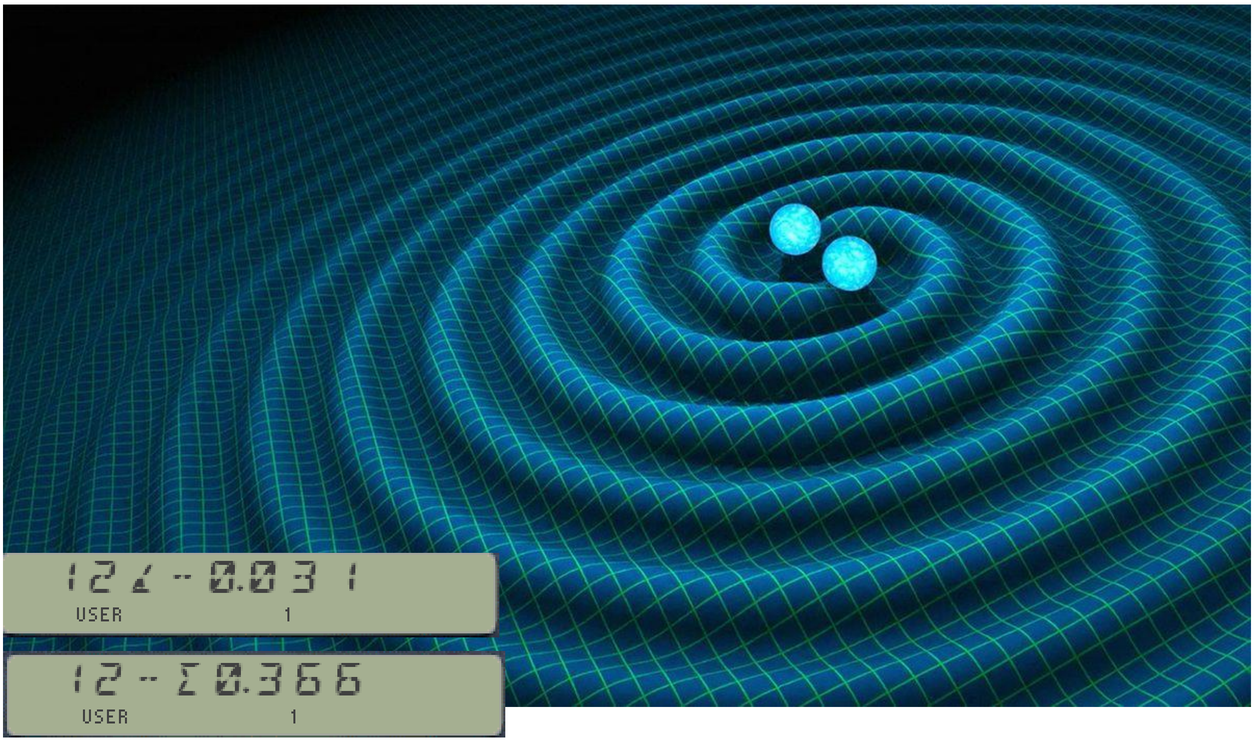
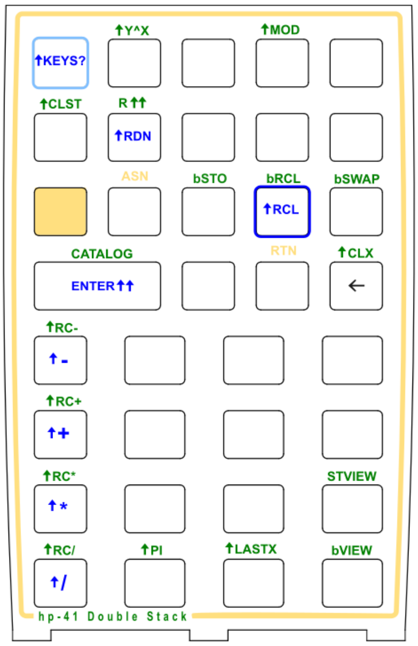
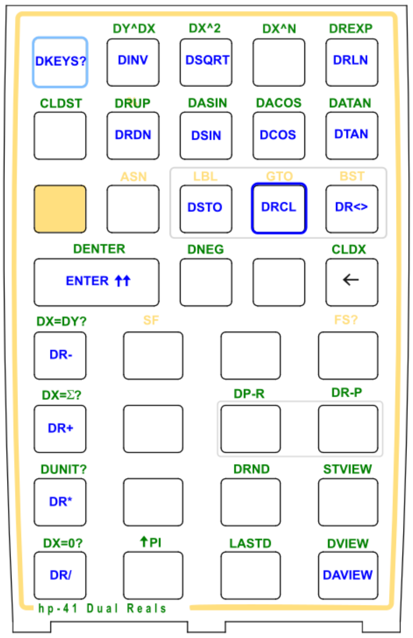
DOUBLE-DOWN ROM
HP-41 MODULE

Double Length Stack and Dual Real Numbers





Double-Length Stack

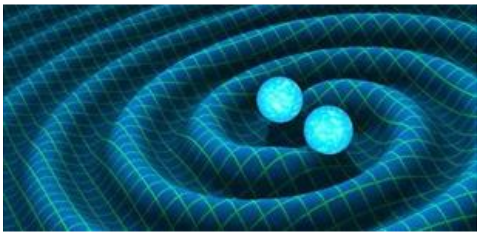


Dual Number ROM

Programmed by Ángel M. Martin

April 2022

Double-Down ROM
HP-41 Module



Introduction.

Welcome to the Double-Down Module for the HP-41, perhaps an exercise in futility for some but also the last opportunity to equip our trusty companion with the Double length Stack that you've always dreamed of but were too coy to ask. And a **Double-Length stack** you'll find in here, complete with all functions needed to support its double size in all the automated stack lift, stack drop and register roll & duplication actions performed by the O/S behind the scenes when you use the calculator. The double stack is 9 registers long; that is four more than the standard {XYZTL}. The additional four registers {ABCD} are located in a dedicated buffer stored in the I/O area of the calculator's RAM, but you'd never know it's there as it is completely transparent to the user.

Some precursor FOCAL programs were written in the past (see for instance Valentín Albillo's article in here: HP Letter 1980-09-27 - Letter from Valentin Albillo to John McGechie - 070588-90.pdf) ; however a smooth implementation really requires MCODE to iron-out all the small wrinkles created by the user code restrictions, really not cut for a low-level control of the stack registers – and frustratingly slow.

This alone would have made a nice mini-module but there was yet another field pending on the task list that fitted perfectly in the design. That field may sound corner-case or too strange but it presented itself as a logical candidate to reusing lots of code and techniques from the 41Z Complex Number Module. We're referring to the **Dual Real numbers** (the strange cousin of the complex number); expressions of pairs of two real numbers arranged in specific manner and following well-defined mathematical rules – as you can see in this general overview: https://en.wikipedia.org/wiki/Dual_number

The implementation is very comprehensive and surprisingly detailed for such a vague subject; all stops have been pulled to make using it a rewarding experience for the math-inclined HP-41 user wishing to get on new rides after all these many years. Are you intrigued yet?

Why the dual game?

Although they are two very different areas, both share common structural details that lend themselves to a joint implementation – taking advantage of the very many routines required to manage the buffer and the data storage & retrieval. I somehow always think of the FORTH/Assembler ROM for the HP-71 as a similar concept, merging two distinctly different but very complementary subjects under the same hood. All in all, a perfect fit from the programming side, and certainly also a very nice doble-whammy in a single 4k module, so grab them while they last!

Dependencies.

This module is designed to be used on a CX O/S with the **Library#4** (**revision R59** or higher) plugged in. No other dependency exists. The XROM id# is "1", thus you can't have this module plugged simultaneously with the 41Z.

Without further ado, here is a list of the functions in the ROM's FAT table.

XROM#	Function	Description	Stack Impact / Specials
01.00	**-DBL STACK**	*Section header*	*n/a*
01,01	**^KEYS?**	Bulk Key assignments	
01,02	**^+**	8-level stack addition	*Drops stack, Duplicates D:*
01,03	**^-**	8-level stack subtraction	*Drops stack, Duplicates D:*
01,04	**^***	8-level stack product	*Drops stack, Duplicates D:*
01,05	**^/**	8-level stack Division	*Drops stack, Duplicates D:*
01,06	**^CLST**	Clears all stack levels (including LastX)	
01,07	**^CLX**	Clears X: level	*Disables stack lift*
01,08	**ENTER^^**	Pushes X: up into 8-level stack	*Lifts stack, disables stack lift*
01,09	**LASTX^**	Recalls LastX value to X:	*Lifts stack*
01,10	**^MOD**	Calculates Y MOD X	*Drops stack, Duplicates D:*
01,11	**^PI**	Puts π in X,	*Lifts stack*
01,12	**R^^**	8-level stack Roll Up	*Rolls Up*
01,13	**^RDN**	8-level stack Roll Down	*Rolls Down*
01,14	**^RCL _ _**	Recall value to 8-level stack	*Lifts stack*
01,15	**^RC+ _ _**	Adds value to contents in X:	
01,16	**^RC- _ _**	Subtracts value from contents in X:	
01,17	**^RC* _ _**	Multiplies value with contents in X:	
01,18	**^RC/ _ _**	Divides content in X by value	
01,19	**^STVIEW**	Enumerates all 8-level registers	
01,20	**^Y^X**	8-level Power function	*Drops stack, Duplicates D:*
01,21	**bRCL _**	Recalls buffer register to X	*Lifts stack*
01,22	**bSTO _**	Stores X in buffer register	
01,23	**bVIEW _**	Views contents of buffer register	
01,24	**bSWAP _**	Swaps X: and buffer register	
01,25	**-DUAL REAL#**	*Section header*	*n/a*
01,26	**CLDST**	Clears Dual stack (DL included)	
01,27	**CLDX**	Clears DX level	*Disables stack lift*
01,28	**DAVIEW**	*Presents Dual Number in LCD*	
01,29	**DENTER^**	Pushes DX: up into dual stack	*Lifts stack, disables stack lift*
01,30	**DKEYS?**	Bulk Key assignments	
01,31	**DR+**	Dual Number Addition	*Drops stack, Duplicates DT*
01,32	**DR-**	Dual Number Subtraction	*Drops stack, Duplicates DT*
01,33	**DR***	Dual Number Product	*Drops stack, Duplicates DT*
01,34	**DR/**	Dual Number Division	*Drops stack, Duplicates DT*
01,35	**DR^N**	DN Power to Integer	
01,36	**DRCL _ _**	Recall Dual Number to Stack	*Lifts Dual Stack*
01,37	**DRDN**	Dual stack Roll Down	*Rolls Down Dual Stack*
01,38	**DREXP**	Dual Exponential function	
01,39	**DRINV**	Dual Inverse function	
01,40	**DRLN**	Dual Logarithm function	
01,41	**DRND**	Dual Number Rounding	
01,42	**DRNEG**	Dual number Sign Change	
01,43	**DRSQRT**	Dual Square Root function	
01,44	**DRUP**	Dual stack Roll Up	*Rolls Up Dual stack*
01,45	**DSTO _ _**	Stored DX into target dual register	*Prompt for target*

01,46	**DUNIT?**	Tests if DX is in the Unit "Circle"	*Skips Line if False*
01,47	**DX=0?**	Tests if DX=0	*Skips Line if False*
01,48	**DX=Σ?**	Tests if DX=ε	*Skips Line if False*
01,49	**DX=DY?**	Tests if DX=DY	*Skips Line if False*
01,59	DX<> _ _	Exchanges DX and target register	*Prompts for target*
01,51	**DX<>DY**	Exchanges DX and DY	
01,52	**DX^2**	Squares DX value	
01,53	**DY^DX**	Dual Number Power function	*Drops stack, Duplicates DT*
01,54	DVIEW _ _	Views Dual number in Target dual register	*Prompt for Target*
01,55	**LASTD**	Recalls DL into DX dual stack level	*Lifts Dual stack*
01,56	**DR-P**	Converts Gaussian to Polar	
01,57	**DP-R**	Converts Polar to Gaussian	
01,58	**DSIN**	Dual Sine function	*Angular mode independent*
01,59	**DASIN**	Dual Inverse Sine function	*Angular mode independent*
01,60	**DCOS**	Dual Cosine function	*Angular mode independent*
01,61	**DACOS**	Dual Inverse Cosine function	*Angular mode independent*
01,62	**DTAN**	Dual Tangent function	*Angular mode independent*
01,63	**DATAN**	Dual Inverse Tangent function	*Angular mode independent*

