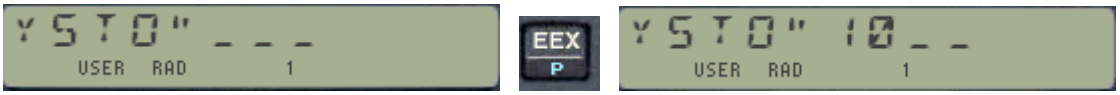
For example: YRC+ RG 111 will be: YRC+, plus "2670"
YRCL IND RG 111 will be: YRCL, plus "2149"

Storing and Recalling ALPHA Data

The expanded functions **YAST** and **YARC** provide the means to store and recall ALPHA data directly to/from the expanded registers area. Like their numeric counterparts, they support direct, INDirect, stack and standard registers indexes for a complete palette of options at your disposal. You can access these directly from the YSTO/YRCL prompts by pressing the ALPHA key at any time.

Going above 999

Revision 2G+ adds a convenient usability feature to the U/I, whereby the 3-digit prompt can be extended to a 4-digit, and furthermore the first 2 fields are automatically set to "10_ _". This is because there are only 1024 Y-Registers in Expanded RAM, thus the second digit must be zero.



At this point you should enter the remaining of the index value, ranging from 00 to 23. If values above 23 are keyed in it will result in a "NONEXISTENT" error message.

Deleting, Moving and Swapping Expanded Registers.

Think of the following functions as analogous to the X-Functions extensions on the original function set of the calculator, only applied to the expanded memory area instead.

- The function **CLYRG** will delete all the 1,024 expanded registers, and therefore it's equivalent to **YMCLR** used on the entire 0x801 block, as follows: "801000-0FFF", **YMCLR**

- Additionally with **CLYRGX** you can selectively delete a defined block of expanded registers as defined by its control word (in X) "bbb.eee:nnn", The bbb,eee digits denote the first and last expanded registers to clear, in the range R0 through R999. Finally nnn is the step number for registers to be cleared. If nnn is zero a value of one used.

  For example, the control word 25,250005 will clear every firth Y-register starting at YR 025 and ending at YR 250 (that is YR25, YR30, YR35, YR40, ….. YR245, YR250).

- **YRGMOV** and **YRGSWP** can be used to move or exchange a block of expanded registers at once – either contiguous or in an increment pattern as provided by the control word bbb.eee:nnn in the X-register. Much the same as the X-Functions RGMOVE and RGSWAP - in case you wonder, so here nnn denotes the block size to be moved or swapped.

Other Block Operations.

- **A<>YRG** and **ST<>YRG** exchange a group of five expanded registers with ALPHA (plus Q) or the Stack (T-L) respectively. The start register is to be entered at the prompt in manual mode, or expected to be in the X-Register when running a program. These functions do not allow INDirect indexing.

Moving around the Expanded Registers Functions.