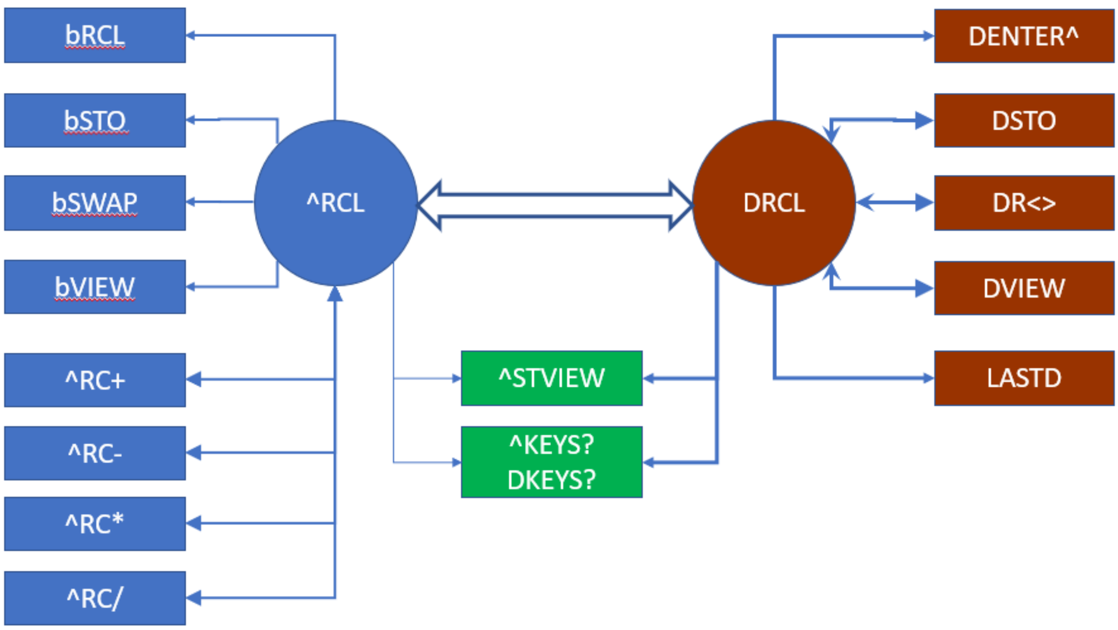
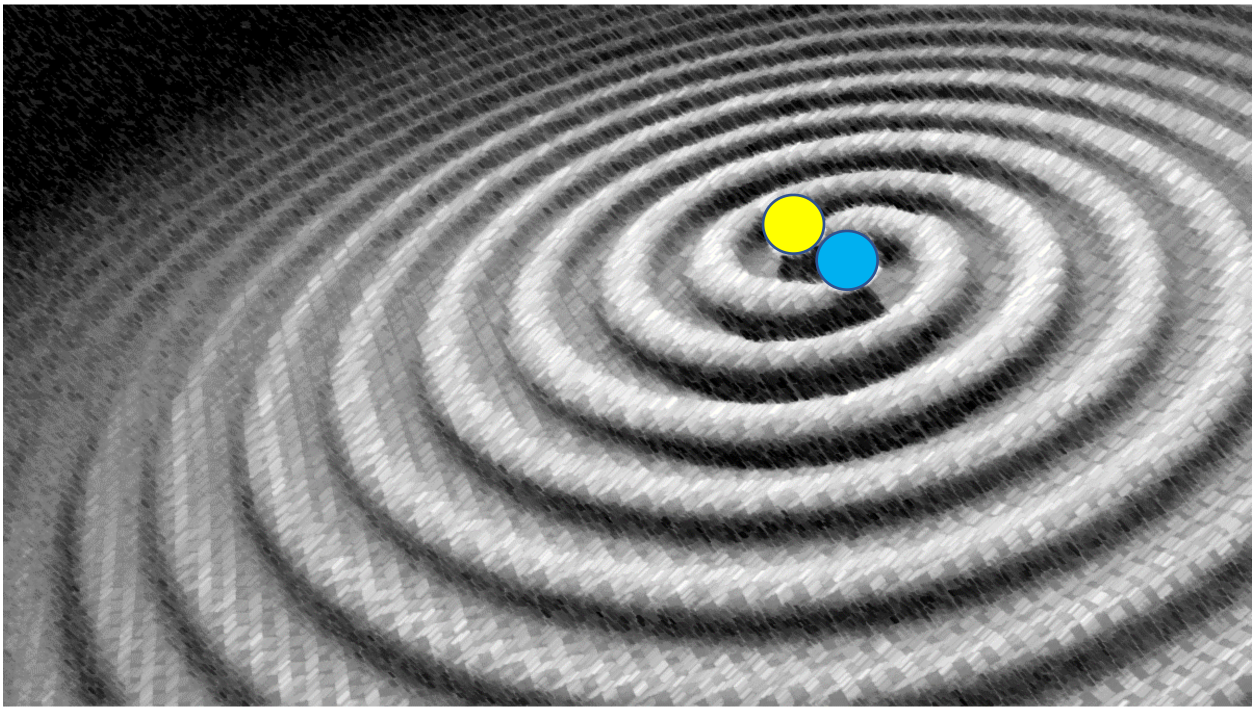


fig. 0.1.- Launcher Interrelationship

Part I – Double-Length Stack



Double Length Stack and Dual Number Stack

The same I/O buffer object caters for the needs of both the double-stack and the dual-number stack. The buffer has **seven** registers named a:, b:, c:, d:, e:, F: and G: - For the double stack the first four are mapped to the additional stack registers { A, B, C, D } and the topmost register G: is used to hold the LGKT (more about this later) - whereas buffer regs E: and F: are unused.

For the dual-number stack on the other hand the first six buffer registers are used, taking two of them for each dual-number stack level as show in the figure below. Note that the Dual number LastX level (DL) is situated at the top of the dual stack, and that the native LastX register L is not part of the construction (used for scratch only). This design doesn't require any ALPHA registers or Data Registers in RAM, which can be freely used for FOCAL programs or any general purpose.

	REAL	Register	DUAL	
	LGKT	G:	*Scratch*	
	Scratch	F:	DL	
	Scratch	e:		
	D	d:	DT	
	C	c:		
	B	b:	DZ	Dual Number
Double	A	a:		Stack
Stack	T	T:	DY	
	Z	Z:		
	Y	Y:	DX	
	X	X:		
	L	L:	*Scratch*	

There are four functions available to access the buffer registers, as follows:

- bSTO - to store the value in X into the buffer register given at the prompt
- bRCL - to recall to X the value in buffer register given at the prompt (*)
- bSWAP - to exchange X and the buffer register given at the prompt
- bVIEW - to view (no data movement) the contents of buffer register at the prompt.

They are prompting functions, asking for the letter of the register { a-e, F, G } in manual mode or expecting a number { 1-7 } as a second line in program mode. This line is added automatically by the function when entered in the program.

*(*) bRCL performs a **double-stack lift**, adhering so to this module's convention.*

Warning note:

The buffer id#7 is the same one used by the "Shadow Stack" in the WARP_Core module, and for the SHFL functions in the WARP_Core and the Formula_Evaluation modules. You should refer to their respective manuals for details. Bear this in mind if you want to use any of those modules together with the Double-Down ROM.

Part I - The Double Stack.

In this section we'll describe the available new functions to manage the double stack. As mentioned before, the double length stack has 9 registers in total: one (L) is for the LastX functionality and eight for the actual stack levels. The stack levels are ordered in the sequence { XYZTABCD } , which is rigorously respected in all (automatic and manual) stack lift and drop actions. For all means and purposes the double-stack is a continuous logical entity as far as the user is concerned, so no worries about keeping track of the actual whereabouts of the data within the calculator's memory.

Managing the Double-Length Stack

There are two groups of functions available to support the automated implementation of the Double Stack:

1. Stack Management functions, and
2. Math functions required to support the double length.

The table below shows the functions grouped by this criterion:

Stack Management		Math Functions	
XROM #	Function Name	XROM #	Function Name
01,06	↗CLST	01,02	↗+
01,07	↗CLX	01,03	↗-
01,08	ENTER↗↗	01,04	↗*
01,09	LASTX↗	01,05	↗/
01,12	R↗↗	01,10	↗MOD
01,13	↗RDN	01,11	PI↗
01,14	↗RCL _ _	01,20	↗Y↗X
01,15	↗RC+ _ _		
01,16	↗RC- _ _	Admin Functions	
01,17	↗RC* _ _	01.,01	↗KEYS?
01,18	↗RC/ _ _	01,19	↗STVIEW
04,26	XFILL↗		

Already you see the reoccurring theme in the used naming conventions, adding the "↗" character either as prefix or postfix of the name depending on whether they drop or lift the double stack – and to differentiate it from the native function with the same purpose.

Notice the presence of RCL Math functions, not available in the native function set but added here for your convenience. Since they condense two operations in one, chained operations are much more efficient and require fewer program steps.

The ↗RCL function group is a particularly powerful set, supporting indirect and stack addressing across the whole range of double-stack and data registers. More about this later.

All functions are meant to be used as *direct replacement of the native ones*, just use them instead of the original, 4-level stack versions. No additional steps, no modifications needed so you already know how to use them and when. We'll see a couple of examples next.

Example.

Compute $1 + 2 * 2.5\text{^}(3/7)$ doing a left-to-right data entry (and so resisting the urge to process the expression from "inside-out", as learned in all your years of RPN usage):

We type:

1 , ENTER^^ , 2 , ENTER^^ , 2.5 , ENTER^^ , 3 , ENTER^^, 7 , ^/ , ^Y^X , ^* , ^+

Obtaining: => 3.961936296

Example.

Using a *strict left-to-right approach* calculate the stack-breaker expression shown below:
(taken from the WP-34 manual, pg. 20)

$$\sqrt{\frac{1+\left|\left(\frac{30}{7}-7.6\times 0.8\right)^{4}-\left(\sqrt{5.1}-\frac{6}{5}\right)^{2}\right|^{0.3}}{\left\{sin\left[\pi\left(\frac{7}{4}-\frac{5}{6}\right)\right]+1.7\times(6.5+5.9)^{3/7}\right\}^{2}-3.5}}$$

Roll up your sleeves and start typing as follows:

RAD , 1 , ENTER^^ , 30 , ENTER^^ , 7 , ^/, 7.6 , ENTER^^ , 0.8 , ^* , ^-, 4 , ^Y^X , 5.1 , SQRT, 6 , ENTER^^ , 5 , ^/, ^-, X^2 , ^-, ABS , 0.3 , ^Y^X, ^+,

Numerator done, 2.948814831 ... take a deep breath and continue:

PI^ , 7, ENTER^^ , 4 , ^/, 5 , ENTER^^ , 6 , ^/ , ^- , ^*, SIN , 1.7 , ENTER^^ , 6.5 , ENTER^^ , 5.9 , ^+, 3 , ENTER^^ , 7 , ^/ , ^Y^X , ^* , ^+, X^2 , 3.5 , ^-,

denominator done, 24.16586119 - ready for the finishing touch now:

^/, SQRT => 0.349319325

Example.

Using a *strict left-to-right approach* calculate the Mach number using the formula below:

$$\sqrt{5\left(\left(\left(\left(\left(1+0.2\left(\frac{350}{661.5}\right)^{2}\right)^{3.5}-1\right)\times\left(1-6.875\times10^{-6}\times25500\right)^{-5.2656}\right)+1\right)^{0.286}-1\right)}$$

The result is => 0.835724536

5 , ENTER^^, 1 , ENTER^^, 0.2 , ENTER^ , 350 , ENTER^^ , 661.5 , ^/ , X^2 , 3.5 , ^Y^X , 1 , ^- , 1 , ENTER^^ , 6.875 E-6 , ENTER^^ , 25500 , ^* , ^- , 5.2655 . CHS , ^Y^X , 1 , ^+ 0.286 , ^Y^X , 1 , ^- , ^* , SQRT

Why not a 10-level stack, given that the buffer holds enough registers for it?

First off, because with the eight levels provided you will be on the safe side even with the most advanced equations to compute in your life as a scientist or engineer. Second, the larger the stack the less useful it is for the automated duplication of the topmost register; and finally, this is the way to ensure commonality between the double-stack and the dual-number stack implementations for buffer lifting and dropping actions – since the same number of buffer registers are involved on either one of the two cases.

Using the double-length functions in a Program.

The single one exception to the "same as the native stack" rule happens when using the functions in a running program. In this case the automatic upper-stack lift when executing numeric steps does not occur, so you need to do it manually. This means that an **ENTER^^** *instruction must precede *every* numeric program step*. Read the next section to understand the reasons for this convention, but before enjoy the FOCAL program example:

Program listing.

```
01 LBL "TST"
02 RAD
03 1
04 ENTER^^
05 30
06 ENTER^^
07 7
08 ^/
09 ENTER^^
10 7.6
11 ENTER^^
12 0.8
13 ^*
14 ^-
15 ENTER^^
16 4
17 ^Y^X
18 ENTER^^
19 5.1
20 SQRT
21 ENTER^^
22 6
23 ENTER^^
24 5
25 ^/
26 ^-
27 X^2
28 ^-
29 ABS
30 ENTER^^
31 0.3
32 ^Y^X
33 ^+
34 ^PI
35 ENTER^^
36 7
37 ENTER^^
38 4
39 ^/
40 ENTER^^
41 5
42 ENTER^^
43 6
44 ^/
45 ^-
46 ^*
47 SIN
48 ENTER^^
49 1.7
50 ENTER^^
51 6.5
52 ENTER^^
53 5.9
54 +
55 ^+
56 ENTER^^
57 3
58 ENTER^^
59 7
60 ^/
61 ^Y^X
62 ^*
63 ^+
64 X^2
65 ENTER^^
66 3.5
67 ^-
68 ^/
69 SQRT
70 END
```

There you have it, not rocket science but a solid tribute to the original designers of the RPN system.

Note the steps in red denoting the sprinkled **ENTER^^** instructions (10 in total), always preceding a numeric program step. These are needed for running program support. Those are not required in manual calculation mode, by virtue of the IO_SVC interrupt trick.

Ensuring a seamless operation: The "Last-T" Register

To ensure a seamless operation of the 8-level stack we're going to need a trigger for "upper stack repair" actions, caused by the user entering numeric data before the execution of the dual-stack functions. To illustrate this condition, consider the following situation:

Say that the double stack holds a relevant value in register T. We know that double-stack friendly functions like **ENTER^^ , LASTX^** or **PI^** will take care of pushing the T: register value over the stack divide and into the A: register; therefore, all is good. But the problem arises when the user calls a function or performs an operation that only alters the lower-stack {XYZT} arrangement before the next double-stack friendly function is called to properly manage things.

And no, this isn't a far-fetched scenario at all, happening only with rogue functions - since *one of those stack-altering operations is entering a new numeric value in X* – just pressing any number key in the calculator with stack lift enabled will push the T: register value off the stack, and thus irrevocably lost for the double-stack functionality. And this happens all the time! So not good, now you'd agree how true is the adagio "*the (stack) devil is in the details*", eh?

The solution is a two-pronged mechanism that will (1:) keep a backup copy of the value in T: in a safe place, so that (2:) recovers the backup when the need to update the upper stack occurs. The fixing sequence would be restoring the "last-good-known" T: value (LGKT), pushing it onto the upper-stack register A: - with the subsequent upper-stack lift.

Obviously, this necessitates the storage of such LGKT value in the safe place, and it should be done *before* the user enters a new value in X. The how and when must meet this, and the only way to do it is as follows:

- Upon the CALC_ON event, a reset is done where the "initial" value in T: is copied to the LGKT location. Obviously, this must be repeated every time the T: register is modified. This is ensured by updating it every time T: changes, and with the module functions we're in control because this only happens when a two-number function or a stack admin routine is called, all of which are double-stack friendly and thus their code (upon completion) saves the new value in T: into the LGKT location in case it should need using afterwards.

- Each and every operation (be that double-stack function or not) *needs to check whether the conditions exist for a potential T-override, i.e. entering a new value with stack lift enabled*. Well, for our own functions in the module we could do an initial check on the status of User fag 22 (the Data Entry flag) to figure out if a digit entry had been done prior, and deal with it in a postponed fashion, lifting the upper-stack registers and copying the LGKT into the A: register. But this is not the only instance of trouble, as the offending action could have been done multiple times, *overwriting T more than once* - yet the UF 22 approach wouldn't know that. Consider for instance the 4-step sequence: { 16, SQRT, 4, X^2 }. This pushes the stack twice, leaving previous Y,X in T,Z; "4" in Y and "8" in X, – and therefore the original contents of Z and T have gone to see the wizard and are irrevocably lost. By the time we get to a double-stack friendly function (such as **R^^, ENTER^^** or whatever) it's too late for our repair action!

- So the UF 22 approach isn't going to cut it, and clearly we don't have any control over the native functions from our module, therefore we need to resort to a more potent method that keeps tabs with *every* key sequence pressed, then decide whether it's one of those creating the problem – and right at that moment perform *in-situ* the upper-stack repair, every time it's needed instead of postponing it for when the double-stack function comes to the picture (if it ever does). This, in MCODE parlance is called making use of the I/O_Service interrupt polling point, and sure enough this is how it works in the module.

- One last detail is crucial for the correct operation of the scheme. We've already mentioned that the stack-repair action is only needed when the data entry is done with stack lift mode enabled; and not for instance right after pressing **ENTER^^** or **^CLX**. Here the O/S has the benefit of having CPU F11 clear when the first number digits is entered, so it knows there's no stack lift to do. But by the time our module receives the baton (via the IO_SVC interrupt), F11 is set again (done by the digit entry itself), thus we missed the point completely. We need another way to tell when not to follow the general rule (upper-stack correction), and that we have solved by anointing the user flag 01 as the "double-stack lift mode flag", i.e. a replica of CPU F11 but persisting until the next double-stack function is run. If UF 01 is set, the correction is done but if it's clear then it's skipped.

For additional information, the code beeps a short tone when the correction is done, so you can always tell when the upper stack is being "restored" to the desired status. Here's the upper-stack repair action described in detail:

[Desired] = [Actual] + [Pending] ; and: [Actual] is done now, [Pending] is postponed

REG#	Values	Action:	Desired	Actual	Pending	
LGKT:	(T)	User inputs a numeric value, "N"	(Z)	(T)	(Z)	Write the new LGKT value
D:	D		C	**D**	C	Moved from C:
C:	C		B	**C**	B	Moved from B:
B:	B		A	**B**	A	Moved from A:
A:	A		T	**A**	(T)	Recovered from LGKT
T:	T		Z	Z	-	*no action*
Z:	Z		Y	Y	-	*no action*
Y:	Y		X	X	-	*no action*
X:	**X**	**N**	**N**	N	-	*no action*

Case Scenarios:	How it's handled in the module:
(1) Automatic Stack Lift	
Keying a number value from the keypad	via the IO/SVC Polling point
RCL, LASTX, PI	use **^RCL**, **^LASTX**, and **^PI** instead
CPU F11 clear disables stack lift	UF 01 signals a previous F11 clear
(2) Automatic Stack duplication:	
Two-number math functions	use the module versions of the same
+ ,- , * , / , Y^X, MOD	**^+** , **^-** , **^*** , **^/** , **^Y^X** , **^MOD**
(3) Other Stack altering functions:	
ENTER^, RDN and R^	use **ENTER^^** , **^RDN** and **R^^** instead

In summary, keep the double-stack functions close to your heart (i.e. always used then instead of the native, lesser 4-level stack counterparts) and don't worry about anything else – it's all taken care by the DoubleDown module.

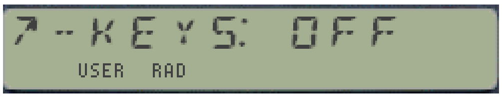
And this is indeed a good segue way into the bulk key assignments facility …

Bulk Key Assignments for Double Stack functions.

You can use the function **^KEYS?** to do a bulk assignment or removal for all double-stack functions available in the Module, each going to their "natural" key location to replace the native lesser counterparts. The function prompts Y/N? for the assignment or removal of the KA's, therefore the question mark in the function name.

, only "Y" or "N" are accepted here:

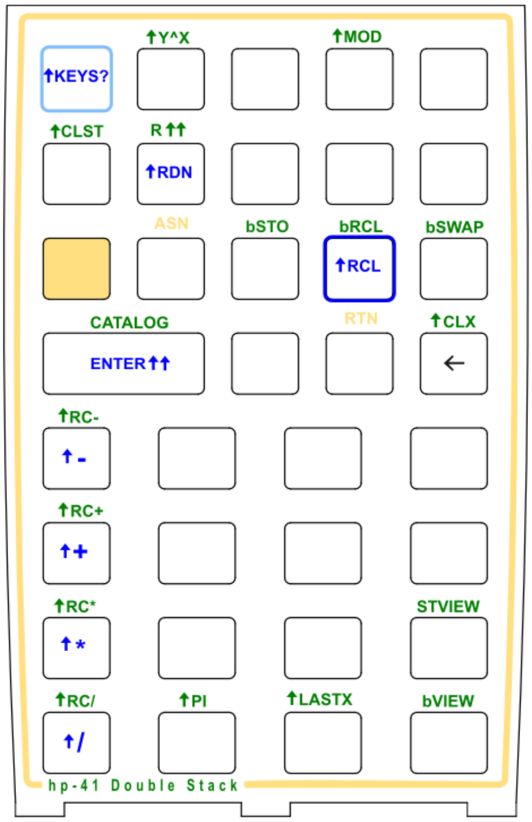
or:



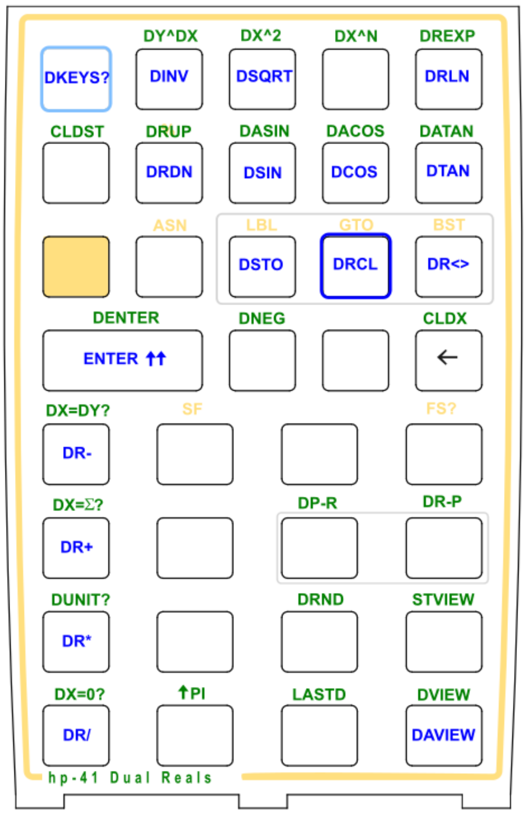
The KA's removal action is followed by a memory PACKING to recover the KA's registers freed.

Note that only the KA's on affected keys will be changed; any other KA on another key will not be modified so you can continue to use them.

See below the complete keyboard assignments made by **^KEYS?** and its counterpart **DKEYS?** That will be covered in the Dual-Number section of the manual.



Double-Length Stack



Dual Number ROM

The New Recall, now double-length stack aware.

An important addition to the module has been a new version of the RCL function present in the SandMath and WARP_Core modules. Like those, the new **^RCL** includes the in-place math operations so sorely missing in the native function set, i.e. **^RC+**, **^RC-**, **^RC*** and **RC/** . But in addition to that, the new **^RCL** is also fully double-stack aware, thus not only the double-stack is lifted respecting the T/A divide, but also (and better yet) it can be used to read data stored *anywhere* (*) in the calculator's standard RAM, be that data registers Rnn or double-stack registers { X-F }. You can also use any of these as pointer for the IND and IND ST operation, which is not a small feat if you think about it – and in fact requires a substantial amount of code to pull it off.

(*) With the single exception of the status registers above "Q", i.e. { a, b, c, d, e, K}. They had to yield their place to the newcomers, the buffer registers { A, B, C, D, E , F }. Not a complete loss, you can still access them using the standard RCL w/ the AMC_OS/X module plugged in of course.

; 

; etc.

The U/I is smart enough to allow for in-situ changes between the different functions, just pressing the arithmetic keys or the RCL key while the prompt is displayed. Try it to get the feeling of the operation.

In terms of Stack-lift properties **^RCL** behaves exactly like the simpler **^LSTX** and **^PI -** the stack will be lifted when F11 is set, but not if F11 is clear. Unlike the simpler two, ^RCL internal code re-uses F11 so its status is transferred to UF 01 upon initialization, and UF 01 is used at the end to decide whether to lift the buffer. As always, UF 01 is left set upon completion (by virtue of the synchronization routine to refresh the Last-T register).

Example. Store the value -44 in the data register R04, and then use stack register "D" to retrieve R04's value using the indirection capability of ^RCL

-44 , STO 04 , ^CLX , 4 , bSTO "d" , ^RCL IND D => -44.000000000

So now you have the capability to recall values from any register, including stack and indirect addressing, but *how can you store them in the upper-stack registers to begin with?*

Clearly there's no ^STO companion function for that, but the module comes with buffer-register handling functions to manage the contents of the upper-stack registers, even if in a way separate from the rest.

- bSTO - to store the value in X into the buffer register given at the prompt
- bSWAP - to exchange X and the buffer register given at the prompt
- bVIEW - to view (no data movement) the contents of buffer register at the prompt.

They are prompting functions, asking for the letter of the register { a-e, F, G } in manual mode or expecting a number { 1-7 } as a second line in program mode. This line is added automatically by the function when entered in the program.

With these dedicated functions most of the use cases are well covered, with the exception of the view, storage and exchange actions with indirect stack addressing for the upper-stack registers.

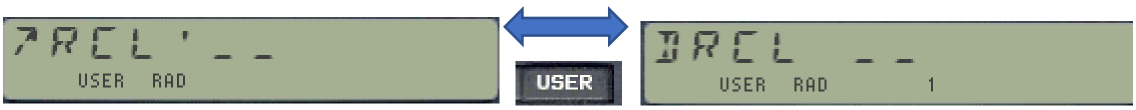
Summary of Memory handling functions and use cases

^RCL	**^RCL _ _**	for Data Regs Rnn<199
	^RCL ST _	for Stack {XYZT|L} and Buffer {ABCD|EF|G}
	^RCL IND _ _	for ALL Data Regs Rnn
	^RCL IND ST _	for Stack {XYZT|L} and Buffer {ABCD|EF|G}
	bRCL _	for buffer {ABCD|EF|G}, but not needed.
STO	**STO _ _**	for Data Regs Rnn
	STO IND _ _	for ALL Data Regs Rnn
	STO ST _	for Stack {XYZT|L}
	bSTO _	for buffer {ABCD|EF|G}
	STO IND ST _	for Stack {XYZT|L}
X<>	**X<> _ _**	for Data Regs Rnn<199
	X<>IND _ _	for ALL Data Regs Rnn
	X<> ST _	for Stack {XYZT|L}
	bSWAP _	for buffer {ABCD|EF|G}
	X<>IND ST _	for Stack {XYZT|L}

Besides being a very powerful function by itself, **^RCL** has a double-duty role as a launcher for the "cluster" of related functions, as follows:

	STO	**bSTO _**		STO	**DSTO _ _**	
	R/S	**bVIEW _**		R/S	**DVIEW _ _**	
	SST	bSWAP _		SST	DR<> _ _	
^RCL _ _	USER	**DRCL _ _**	→	USER	**^RCL _ _**	*(*) full circle!*
	ENTER	**ENTER^^**		ENTER	**DENTER^**	
	PRGM	**^STVIEW**		PRGM	**^STVIEW**	
	XEQ	**^KEYS?**		XEQ	**DKEYS?**	

(*) Both RCL launchers are interconnected with each other, so you have access to all of the "parallel" cluster functions also starting from this launcher.



Lest we forget the in-place ^RCL-Math functions as well:

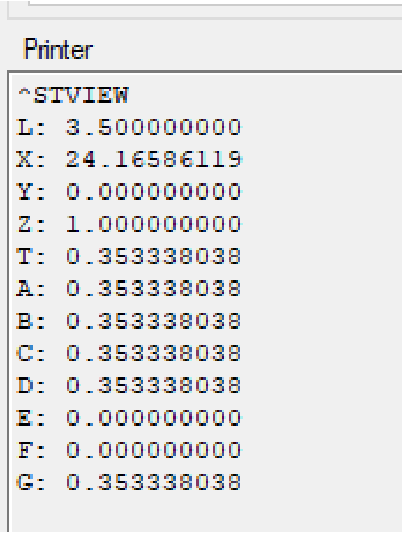
	+	**^RC+ _ _**
	-	**^RC- _ _**
^RCL _ _	*	^RC* _ _
	/	**^RC/ _ _**
	RCL	^RCL _ _

This justifies that the buffer register functions { **bRCL**, **bSTO**, **bSWAP** and **bVIEW** **}** are not assigned by **^KEYS?** in the bulk option: having too many key assignments takes more I/O memory and clutters the USER keyboard with conflicting function for program operation. It's not a problem since they can still be conveniently accessed to via the **^RCL** route.

Snooping the Double Stack with **^STVIEW**

The module comes with its own spyware application so you can always look at the COMPLETE double stack not altering the register order or contents. The function **^STVIEW** will produce a sequential enumeration of all registers, with the register name preceding the values.

You can halt the listing be pressing any key, and the numeration will continue after you release it. If a printer is connected the listing will be printed with user flag F15 set, as you can see in the snapshot from the ILPER below:



Note how register G: has a copy of the T register, in its LGTK role explained before.

Filling the Double-Stack with the value in X:

The automatic D: register duplication on stack drop is a handy feature for diverse arithmetic calculations such as polynomial evaluation, etc. but using it efficiently in an 8-level stack needs a more involved preparation to fill D: with the duplicating value.

The long way to do this is pressing **ENTER^^** seven times, a tedious 14-byte sequence not very elegant to say the least.

An alternative would be using three times ENTER^ (the native function) to fill the lower stack, plus four **bSTO** calls to fill the upper-stack half. This is also very inefficient and takes 11 bytes, not state of the art either.

That's why the companion DUAl_APPS Module has the function **XFILL^** to do a complete stack filling using the value in the X-register. A two-byte, one instruction solution to patch this gap at MCODE speed – perfect for the job, and a good reason to have the DUAL_APPS module also plugged in.