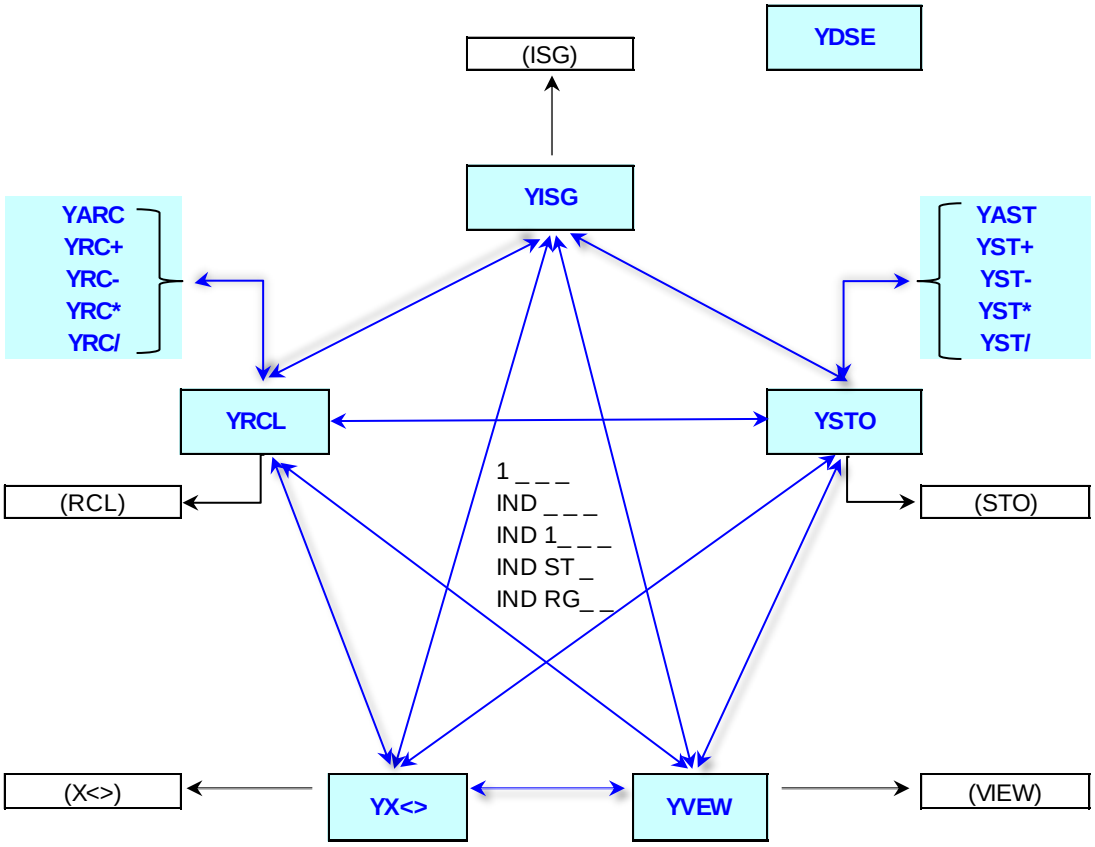


Figure 1.

Even if there isn't a dedicated launcher for these functions, navigation amongst them is as easy as intuitive. First off, assign one of the functions to its "natural" key, for instance **YSTO** to the STO key.

Then while the YSTO _ _ _ prompt is shown you can move about all the Y-Reg functions by pressing the key for the corresponding action, i.e.

- RCL will toggle to YRCL _ _ _
- SST (for X<>) will launch YX<> _ _ _
- R/S (for VIEW) will launch YVEW _ _ _
- CHS (for ISG) will launch YISG _ _ _
- ALPHA will trigger YAST _ _ _
- The math keys will launch the corresponding math function, ie. YST+ _ _ _
- EEX will add one field to the prompt: YSTO 1 _ _ _
- SHIFT will add the IND prompt (assuming EEX isn't already up)
- RADIX will add the ST _ prompt. All 16 status regs are selectable.
- RADIX again to toggle between ST _ and RG _ _. Choose any standard reg up to 99.
- STO again to exit to the native STO function (no way back!)
- Note that the DIRECT Stack/REG prompt is not strictly needed – that's the native function already. However, the RCL Math functions are useful and are available using this approach.

All the options above are available from within any of the 15 functions (**YDSE** is not part of this scheme) – regardless of which one you used to start the sequence. See the descriptions earlier in this manual for additional details.

Extra bonus: Finding the X-needle in the Y-haystack.

For those times when you'd like to know if a certain value is stored in the Y-data register, the sub-function **YFINDX** (a.k.a. **XF# 21**) is available to do a cursory comparison looking for a match with the value in the X-register. All Y-data registers are checked, starting with YR00 until YR1023 – which could take a long time depending on where the match exists.

The function returns the number of the first Y-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 stack register Y upon completion.

Below there is a FOCAL routine that checks up to YR999, as well as an equivalent routine for the standard data registers - for comparison purposes. See the Total_Rekall manual for yet another routine to tackle this "where is Waldo" problem using other advanced functions.

```
18  LBL "YFNDX"
19  .999
20  X<>Y
21  LBL 00
22  YRCL (IND Y)
23  3074
24  X=Y?
25  GTO 02
26  RDN
27  ISG X
28  GTO 00
29  CLX
30  -1
31  RTN
32  LBL 02
33  X<> Z
34  INT
35  END
```

```
22  LBL "FINDX"
23  SIZE?
24  E
25  –
26  E3
27  /
28  X<>Y
29  LBL 00
30  RCL IND Y
31  X=Y?
32  GTO 02
33  RDN
34  ISG Y
35  GTO 00
36  CLX
37  -1
38  RTN
39  LBL 02
40  X<> Z
41  INT
42  END
```

The possibilities of having an additional set of 1,024 registers available to your own programs are wide and deep. For starters you could permanently operate with a SIZE 000 and use all the 319 standard registers for User Code programs, key assignments and I/O buffers; so a few more bytes taken up by the parameter lines won't be a problem.