Trapping the Back-Arrow action

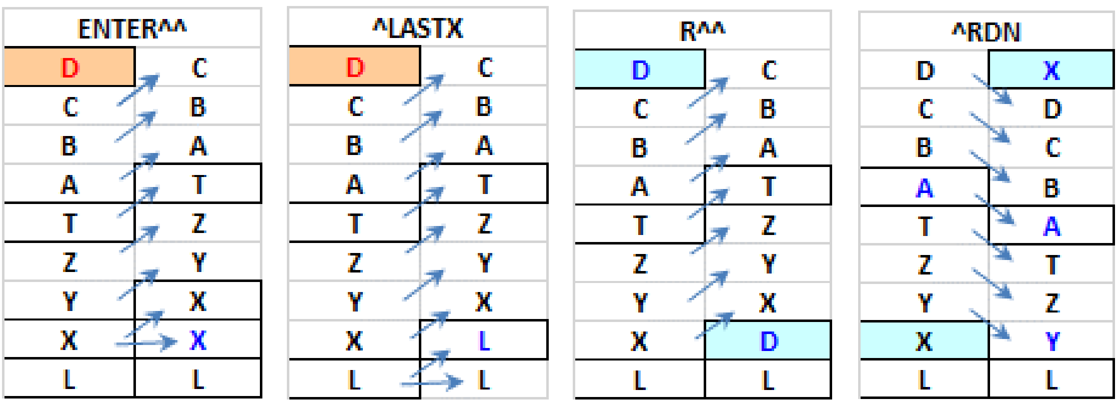
This boundary condition required additional consideration in the I/O_SVC coding. It's funny how such simple operations are taken for granted but have strong implications in the design of system enhancements like the double-length stack.

During the data entry process the back-arrow is used to do corrections, be that for a single digit or when pressed repeated times clearing all digits, to invoke the native CLX function - which in turn clears F11 to disable the automatic stack lift, and thus allowing that the following value replace the zero that was put in X.

Alright then, so it's clear that this contingency needs to be covered in the I/O_SVC control code, clearing UF 01 when the back-arrow-invoked CLX disables the automatic stack lift – or, in other words, when a back-arrow keypress causes a F11 clear condition.

With this boundary condition under our belt, all the needed use cases are solved. The summary table below details the different possible scenarios that need to be managed by the module.

Function	**Stack Lift**	**Stack Drop**	**Clears F11**	**Trapped?**
ENTER^	Yes when F11 set	No	Yes	Yes
CLX	No	No	Yes	Yes
Back-arrow	No	No	Yes if Last Digit	Yes
LASTX, RCL, PI	Yes when F11 set	No	No	Yes
Two-number Fnc.	No	Yes	No	Yes
RDN, R^	Roll Up/Down		No	Yes



[From: -> To:] diagrams for the "Fantastic Four"

Automated actions and Controls.

There are several actions performed behind the scenes every time a double-length stack function is used. Even if they're done in automatic fashion, the user needs to understand them to have a good grasp of the conditions for data input/output.

1. Every single function except **^CLX** and **^CLDST** saves the argument in X: into the L: stack level. This is equally done for one- and two-argument functions.

2. *Any function that alters the content of the T: register* (**^CLST ^RDN**, **R^^**, **ENTER^^** and all two-number functions (**^+**, **^-**, **^***, **^/** , **^MOD** and **^Y^X**) will, upon completion, make a copy of the value in the stack T: register into the LGTK register to ensure that it is up-to-date when/if needed.

3. *All two-number functions* (**^+** , **^-** , **^*** , **^/** , **^MOD** and **^Y^X**) will also perform a stack drop, duplicating the value in D: into the C: double-length stack level. This is done *before* T: is copied into LGTK, obviously.

4. Pressing **ENTER^^** or just typing numbers using the calculator numeric keys does an automated stack lift, losing the value that was in T: before. This is corrected by the IO_SVC control, restoring the value saved in LGTK back in stack register A: - where it should have been placed had it not been lost. You'll hear a short tone each time this correction action takes place, so you know your back is covered ;-)

5. **ENTER^^** and ^**CLX** will clear both CPU F11 and user flag UF 01 upon completion, signaling a stack lift disable condition for the subsequent operation.

6. All other functions need to leave UF 01 set upon completion. This also done by the IO_SVC control, which sets it when the LGTK doesn't need updating or when the pressed key isn't a numeric key (thus covering the whole range of scenarios).

7. **^LASTX**, **^PI** and **^RCL** will use the signal left by the pair above to **<u>replace</u>** X: with the recalled argument, *without pushing the stack first* and then writing it into the X: stack level. If F11 / UF 01 are set the operation will perform normally, that is making the stack lift and copying the recalled value into X:

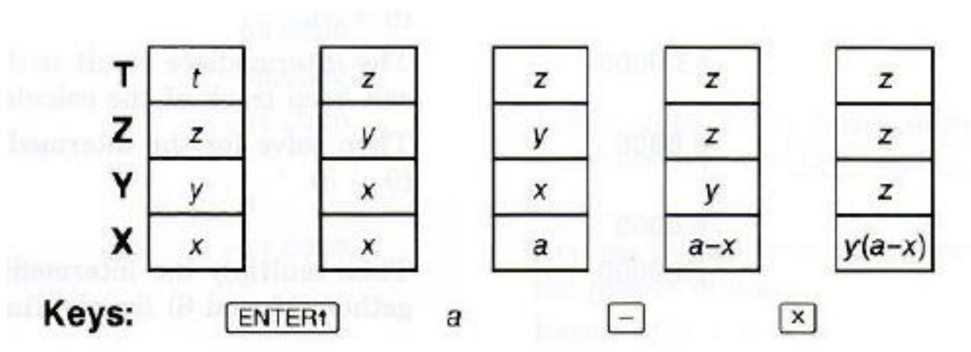
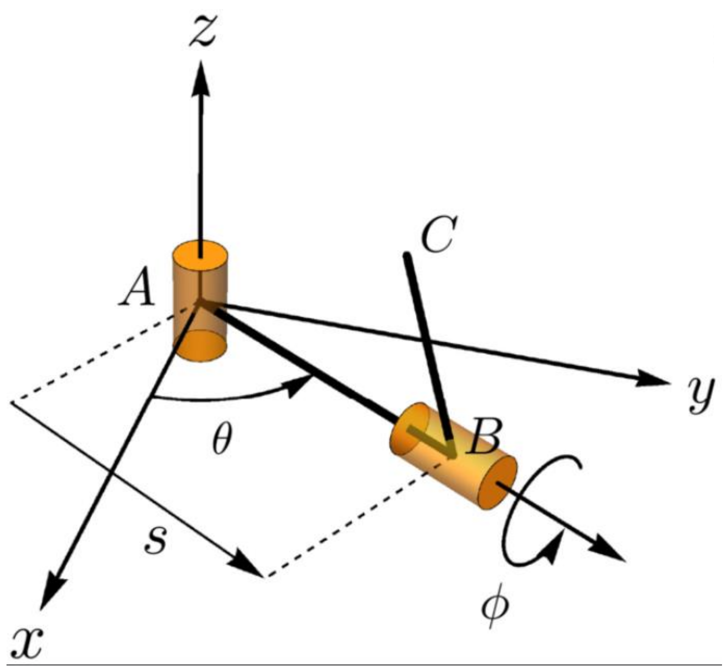


Figure 1 – When things were simple…



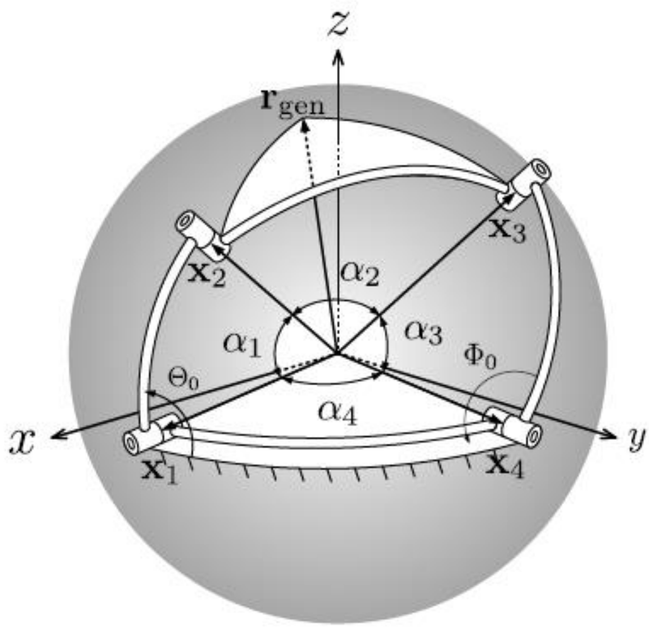
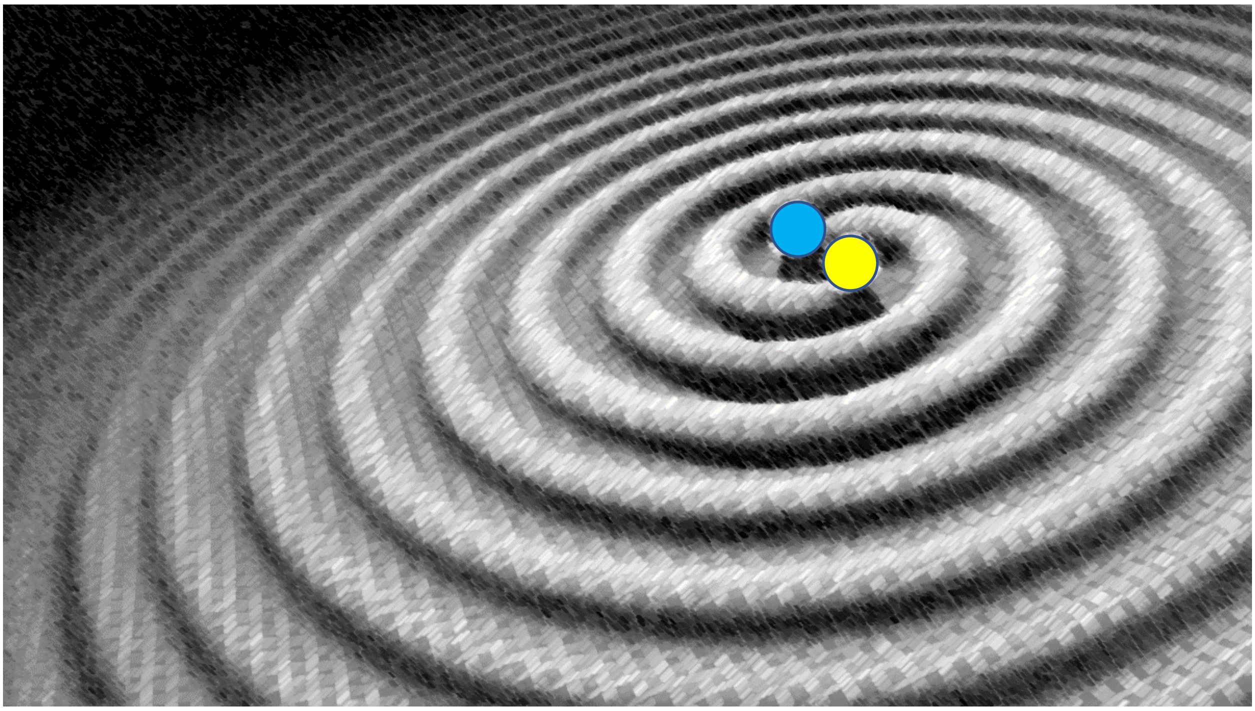


Figure 0.- 4-Bar mechanism

Part II – Dual Numbers



Part II - Dual Numbers.

In this section we'll describe the functions and capabilities provided by the DoubleDown module to operate with Dual Numbers and to manage the Dual-stack required for them. If you're familiar with the 41Z Module you'll recognize the concepts and this section will be a breeze; but if you're new to the game get ready for a fun ride with a not steep learning curve.

Managing the Dual Number Stack

There are two groups of functions available to support the Implementation of the Dual Number module:

3. Stack Management functions, and
4. Math functions for dual numbers.

The table below shows the functions grouped by this criterion:

Admin / Stack Management		Math Functions	
XROM #	Function Name	XROM #	Function Name
01,26	CLDST	01,31	DR+
01,27	CLDX	01,32	DR-
01,28	DAVIEW	01,33	DR*
01,29	DENTER^	01,34	DR/
01,30	DKEYS?	01,35	DR^N
01,36	DRCL _ _	01,38	DREXP
01,37	DRDN	01,39	DRINV
01,41	DRND	01,40	DRLN
01,44	DRUP	01,42	DRNEG
01,45	DSTO _ _	01,43	DRSQRT
01,46	DUNIT?	01,53	DY^DX
01,47	DX=0?	01,56	DR-P
01,48	DX=Σ?	01,57	DP-R
01,49	DX=DY?	01,58	DCOS
01,50	DX<> _ _	01,59	DACOS
01,51	DX<>DY	01,60	DSIN
01,54	DVIEW	01,61	DASIN
01,55	LASTD	01,62	DTAN
		01,63	DATAN

Dual numbers have a real and a dual part. The convention used in this module is that *the dual part is stored in the Y-register, and the real part is stored in the X-register*. Besides the functions on the table above, the double-length stack functions **^PI** and **ENTER^^** described in the previous chapter also belong to this section. **ENTER^^** plays a crucial role for dual number data entry, so you need to be familiar with it as well.

The data entry process is then: { dual part, **ENTER^^**, real part }

Bulk Key Assignments for Dual Number functions.

You can use the function **DKEYS?** to do a bulk assignment or removal for all double-stack functions available in the Module, each going to their "natural" key location to replace the native lesser counterparts. The function prompts Y/N? for the assignment or removal of the KA's, therefore the question mark in the function name.

, only "Y" or "N" are accepted here:

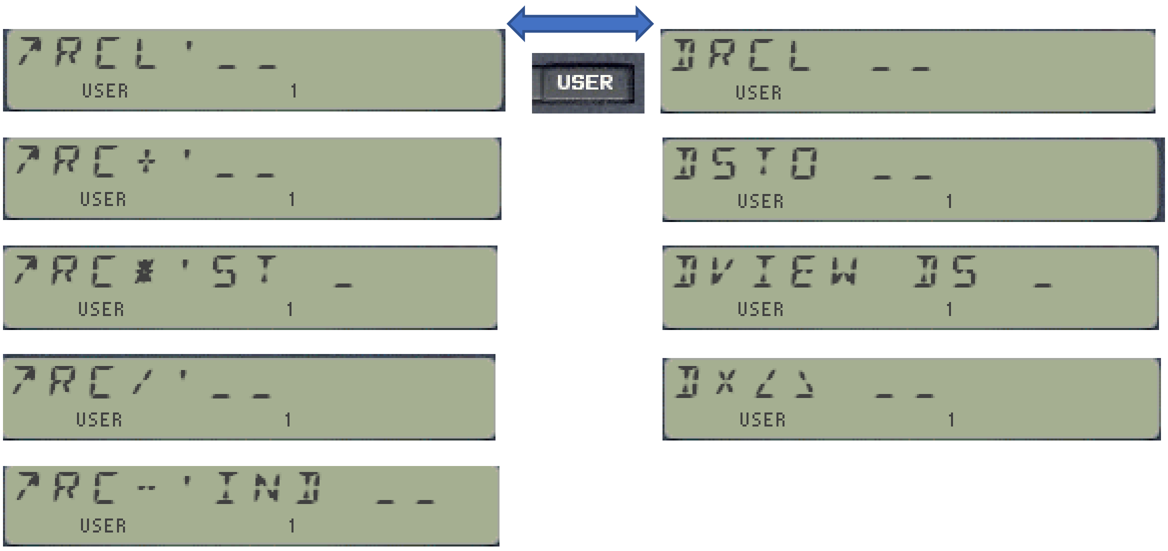
or:

The KA's removal action is followed by a memory PACKING to recover the KA's registers freed. Note that only the KA's on affected keys will be changed; any other KA on another key will not be modified so you can continue to use them.

Besides being a very powerful function by itself, **DRCL** has a double-duty role as a launcher for the "cluster" of associated functions, as follows:

	Key	Function		Key	Function	
	STO	**DSTO _ _**		STO	**bSTO _**	
	R/S	**DVIEW _ _**		R/S	**bVIEW _**	
	SST	**DR<> _ _**		SST	**bSWAP _**	
DRCL _ _	USER	**^RCL _ _**	→	USER	**^RCL _ _**	*full circle!*
	ENTER	**DENTER^**		ENTER	**ENTER^^**	
	PRGM	**^STVIEW**		PRGM	**^STVIEW**	
	XEQ	**DKEYS?**		XEQ	**^KEYS?**	

Notice how we can navigate across the two function clusters using the main anchor function as a passageway across them:



Test Functions

The module includes four test functions that operate on the dual-register value as a whole unit. They compare the specific condition and return a Boolean YES/NO in manual mode, plus the customary "skip next line if false" in a running program.

There are no order relationships in the dual number plane, so the tests are limited to equal comparisons, both between the stack levels DX and DY as well as checking for dual-zero and (1+e).

Notable case is **DUNIT?**, which here is simpler than in the complex world as the "modulus" is just the real part: |z| = x.

DUNIT?	Checks for x = +-1
DX=0?	Checks for x=y=0
DX=Σ?	Checks for x=0 and y=1
DX=DY?	Checks for X=Z, and Y=T

These functions are totally analogous to the native set of functions in the base machine for standard registers, do there's no point describing them at length. Instead, how about a glimpse of the actual MCODE under the hood?

1	DX=0?	*Header*	A3D6	*0BF*	**"?"**	
2	DX=0?	*Header*	A3D7	*030*	**"0"**	***Is DR = ε?***
3	DX=0?	*Header*	A3D8	*03D*	**"="**	*must have a=0 and b=1*
4	DX=0?	*Header*	A3D9	*018*	**"X"**	
5	DX=0?	*Header*	A3DA	*004*	**"D"**	*Ángel Martin*
6	DX=0?	**DX=0?**	**A3DB**	**0F8**	**READ 3(X)**	*a*
7	DX=0?		A3DC	2EE	?C#0 ALL	
8	DX=0?		A3DD	***067***	***JC +12d***	*[SKP]*
9	DX=0?		A3DE	0B8	READ 2(Y)	
10	DX=0?		A3DF	2EE	?C#0 ALL	
11	DX=0?		A3E0	***04F***	***JC +09***	*[SKP]*
12	DX=0?		A3E1	***083***	***JNC +16d***	*[NOSKP]*
1	DR=ε?	*Header*	A3E2	*0BF*	**"?"**	
2	DR=ε?	*Header*	A3E3	*04E*	**"Σ"**	***Is DR = ε?***
3	DR=ε?	*Header*	A3E4	*03D*	**"="**	*must have a=0 and b=1*
4	DR=ε?	*Header*	A3E5	*018*	**"X"**	
5	DR=ε?	*Header*	A3E6	*004*	**"D"**	*Ángel Martin*
6	DR=ε?	**DR=ε?**	**A3E7**	**0F8**	**READ 3(X)**	*a*
7	DR=ε?		A3E8	2EE	?C#0 ALL	
8	DR=ε?	`SKIP	A3E9	*0B9*	*?NC GO*	*False*
9	DR=ε?		A3EA	*05A*	*->162E*	*[SKP]*
10	DR=ε?		A3EB	00E	A=0 ALL	
11	DR=ε?		A3EC	35C	PT= 12	*Builds "1"in A*
12	DR=ε?		A3ED	162	A=A+1 @PT	
13	DR=ε?		A3EE	0B8	READ 2(Y)	
14	DR=ε?		A3EF	36E	?A#C ALL	a#1?
15	DR=ε?		A3F0	*3CF*	*JC -07*	
16	DR=ε?	NOSKIP	A3F1	*065*	*?NC GO*	*True*
17	DR=ε?		A3F2	*05A*	*->1619*	*[NOSKP]*

A brief intro to Dual numbers

See: https://en.wikipedia.org/wiki/Dual_number

In algebra, the dual numbers are a hypercomplex number system first introduced in the 19th century. They are expressions of the form a + bε, where a and b are real numbers, and ε is a nilpotent number taken to satisfy ε^2 = 0 ; but ε # 0.

Thus, the dual numbers are elements of the 2−dimensional real algebra

$$\mathbb{D} = \mathbb{R}[\varepsilon] = \left\{ z = x + y\varepsilon \mid (x, y) \in \mathbb{R}^2, \varepsilon^2 = 0 \text{ and } \varepsilon \neq 0 \right\},$$

Dual numbers were introduced in 1873 by William Clifford, and were used at the beginning of the twentieth century by the German mathematician Eduard Study, who used them to represent the dual angle which measures the relative position of two skew lines in space. Study defined a dual angle as θ + dε, where θ is the angle between the directions of two lines in three-dimensional space and d is a distance between them.

This nice concept has lots of applications in many fields of fundamental sciences; such, algebraic geometry, Riemannian geometry, quantum mechanics and astrophysics. Dual numbers find applications in mechanics, notably for kinematic synthesis. For example, the dual numbers make it possible to transform the input/output equations of a four-bar spherical linkage, which includes only rotoid joints, into a four-bar spatial mechanism (rotoid, rotoid, rotoid, cylindrical)

The dual numbers were originally introduced within the context of geometrical studies. They were later exploited to deal with problems in pure and applied mechanics. For instance, it has been demonstrated how to formulate the equations of rigid body motion in terms of just three "dual" equations instead of their six "real" counterparts (thereby realizing an equivalence between spherical and spatial kinematics). More recently, their importance has been recognized in numerical analysis to reduce round-off errors.

Note: The concept of a non-zero value that becomes zero when squared is a conflicting one at first sight – but not more so than the imaginary unit when it began to be used in complex number theory. I found the notion of -0 somehow helped me to accept the scheme, although there isn't such a thing as -0 of course, but it offers certain symmetry if we parallel it to: x^2 = (-x)^2

Another interpretation of ε (and probably more founded) assigns for it an infinitesimally small value that, even if not zero, it becomes zero when squared.

In the module the Greek character sigma "Σ" is used to represent "epsilon". This can be seen in some function names, as well as the standard presentation of the dual values in the display:

, or:


Note how for integer values the presentation omits the unneeded decimal digits for clarity.

This presentation is done automatically in manual mode by all the dual number functions. In program mode it is not shown (imagine the clutter?), thus the function **DAVIEW** can be used at the end of the program to produce the display.

Dual Number representations: The Unit "Circle".

Dual numbers can be represented as follows:

- Gaussian representation: z = x + yε.
- Polar representation: z = x (1 + ε arg z) , where arg z = y/x, x # 0, is the argument of z.

How this relates to the exponential form used with complex numbers can be seen if we consider that the "unit circle" of dual numbers consists of those with a = ±1, since these satisfy z * z~ = 1

However, note that the exponential map applied to the ε-axis covers only half the "circle":

$$e^{b\varepsilon} = \sum_{n=0}^{\infty} \frac{(b\varepsilon)^n}{n!} = 1 + b\varepsilon,$$

, using this expression in the definition or argument:

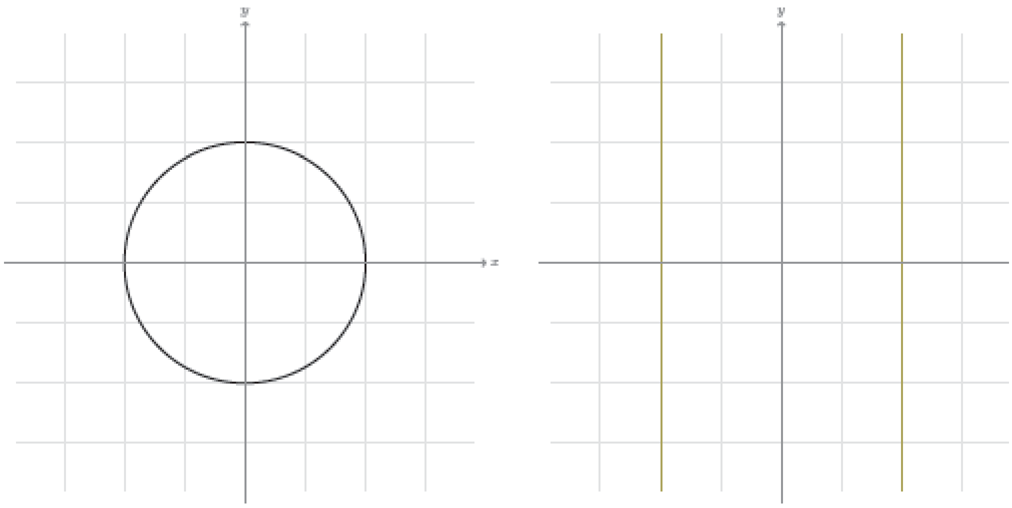
z = x (1 + ε arg z) = x . e^(ε. argz) ; identical form as the complex number z = |z|.e^(i. argz)

Therefore, the modulus (or norm) of a dual number is its real part, and its argument is the dual part over the real part, when the real part is not zero. For example:

<=>

The conversions between Rectangular and Polar are available with functions **DR-P** and **DP-R**

All throughout the module the dual numbers are represented in gaussian form.



(a) Unit circle $\hat{\mathbb{C}} \subset \mathbb{C}$ in complex number plane.

(b) Unit circle $\hat{\mathbb{D}} \subset \mathbb{D}$ in dual number plane.

The Dual RCL, STO, View and Exchange.

By the very definition dual numbers are formed by two real numbers, and therefore use two data registers. This makes memory handling functions like the native STO and RCL ill-prepared to handle them and thus we need to replace them with dual-stack aware counterparts.

The set of Dual doppelgangers have the same capabilities in terms of indirect and stack register addressing, as well as the dual register index. Note that each dual register takes two data registers thus their indexes really point at the double number:

- **DSTO** n saves the real part in X to R(2n) and the dual part in Y to R(2n+1)
- **DRCL** n recalls the registers R2n and R(2n+1) to X,Y respectively.
- **DX<>** n exchanges the said pair of registers, and
- **DVIEW** n shows the contents of R2n and R(2n+1) as a dual number in the display.

The stack addressing supports *the five levels of the dual stack*: DX, DY, DZ, DT, and DL

, valid entries: { X,Y,Z,T,L }

The INDirect stack addressing supports *all status registers* as targets:

, valid entries: { X.Y.Z.T.M.N.O.P.Q.K a.b.c.d.e}

Don't mistake them with the upper-stack registers { ABCDEF|G} – those are already part of { DX,DY,DZ,DT,DL } and thus not suitable for an indirect addressing!

Note that there's no support for in-place RCL or STO math operations, sorry but that was beyond the project scope at this time.

Stack Mechanics of DRCL and LASTD.

Here again we encounter the stack-lift topic in our path, a real trademark of the RPN stack design that needs to be looked at carefully in the context of dual number stack as well. Mimicking the standard native operation with (single) real numbers, both **DRCL** and **LASTD** should check whether the stack lift is enabled prior to pushing the recalled value into it.

Two problems arise that need to be addressed - one is solved but the other is not.

- The easier one is deciding whether to lift the dual stack. We know that this is signaled by the O/S using CPU Flag 11, thus we'll check if F11 is set - not a difficult thing to do just checking its status. The code will also clear User Flag 01 if F11 is set on entry, and UF 01 will remain clear during the execution of the function. Both **DRCL** and **LASTD** will use this simple approach, so all it's good here.

- The difficult one is making the disabled stack-lift condition *persist until both components of the dual number are entered*. For all purposes the O/S is going to clear F11 when the first part is introduced, and therefore we would need an additional marker to be used as semaphore in the subsequent action, entering the second part of the dual number. The potential solution would then use the proxy UF 01 as deciding factor.

Stack Mechanics of DENTER^ and CLDX

There is an important fact in the way these two functions work in the Dual stack implementation: contrary to their "native" counterparts, *the stack lift is only half-way disabled upon their termination*. The implications of this are that typing new digits after CLDX or DENTER^ works as expected, thus overwriting the X-register (i.e. the value is not pushed). However, introducing the second part of the dual number finds both F11 and UF 01 set, and the second value is pushed into the stack – mangling the dual-number stack into a straddled arrangement.

This applies to the following scenarios:

a. Use **CLDX** *to replace DX with a new Dual value*, not lifting the stack
b. Use **DENTER^** to push DX into DY and enter a new argument in DX for a dual number operation.

Unless taking corrective action, after CLDX or DENTER^ the *DX stack level will not be properly overwritten with newly entered digits*, rather the second part will be pushed up into the stack. This is an undesired situation that need to be avoided.