CODA: MCODE listing for YFINDX

Header	AB7E	*0D8*	*"X"*	
Header	AB7F	*044*	*"D"*	***Find Y-Register w/ X value***
Header	AB80	*04E*	*"N"*	*within the extended block*
Header	AB81	*049*	*"I"*	
Header	AB82	*046*	*"F"*	
Header	AB83	*059*	*"Y"*	*Ángel Martin*
YFINDX	**AB84**	*36D*	*PORT DEP:*	*check for CL mem*
	AB85	*08C*	*XQ*	*to avoid memory lost!*
	AB86	*09D*	*->A49D*	*[CLMEM?]*
	AB87	0F8	READ 3(X)	get target value
	AB88	158	M=C ALL	convenient storage
	AB89	130	LDI S&X	
	AB8A	3FF	CON: 1,023	
	AB8B	226	C=C+1 S&X	"400"
	AB8C	0E6	C<>B S&X	save pointer in B.X
	AB8D	0C6	C=B S&X	get current pointer
	AB8E	270	RAMSLCT	select register
	AB8F	226	C=C+1 S&X	point to next reg
	AB90	0E6	C<>B S&X	save pointer in B.X
	AB91	038	READATA	read contents
e long;	AB92	10E	A=C ALL	put in A for compares
egisters	AB93	198	C=M ALL	recall X value
!	AB94	36E	?A#C ALL	test for match
	AB95	***063***	***JNC +12d***	*found!*
	AB96	21C	PT= 2	
	AB97	210	LD@PT- 8	
	AB98	116	A=C XS	
	AB99	336	?A<B XS	reached the top?
	AB9A	*39B*	*JNC -13d*	*no, do next*
	AB9B	04E	C=0 ALL	*yes, NOT found*
	AB9C	2DC	PT= 13	
	AB9D	250	LD@PT- 9	*returns -1 instead*
	AB9E	050	LD@PT- 1	
	AB9F	0EE	C<>B ALL	
	ABA0	***05B***	***JNC +11d***	
	ABA1	046	C=0 S&X	
	ABA2	270	RAMSLCT	
	ABA3	066	A<>B S&X	
	ABA4	1A6	A=A-1 S&X	
	ABA5	130	LDI S&X	
	ABA6	3FF	CON: 1,023	
	ABA7	226	C=C+1 S&X	"400"
	ABA8	1C6	A=A-C S&X	
	ABA9	*1F5*	*?NC XQ*	
	ABAA	*0C4*	*->317D*	*[BIN--D]*
	ABAB	*0B9*	*?NC GO*	
	ABAC	*04A*	*->122E*	*[RCL]*

Converting Standard Programs

Having a complete function set ensures you can convert programs very easily, simply by replacing the standard functions with their expanded version. As of release 2B even the ALPHA storage functions **YAST** and **YARC** are included, which can also use the expanded register range.

Then you have the benefit of a much larger set of registers available for your program, a sheer advantage to manage larger size cases of the problem you're trying to solve – from matrix operations to sorting data, to mention just a couple.

For example, with a few modifications the PPC ROM programs **S2** and **S3** can be used to sort more than 1,000 registers in a very efficient way. – with random data populating those registers it took about 32 seconds to sort 1,000 registers on TURBO 50 mode!

See below two simple routines I used to populate the registers and to view them. They expect the control word bbb.eee in X before you run them.

01	**LBL "YVIEW"**
02	LBL 00
03	"YR"
04	ARCLI
05	"|-: "
06	**YRCL (IND X)**
07	**3075**
08	ARCLX
09	AVIEW
10	PSE
11	RDN
12	ISG X
13	GTO 00
14	END

01	**LBL "YRAN"**
02	RCL X
03	LBL 00
04	RNDM
05	**YSTO (IND Y)**
06	**3074**
07	RDN
08	ISG X
09	GTO 00
10	RDN
11	END

Functions **ARCLI** and **RNDM** are available in the AMC_OX/X Module.

Note.- In case you're interested, the parameter lines used by these functions correspond to the following: (with SIZE 319 for the standard registers case)

7. The register index for direct access, from 0 to 1,023 [000 – 3FF]
8. The indirect register index for IND from 1,024 to 2,047 [400 – 7FF]
9. The hybrid standard register IND RG, from 2,048 to 2,367 [800 – 93F]
10. The direct standard registers, from 2560 to 2879 [A00 – B3F]
11. The indirect Stack register index, from 3,072 to 3,088 [C00 – C10]
12. The direct Stack registers index, from 3328 to 3343 [D00 – D10]

Obviously, there's a few voids like between 2,368 and 2,559 – but you shouldn't be concerned with this at all; after all the parameters are entered automatically by the functions (totally transparent to the user), and it takes the same number of bytes to use a 4-digit number, regardless of its value.

Note that the status register Q(9) is used internally by the functions MCODE, and therefore should not be used in your FOCAL programs as synthetic register when the expanded registers functions are also used.

Warning: The expanded Register functions are not to be used if the first block is already used as back-up location. Use blocks #2 and #3 instead!