Register	*Initial*	*CLDX*	*digit entry*
F:	DL	DL	DX Real Part
e:			DT
d:	DT	DT	
c:			DZ
b:	DZ	DZ	
a:			DY
T:	DY	DY	
Z:			0
Y:	DX	0	
X:			*new digits*

or:

Register	*Initial*	*DENTER^*	*digit entry*
F:	DL	DT	DT Real Part
e:			DZ
d:	DT	DZ	
c:			DY
b:	DZ	DY	
a:			DX
T:	DY	DX	
Z:			DX
Y:	DX	DX	
X:			*new digits*

Although it's unfortunately not possible to prevent this issue from happening, there is an easy way to avoid the problem to begin with - not ideal but not insurmountable either, and arguably easy-peasy with a little discipline:

1. Press **ENTER^^** *instead of* the offending function **CLDX** or **DENTER^**
2. Enter the digits for the dual part , as you always do
3. Press X**<>Y** and CLX, to move it to the Y: register, and to disable UF 01 again
4. then enter the digits for real part, and you're done.

If that's so, then why having the **CLDX** and **DENTER^** functions at all? Just because there are genuine reasons to either clear the DX level (instead of typing { ENTER^^ , 0, ENTER^^, 0 } or to copy it into DY while you do other calculations with the original saved in DX

This is a byproduct of the dual stack design, which doesn't have any "scratch pad" reserved for auxiliary operations or number data entry. The buffer model used here is certainly simpler (and faster) than the model used in the 41Z Module, but the "fly in the ointment" is this inconsistent behavior – divergence from the native real stack.

Why not use the same 41Z Buffer design, I hear you asking? Well, as I mentioned it's faster & nimbler – but the main reason is because this one here serves a dual purpose, not only for the Dual-Real numbers but also for the Double Length stack, remember? Killing two birds with the same buffer #7 stone has this small drawback but it's worth the price of admission.

Automated actions and Controls.

There are several actions performed behind the scenes every time a dual number function is used. Even if they're done in automatic fashion, the user needs to understand them to have a good grasp of the conditions for data input/output. Note that points #4 and #6 were already explained in the double-length stack section, but they are repeated here for completion's sake as it's also appropriate and useful.

1. Every single function except **DRNEG**, **CLDX** and **CLDST** saves the original dual argument in DX into the DL dual stack level. This is equally done for one- and two-argument function.

2. *Any function that alters the content of the T: register* (**CLDST DRDN**, **DRUP**, **DENTER^** and all two-number functions (**DR+**, **DR**-, **DR***, **DR/** and **DY^DX**) will, upon completion, make a copy of the value in the stack T: register into the LGTK register to ensure that it is up-to-date when/if needed.

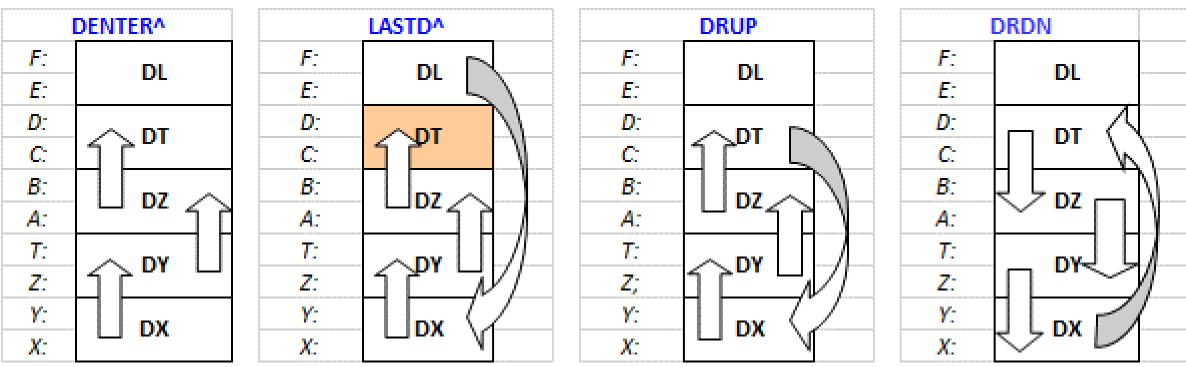
3. *All two-number functions* (**DR+**, **DR**-, **DR***, **DR/** and **DY^DX**) will also perform a stack drop, duplicating the value in DT into the DZ dual stack level. This is done *before* T: is copied into LGTK, obviously.

4. Pressing **ENTER^^** or just typing numbers using the calculator numeric keys does an automated stack lift, losing the value that was in T: before. This is corrected by restoring the value saved in LGTK back in stack register A: - where it should have been placed had it not been lost. You'll hear a short tone each time this correction action takes place, so you know your back is covered ;-)

5. **DENTER^** and **CLDX** will clear both CPU F11 and user flag UF 01 upon completion, signaling a stack lift disable condition for the subsequent operation. Note: if you want the following operation to lift the stack you need to re-enable it, and in a running program this requires to set UF 01 in an explicit program step.

6. All other functions need to leave UF 01 set upon completion. In manual mode this also done by the IO_SVC control, which sets it when the LGTK doesn't need updating or when the pressed key isn't a numeric key (thus covering the whole range of scenarios). Yet in a running program there may be required to set UF 01 in an explicit program step.

7. **LASTD** and **DRCL** will use the signal left by the pair above to **replace** DX with the recalled argument, *without pushing the stack first* and then writing it into the DX stack level. If F11 / UF 01 are set the operation will perform normally, that is making the stack lift and copying the recalled value into DX.



Dual Number algebraic functions.

The Dual Number algebraic rules, summarized below, are a straightforward consequence of the previous identity (with z = x + εy and w = u + εv):

Component-wise algebraic addition

$$z + w = x + u + \epsilon(y + v)$$

Product

$$z \cdot w = xu + \epsilon(xv + yu)$$

Inverse

$$z^{-1} = \frac{1}{x}\left(1 - \epsilon\frac{y}{x}\right) \quad (x \neq 0)$$

Power

$$z^n = x^n\left(1 + n\epsilon\frac{y}{x}\right) \quad (n \in \mathbb{Z}_{\geq 0},\ x \neq 0)$$

This multiplication is commutative, associative and distributes over addition.

The algebra of dual numbers D has the numbers εy, y ∊ R, as divisors of zero. No number εy has an inverse in the algebra D.

The Power to an integer function expects the dual number *stored in a hybrid way*, in the stack registers { Y,Z }, and *the exponent n in the X-register*. This is the natural logic for the date entry, for instance let's calculate (2+3ε)^4 :

3 , **ENTER^^** , 2 , **ENTER^^** , 4 , **DR^N** => 16+Σ96

You can verify it by squaring the argument twice:

LASTD , **DX^2** , **DX^2** => 16+Σ96

Division of dual numbers is defined when the real part of the denominator is non-zero. The division process is analogous to complex division in that the denominator is multiplied by its conjugate in order to cancel the non-real parts.

The conjugate z~ of the dual number z = x + εy is defined by z~ = x – εy, so: z * z~ = x^2
Thus, the division z1 / z2 is possible and unambiguous if x2 # 0.

$$\frac{z_1}{z_2} = \frac{z_1\overline{z_2}}{\overline{z_2}z_2} = \frac{x_1x_2 + (x_1y_2 - x_2y_1)\,\varepsilon}{x_2^2},$$

Because we're not familiar with double numbers we tend to expect similar results to those in complex numbers, but that's not always the case. Some of the expressions strike an unfamiliar chord, and sure enough the results are at times very counter intuitive. For example:

1/(1+Σ1) = 1+Σ1
(4+Σ2)^2 = 16+Σ16
(1+Σ3)^2 = 1+Σ6

and in general: (x + y ε)^2 = x^2 + 2 x.y ε

Other Holomorphic Functions of Dual Numbers

The following formulas have been used to program the functions in the module.
Note the differences with the complex number expressions across the board!

Powers:

$$(a+b\epsilon)^{c+d\epsilon} = a^c + ca^{c-1}b\epsilon + \ln(a)a^c d\epsilon.$$

which is going to require the logarithm, not a surprise here.

Example:

(1+ε)^(1-ε) = 1+ε

(1+ε)^(2+0ε) = (1+2 ε) ;

Transcendental functions:

$$\exp(z) = e^z = e^x + e^x y\varepsilon = e^x(1+y\varepsilon).$$

Very easy to deduce using the power series expression for exp(z) and considering that all terms with ε^n | n>=2 are null.

$$\log z = \log x + \frac{y}{x}\varepsilon = \log x + (\arg z)\,\varepsilon \quad \forall z \in \mathbb{R}_+^* \times \mathbb{R} \subset \mathbb{D}.$$

Also easy to come to using the polar representation of the dual number, of course.

Example: Ln(1+ε) = ε ; e^(1+ε) = e (1+ε)

Trigonometric functions:

$$\sin z = \sin x + (\cos x)\, y\varepsilon \quad \forall z \in \mathbb{D},$$

$$\cos z = \cos x - (\sin x)\, y\varepsilon \ \forall z \in \mathbb{D},$$

$$\tan z = \tan x - \frac{y}{\cos^2 x}\varepsilon = \frac{\sin z}{\cos z} \ \forall z \in \mathbb{D} - \{(2k+1)\,\pi,\ k \in \mathbb{Z}\} \times \mathbb{R}.$$

Note that the angular mode has no relevance on these.

Examples:

Sin (1+ε) = 0.841470985+Σ0.540302306

Cos (1+ε) = 0.540302306-Σ0.841470985

Tan (1+ε) = 1.557407725+Σ3.425518820

The Hyperbolic functions are also easy to figure out using their exponential forms:

$$\sinh z = \frac{e^z - e^{-z}}{2} \quad \forall z \in \mathbb{D},$$

$$\cosh z = \frac{e^z + e^{-z}}{2} \quad \forall z \in \mathbb{D},$$

$$\tanh z = \frac{e^z - e^{-z}}{e^z + e^{-z}} \quad \forall z \in \mathbb{D}.$$

$$\left\{\begin{array}{l} \sinh z = \sinh x + (\cosh x)\, y\varepsilon \quad \forall z \in \mathbb{D}, \\ \cosh z = \cosh x + (\sinh x)\, y\varepsilon \; \forall z \in \mathbb{D}, \\ \tan z = \dfrac{\sinh z}{\cosh x} \; \forall z \in \mathbb{D}. \end{array}\right.$$

These are not included in the module (no more FAT entries were left!) but a very simple FOCAL program can be used – and it serves as a good example of the utilization in a program of the other functions as well, see below.

Note that this program uses **ENTER^^** and **^/** (steps #07, 10, 19 and 22). This method preserves the integrity of the DY and DZ dual stack levels (only the dual part of the DT level is lost). Otherwise we would end up with a straddled situation, where the logical dual number occupies the wrong locations in the double-stack.

01 LBL "DSINH"
02 **DREXP**
03 **LASTD**
04 **DRNEG**
05 **DREXP**
06 **DR-**
07 **ENTER^^**
08 2
09 ST/ Z
10 **^/**
11 **DAVIEW**
12 RTN
13 LBL "DCOSH"
14 **DREXP**
15 **LASTD**
16 **DRNEG**
17 **DREXP**
18 **DR+**
19 **ENTER^^**
20 2
21 ST/ Z
22 **^/**
23 **DAVIEW**
24 RTN
25 LBL "DTANH"
26 DREXP
27 **DENTER^**
28 **LASTD**
29 **DRNEG**
30 **DREXP**
31 **DENTER^**
32 **DRUP**
33 **DR+**
34 **DRDN**
35 **DR-**
36 **DRUP**
37 **DR/**
38 **DAVIEW**
39 END

Examples:

Sinh (1+Ɛ) = 1.175201194 + Σ 3.086161269
Cosh (1+Ɛ) = 1.543080635 + Σ 2.350402387
Tanh (1+Ɛ) = 0.761594156 + Σ 0.419974342

Dual Number Inverse Trigonometric functions

I didn't find any reference in the available literature to the calculation of inverse trigonometric functions, so I had to come up with my own approach. The basis takes advantage of the automatic differentiation of analytical dual functions, whereby:

$$f(a+b\varepsilon)=\sum_{n=0}^{\infty}\frac{f^{(n)}(a)b^n\varepsilon^n}{n!}=f(a)+bf'(a)\varepsilon,$$

With that in mind it's a simple matter to obtain the inverse trigonometric functions from their derivatives, which thankfully don't need any direct trigonometric functions at all:

$$\frac{d}{dz}\arcsin(z)=\frac{1}{\sqrt{1-z^2}}\ ; \qquad z\neq -1,+1$$

$$\frac{d}{dz}\arccos(z)=-\frac{1}{\sqrt{1-z^2}}\ ; \qquad z\neq -1,+1$$

$$\frac{d}{dz}\arctan(z)=\frac{1}{1+z^2}\ ; \qquad z\neq -i,+i$$

This round-about approach may seem a little complicated but in fact the resulting code is very simple and short, so I'm more than happy with the end result.

asin (x+yε) = asin(x) + ε y / sqrt(1-x^2)
acos (x+yε) = acos(x) - ε y / sqrt(1-x^2)
atan (x+yε) = atan(x) + ε y / (1+x^2)

Examples:

asin (sin (1+ε)) = asin (0.841 + ε 0.540) = 1 + ε 1
acos (cos(1+ε)) = acos (0.540 – ε 0.841) = 1 + ε 1
atan (tan(1+ε)) = atan (1.557 + ε 3.426) = 1 + ε 1.000

and if you want to impress your friends press this mutually cancelling sequence of keys:

1 , ENTER^^, 1 , DSIN, DCOS , DTAN , DATAN , DACOS, DASIN => 1 + Σ 1

Interestingly the cumulative error in DATAN is cancelled back in the complete chain, so the final result is accurate to 10 decimal places using the internal O/S 13-digit routines.

Dual Number Inverse Hyperbolic functions

Now that we've developed a working system I's a simple matter to come up with the expressions for both the inverse hyperbolic functions.

The derivatives are very resemblant of the trigonometric case, with only a transposition of terms and/or signs:

$$
\begin{aligned}
\frac{d}{dx}\operatorname{arsinh} x &= \frac{1}{\sqrt{x^2+1}} & & \\
\frac{d}{dx}\operatorname{arcosh} x &= \frac{1}{\sqrt{x^2-1}} & & 1 < x \\
\frac{d}{dx}\operatorname{artanh} x &= \frac{1}{1-x^2} & & |x| < 1
\end{aligned}
$$

Hence:

asinh (x+yε) = asinh(x) + ε y / sqrt(1+x^2)
acosh (x+yε) = acosh(x) - ε y / sqrt(x^2 - 1)
atanh (x+yε) = atanh(x) + ε y / (1-x^2)

Where the main annoyance resides in the lack of real variable hyperbolic functions in the native function set, and therefore we need to include the MCODE for them as well.

Examples:

asinh (sinh (1+ε)) = asinh (0.175 + ε 1.543) = 1 + ε 1
acosh (cosh(1+ε)) = acosh (1.543 – ε 1.175) = 1 + ε 1
atanh (tanh(1+ε)) = atanh (0.762 + ε 0.420) = 1 + ε 1.000

The all-around test yields a slightly less accurate final result:

1 , ENTER^^, 1 , DSINH, DCOSH , DTANH , DATANH , DACOSH, DASINH =>
1.000000000 + Σ 1

missing out only in the tenth decimal digit, not bad at all even if not perfect.

Note that although these functions are included in the DBLDOWN ROM, the FAT was already full so their calling entry points are in the DUAL_APPS ROM .

Dual Number AGM and HGM.

As a direct application of addition, product and square roots we can proceed with the calculation of the Arithmetic-Geometric Mean AGM, and the Harmonic-Geometric Mean (GHM).

In mathematics, the arithmetic–geometric mean (AGM) of two positive real numbers x and y is defined as follows: First compute the arithmetic mean of x and y and call it a1. Next compute the geometric mean of x and y and call it g1; this is the square root of the product xy. Then iterate this operation with a1 taking the place of x and g1 taking the place of y. In this way, two sequences (an) and (gn) are defined:

$$a_1 = \frac{1}{2}(x+y) \qquad a_{n+1} = \frac{1}{2}(a_n + g_n)$$

$$g_1 = \sqrt{xy} \qquad g_{n+1} = \sqrt{a_n g_n}$$

These two sequences converge to the same number, which is the arithmetic–geometric mean of x and y; it is denoted by M(x, y), or sometimes by agm(x, y).

The Geometric-Harmonic Mean on the other hand can be obtained from the AGM using the relationship show below:

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

Programming these expressions is easy using our dual number function set. The only needed precaution is that we must set the number of decimal digits to 8 to avoid oscillations in the partial results that would delay or event prevent the convergence altogether. That's why we use a rounded comparison instead of a full-fledge one.

```
01  LBL "DAGM"
02  SF 00          ; flag case
03  GTO 00         ; merge
04  LBL "HGM"
05  SF 00          ; flag case
06  DRINV          ; invert a0
07  DX<>DY         ; swap arguments
08  DRINV          ; invert bo
09  LBL 00         ; common code
10  FIX 8          ; adjust precision
11  LBL 01
12  DENTER^        ;bn in DZ
13  DENTER^        ; bn in DT
14  DRUP^          ; bn in DX
15  DR*            ; an.bn
16  DRUP           ; bn in DX
17  LASTD          ; bn
18  DR+            ; an+bn
19  ENTER^^
20  2
21  ST/ Z
22  ^/             ; (an+bn)/2
23  DRND           ; rounded
24  DX<>DY         ; an.bn
25  DRSQRT         ; sqrt(an.bn)
26  DRND           ; rounded
27  DX=DY?         ; are equal?
28  GTO 02         ; yes, exit
29  GTO 01         ; no, next iteration
30  LBL 02
31  FS? 00         ; HGM case?
32  DRINV          ; yes, invert value
33  FIX 3          ; restore defaults
34  DAVIEW         ; show value
35  END            ; all done.
```


Examples:

AGM [(8+ε), (23+ε)] = 14.51619896 +Σ 1.074911790
HGM [(8+ε), (23+ε)] = 12.67549219 +Σ 1.1196937195

Corollary: Complete Elliptic Integral of 1st. kind via the AGM.

Here's another low-hanging fruit that is begging to be picked – so ready or not here we go ; even if this is likely not relevant in this domain.

The trigonometric and Legendre forms of the Complete Elliptic Integral are as follows:

$$K(k)=\int_0^{\frac{\pi}{2}}\frac{d\theta}{\sqrt{1-k^2\sin^2\theta}}=\int_0^1\frac{dt}{\sqrt{(1-t^2)\,(1-k^2t^2)}},$$

We can re-write the expression using the agm, as follows:

$$K(k)=\frac{\pi}{2\,\mathrm{agm}\left(1,\sqrt{1-k^2}\right)}.$$

, for k^2 <1

Here's the FOCAL program used for the calculation. Note that for the most part we don't care about the dual stack condition because DAGM is going to use it all up anyway.

```
01 LBL "DELK"
02 DX^2
03 DRNEG
04 E
05 +
06 DRSQRT
07 ENTER^^
08 0
09 ENTER^^
10 E
11 XEQ "DAGM"
12 2
13 ST* Z
14 *
15 DRINV
16 PI
17 ST* Z
18 *
19 DAVIEW
20 END
```

Example:

ELK (0.5+ε) = -2.622842115 +Σ 2.858860658

Dual Number Lambert function.

Now going for the stretch goal – suffice it to say I have no idea if this is a regular option with dual numbers, but I thought it'd be very interesting to explore the concept. Obviously, the singular form of the exponential function hugely facilitates things, so we take good advantage of it.

The Lambert function W(z) is defined such as: $W(z) \,.\, \exp[W(z)] = z$

Let $z = a+b\varepsilon$; and $W(z) = u + v\varepsilon$

Using the defining equation for W(z):

$$a+b\varepsilon = (u+v\varepsilon).\ \exp(u+v\varepsilon) = (u+v\varepsilon)\ e^{\wedge}u\ (1+v\varepsilon) = e^{\wedge}u\ [u + v(1+u)\varepsilon]$$

equating the real and dual parts on both sides of the expression:

$$a = u.e^{\wedge}u, \qquad \Rightarrow \qquad u = W(a)$$
$$b = e^{\wedge}u.v(1+u), \qquad \Rightarrow \qquad v = b.e^{\wedge}(-u) / (1+u)$$

regrouping the terms, we have the final expression below – certainly a beauty:

$$W(z) = u + v\varepsilon = W(a) + \epsilon \frac{be^{-W(a)}}{1+W(a)}$$

So there you have it, to my knowledge another "original" contribution to the field – or a flunk of biblical proportions ;-)

Since we have reduced the problem to the real number domain, we can program this expression using **WL0**, the Lambert function for real numbers included in the SandMath module. The short program below *does the job "in-place"*, i.e. only using the DX level and therefore preserving the other dual stack values:

```
01  LBL "DRW"     ; a in X:, b in Y:
02  WL0           ; W(a)
03  SIGN          ; 1 in X, W(a) in L
04  ST+ L         ; 1+W(a) in L
05  X<> L         ; 1+W(a)
06  ST/ Y         ; b/(1+W(a)) in Y:
07  X<> L         ; 1
08  ST- L         ; W(a)  in L:
09  X<> L         ; W(a)
10  CHS           ; -W(a)
11  E^X           ; e^-W(a)
12  ST* Y         ; b. e^-W(a)/[1+W(a)] in Y:
13  CLX
14  LASTX         ; -W(a)
15  CHS           ; W(a)
16  DAVIEW
17  END
```

Example: calculate the Lambert function for: $z = 1+\varepsilon$

1, ENTER^^ , XEQ "DRW" => 0.567143290+Σ0.361896257

where in this case the real part is W(1) = Ω ; the Omega constant.

Dual Gamma and Psi functions.

Now boldly going to no-man's territory and possibly breaking all conventional math's rules, let's push forward this route calculating the Gamma and Psi (Digamma) functions.

Using once again the automatic differentiation rule it's feasible to obtain an expression to calculate the Gamma function for dual numbers, assisted by the real number versions fg Gamma and Psi available in the SandMath.

$$\Gamma(x + y\varepsilon) = \Gamma(x) + y\, \Gamma'(x)\, \varepsilon \ ;$$

and $\Gamma'(x)$ is derived from the relationship linking it with Psi and Gamma itself:

$$\Psi(x) = \Gamma'(x) / \Gamma(x) \ ; \ \text{hence:} \ \ \Gamma'(x) = \Gamma(x) \,.\, \Psi(x)$$

Substituting in the initial expression we obtain the resulting formula:

$$\Gamma(x + y\varepsilon) = \Gamma(x) + y\, \Gamma'(x)\, \varepsilon = \Gamma(x) + \varepsilon\, y\, \Gamma(x) \,.\, \Psi(x)$$

The mini-routine below calculates the Gamma value for a dual number zin DX. The *calculation is done in-place,* so $\Psi(z)$ replaces z in DX and the dual stack levels DY, DZ and DT remain unaltered.

```
01 LBL "DGAMMA"
02 PSI          ; Ψ(x)
03 ST* Y        ; y.Ψ(x) in Y:
04 CLX          ; disables stack lift
05 LASTX        ; x
06 GAMMA        ; Γ(x)
07 ST* Y        ; y.Γ(x).Ψ(x) in Y:
08 DAVIEW       ; show the world
09 END          ; done!
```

Example: $\Gamma(1+\varepsilon)$ = 1-Σ0.577215665

For large values of the argument we run into range limitations of the machine, so it's always good to have LogGamma (natural logarithm of Gamma) in the function set. Besides, this one is a double-win (pun intended) of the method, almost too easy to be true but that's how the cookie crumbles in this case.

We'll apply the automatic differentiation rule to the definition of the LNGAM function:

$$f(z) = \mathrm{Ln}(\Gamma(z)) = \mathrm{Ln}(\Gamma(x) + \varepsilon\, y\, d(\mathrm{Ln}\Gamma(x))/dx$$

$$f'(z) = \Gamma'(z) / \Gamma(z) = \Psi(z) \ ; \ \text{thus we have:}$$

$$\mathrm{Ln}(\Gamma(z)) = \mathrm{Ln}(\Gamma(x) + \varepsilon\, y\ \Psi(x)$$

This can be easily programmed in the super short routine below, which returns the value in the DX stack level. Note that the routine does the job "in-place" (preserves DT, DZ and DY) and it uses no data register.

```
01 LBL "DLNGM"
02 PSI
03 ST* Y
04 CLX          ; disable stack lift
05 LASTX        ; x
06 LNGM         ; Ln(G(x))
07 DAVIEW
08 END
```

Example: LnΓ $(1+\varepsilon)$ = -2.7140E-09 -Σ0.577215665

Thus: XEQ "DREXP" => 1 -Σ0.577215665

Finally, what about the dual number Psi function?

The approximation formula used for Psi is as follows:

$$\Psi(x) = \log(x) - \frac{1}{2x} - \frac{1}{12x^2} + \frac{1}{120x^4} - \frac{1}{252x^6} + O\left(\frac{1}{x^8}\right)$$

, or using w = 1/x:

$$Psi(z) = -\left(Ln\, w + \frac{w}{2} + \frac{w^2}{12}\left(1 - \frac{w^2}{10}\left(1 - \frac{10}{21}w^2\right)\right)\right)$$

This is a regular calculus using our dual number functions, with the only condition imposed by the inversion rule being that the real part of the argument cannot be zero. Thus, we can consider Psi available in the dual number domain.

This formula is accurate to at least 9 decimal figures for arguments with real part x > 9. For smaller arguments we'll use the following recurrence relationship:

$$\Psi(x) = \Psi(x+9) - \Sigma\ [1/(x-k-1) \quad ; k= 1,2...9$$

We'll apply the approximation formula to calculate Ψ(x+9), and then subtract the correction.

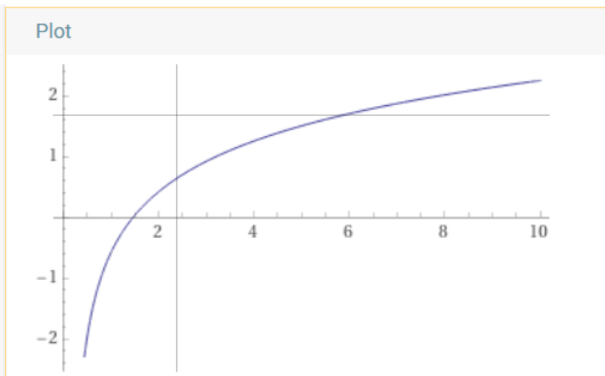
You can find the program that calculates its value in next page. Note the use of some *double-length stack functions* along with the *dual number functions*, in a wonderful demonstration of the common synergies between both subjects of this module.

Example: Calculate $\Psi(1 + \varepsilon)$

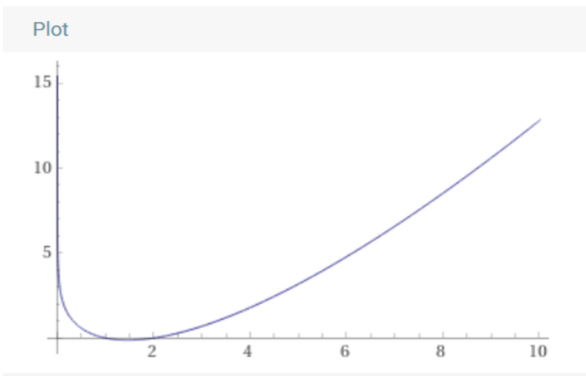
1 , ENTER^^, 1 , XEQ "DPSI" => -0.577215665 -Σ 1.644934067

Note that the program returns real numbers when the argument has a null dual part, which is rather logical if you ask me. For instance:

Ψ (1+0ε) = -0.577215665 +Σ 0 , the opposite of Euler's γ constant
Ψ (2+0ε) = 0.422784336 +Σ 0



Ψ(x)



Ln(Γ(x))

Program listing:

```
01 LBL "DPSI"     ; z in DX
02 STO M          ; save for later
03 ENTER^^        ; push for input
04 9              ; scale factor
05 STO O          ; save for later
06 ^+             ; 9+x
07 X<>Y
08 STO N          ; save for later
09 X<>Y
10 DRINV          ; w
11 DENTER^        ; w in DX and DY
12 DX^2           ; w^2
13 DENTER^        ; w^2 in DX and DY
14 ENTER^^        ; stack lift
15 10             ; 10 in X:
16 ENTER^^        ; stack lift
17 21             ; 21 in Y:
18 ^/             ; 10/21
19 ST* Z
20 ^*             ; 10.w^2 / 21
21 DRNEG
22 ENTER^^
23 E
24 ^+             ; 1 - 10.w^2 / 21
25 DRCL Y         ; w^2
26 DR*            ; w^2*(..)
27 ENTER^^
28 10
29 ST/ Z
30 ^/             ; w^2 * (..) / 12
31 DRNEG
32 ENTER^^
33 E
34 ^+             ; 1-w^2*(..)/10
35 DR*            ; w^2(1+w^2(..)/12)
36 ENTER^^
37 12
38 ST/ Z
39 ^/             ; (...) partial result
40 DX<>DY         ; w
41 DRLN           ; Ln(w)
42 LASTD          ; w
43 ENTER^^
44 2
45 ST/ Z
46 ^/             ; w/2
47 DR+            ; w/2 + Ln(w)
48 DR+            ; Ln(w) + w/2 + (...)
49 DRNEG          ; unscaled result
50 LBL 00         ; correction steps
51 RCL M          ; original x
52 RCL O          ; index k
53 DSE X          ; k-1
54 NOP
55 +              ; x+k-1
56 RCL N          ; original y
57 X<>Y           ; dual number in DX
58 DRINV          ; 1/[(x+k-1)+yε]
59 DR-            ; subtract from result
60 DSE O          ; next index
61 GTO 00         ; loops 9 times
62 DAVIEW         ; show it
63 END
```

$$\psi(z) = -\gamma + \int_0^1 \frac{1 - x^{z-1}}{1 - x}\, dx$$

Fig.1: Integral representation of Psi

Dual Bessel functions of first kind.

We manage this one as another direct application of the automatic differentiation rule whereby:

$$J(n, z) = J(n, x) + y. J'(n, x)\, \varepsilon$$

Using the derivative formula below:

$$2J_{\nu}'(z) = J_{\nu-1}(z) - J_{\nu+1}(z)$$

We can substitute the term J'(n,x) with the equivalent given by the formula, resulting:

$$J(n, z) = J(n, x) + \varepsilon\ y.[\ J(n\text{-}1, x) - J(n+1, x)] / 2$$

Again, this is another straightforward application of the SandMath JBS function. The program below expects n in register Z: and z = (x+yε) in stack registers {X,Y} - as you'd get them by typing:

"n" , ENTER^^, "y" , ENTER^^, "x" , ENTER^^

Note that except for JBS and DAVIEW, only standard functions are used, thus this program operates strictly within the "lower stack" XYZY and leaves the upper part alone. That's always an option, with the advantage of keeping DZ and DT untouched but obviously it destroys the data in the DY level. Also, there are several stack functions needed due to the format of JBS output, leaving the result in X and half the order (n/2) in Y.

```
01  LBL "DJBS"     ; n in Z:, x,y in {XY}
02  STO 00
03  RCL Z          ; n
04  STO 01
05  E
06  +              ; n+1
07  X<>Y           ; x in X:
08  JBS            ; J(n+1, x)
09  X<>Y
10  RDN            ; x in X
11  RCL 01         ; n
12  E
13  -              ; n-1
14  RCL 00         ; x
15  JBS            ; J(n-1, x)
16  R^
17  R^
18  RCL Z
19  -              ; J(n+1) – J(n-1)
20  2
21  /              ;
22  *              ; dual part
23  RCL 00
24  RCL 01
25  JBS            ; real part
26  X<>Y
27  RDN
28  DAVIEW         ; show result
29  END            ; end
```

With just a few modifications the same program can be used to calculate the derivative of the modified Bessel function of first kind, just replacing the JBS lines with IBS, and using "+" in line 19 instead.

Example: Calculate J[1, (1+ε)]

1, ENTER^^, ENTER^^, ENTER^^, XEQ "DJBS" => 0.440050586 -Σ 0.325147101

Dual-Step Root Finding for Real Functions

For the skeptical amongst you (oh faithless!), here's the clear proof that the dual number field has practical applications.

The concept of 'Dual-Step" is borrowed from complex analysis, where the Complex-Step derivative is a well-known method to calculate the derivative of a real function; just by evaluating the equivalent complex function instead, displaced an incremental amount and taking the imaginary part (see the 41Z Deluxe manual for details).

$$F'(x_0) \approx Im(F(x_0 + ih))/h$$

, with "h" sufficiently small.

We can use the analogous scheme with dual numbers, where we have the advantage of a simultaneous calculation of the real function and its derivative already built in the very result of each equivalent dual-function evaluation (a.k.a. the *automatic differentiation rule*). Moreover, this leaves things neatly prepared for a direct root-finding application using Newton's method, where the iterative correction factor is already known:

$$x_1 = x_0 - \frac{f(x_0)}{f'(x_0)}$$

The trick is using the value "1' for the dual part and thus calculating:

DF(x+ε) = f(x) + ε f'(x)

Say, what's that for direct applicability of a result with no need for additional steps?

The routine below exploits this idea and can be used to replace your trusty SOLVE or FROOT. All you need is a global program in memory for the Dual-Number function equivalent to the real function whose roots you want to calculate.

```
01  LBL "DFX=0"
02  "GUESS=?"
03  PROMPT          ; input guess
04  STO 00          ; xi in R00
05  "FNAME?"
06  AON
07  PROMPT          ; input name
08  AOFF
09  ASTO X          ; temporary
10  STO  M          ; saves one data reg.
11  FIX 9           ; for rounding
12  LBL C           ; subroutine entry
13  E               ; dual part
14  RCL 00          ; real part
15  XEQ IND M       ; evaluate fnc.
16  X<>Y            ; f'(x)
17  /               ; f(x)/f'(x)
18  ST- 00          ; x(i+1) in R00
19  FS? 10
20  VIEW 00         ; show current
21  RND             ; round value
22  X#0?            ; equal?
23  GTO C           ; no, do next
24  RCL 00          ; yes, recall result
25  FIX 3           ; reset defaults
26  CLD             ; clear LCD
27  RTN             ; all done.
28  GTO C
```

The routine uses ALPHA (register M), data register R00 and stack registers.

Let's see a couple of examples to get familiar with the approach.

1. Calculate the real root of f(x) = e^x – 5
2. Solve the Kepler equation M = x - E sin(x) ; for E=M=0.5

To tackle the first example, we write a small routine to program the equivalent DUAL function, i.e.

DF(z) = exp(z) - 5

For the second example the DUAL function is Kepler's equation with explicit parameters, i.e.

DF(z) = z – 0.5 sin(z) – 0.5 ; or easier: DF2(z) = 2z – sin(z) - 1

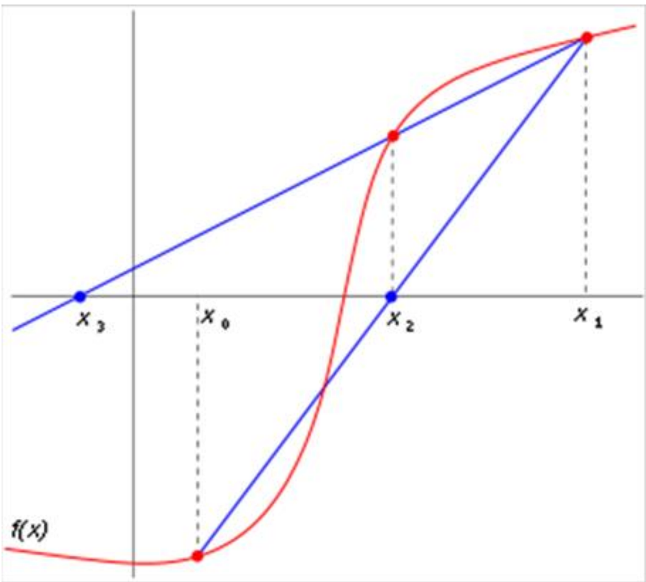
This is how they've been programmed:

```
01 LBL "DF1"
02 DREXP
03 ENTER^
04 0
05 ENTER^^
06 5
07 DR-
08 RTN
```

```
09 LBL "DF2"
10 DSIN
11 LASTD
12 2
13 ST* Z
14 *
15 DR+
16 1
17 -
18 END
```

Running our root-finding program couldn't be any easier, say we use an initial guess of x=1 for both:

XEQ "DFX=0"		GUESS=?
1, R/S		FNAME?
DF1, R/S	=>	1.609437912
XEQ "DFX=0"		GUESS=?
1, R/S		FNAME?
DF2 , R/S	=>	0.335418032



Dual Error Function and Exponential Integrals.

A final relapse to the special functions analysis that – again - takes advantage of the automatic differentiation in tight collaboration with the SandMath for the real variable functions. It;s just too convenient not to keep coming to it over and over again!

Error function: definition and derivative.

$$\operatorname{erf} z = \frac{2}{\sqrt{\pi}} \int_0^z e^{-t^2}\, dt.$$ and: $$\frac{d}{dz} \operatorname{erf} z = \frac{2}{\sqrt{\pi}} e^{-z^2}.$$

Exponential Integral: Definition and derivative.

$$\mathrm{Ei}(x) = -\gamma - \ln x + \int_0^x \frac{1-e^{-t}}{t}\, \mathrm{d}t$$ and: $$\frac{\mathrm{d}}{\mathrm{d}x}(\mathrm{Ei}(x)) = -\frac{e^{-x}}{x}$$

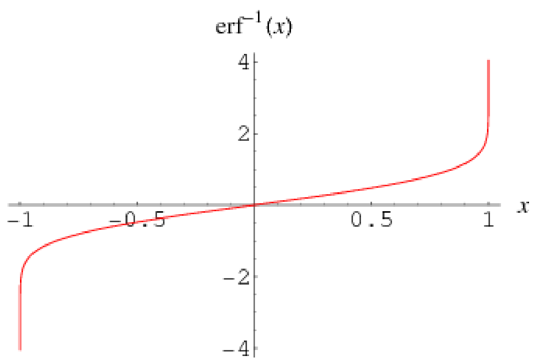
Sine and Cosine Integrals.

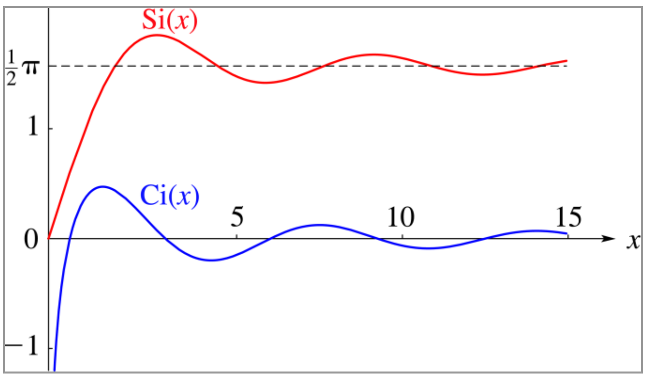
$$\mathrm{Shi}(x) = \int_0^x \frac{\sinh t}{t}\, dt$$, and: $$\mathrm{Chi}(x) = \gamma + \ln x + \int_0^x \frac{\cosh t - 1}{t}\, dt$$

The derivative is a trivial exercise from the definition.

The calculation method is always the same:

- Save the DX argument in the DL dual stack level, where it remains untouched,
- Start with the real variable circulation using the SandMath functions to get the real part,
- Recall the dual argument and perform the derivative calculations to obtain the dual part.
- Show the dual result.





See the program in next page that applies the described technique.

Program listing:

```
01 LBL "DERF"
02 DSTO L        ; save argument
03 ERF           ; Error function
04 DX<>DY        ; move result to DY
05 CLDX          ; get rid of scratch
06 LASTD         ; DX, stack not lifted
07 X^2
08 CHS           ; -x^2
09 E^X           ; e^(-x^2)
10 ST+ X         ; 2.e^(-x^2)
11 ^PI
12 SQRT          ; sqrt(π)
13 ^/
14 ^*            ; mult by dual "y"
15 X<>Y          ; put in place
16 DAVIEW        ; show result
17 RTN           ; done.
18 LBL "DEI"
19 DSTO L        ; save argument
20 EI            ; exponential Int.
21 DX<>DY        ; move result to DY
22 CLDX          ; get rid of scratch
23 LASTD         ; DX, stack not lifted
24 E^X           ; e^x
25 LASTX^        ; x
26 ^*            ; x.e^x
27 ^/            ; y / x.e^x
28 X<>Y          ; put in place
29 DAVIEW        ; show result
30 RTN           ; done.
31 LBL "DSI"
32 RAD
33 DSTO L
34 SI            ; Sine Integral
35 DX<>DY        ; move result to DY
36 CLDX          ; get rid of scratch
37 LASTD         ; DX, stack not lifted
38 SIN           ; sin(x)
39 GTO 00
40 LBL "DCI"
41 RAD
42 DSTO L        ; save argument
43 CI            ; Cosine Integral
44 DX<>DY        ; move result to DY
45 CLDX          ; get rid of scratch
46 LASTD         ; DX, stack not lifted
47 COS           ; cos(x)
48 LBL 00
49 LASTX^        ; x
50 ^/            ; cos(x) / x
51 ^*            ; y.cos(x) / x
52 X<>Y          ; put in place
53 DAVIEW        ; show result
54 END           ; DONE.
```

Remarks:

The SandMath functions use the "lower stack", thus the dual value in DY will be trashed.

However, we're using a combination of dual number and double-length stack functions in the calculation of the dual parts to preserve the dual values in DZ and DT. This can be seen in the use of DX<>DY , CLDX right after the SandMath function, which will make the LASTD argument to overwrite the DX level , not lifting the dual stack.

Examples:

DERF (0.5 + 0.5 ε) = 0.520499878+Σ0.439391289

DEI (1 + ε) = 1.895117817+Σ0.367879441

DSI (1 + ε) = 0.946083070+Σ0.841470985

DCI (1 + ε) = 0.337403923+Σ0.540302306

Poly-Dual-Nomials, - come again?

Reeling it back a tad, let's end this chapter with a few routines covering basic aspects of Polynomials in the Dual number plane, should we?

There's noting strange in the *poly-dual-nomial* concept (yes, the new name is officially coined, and it will stick!) thus we'll assume both the coefficients and the variable are dual numbers. We'll write Data Input and Evaluation routines and will try to get to the Root finding subject using the automatic differentiation. By the way, the Dual plane is not as forgiving as the Complex plane in that the formulas used must watch for the same data error situation as the real numbers, such as square roots of negative numbers, and set the necessary error trapping to avoid the crash.

We'll use the naming convention where the n-th. Index is for the coefficient of the x^n term;

$$P(x) = \sum_{k=0}^{n} a_k x^k$$

Data Input and Evaluation routines.

The program below can be used to enter the coefficients and to evaluate the *polydualnomial* at a given data point of the variable. If its degree is "N" we'll store the N+1 coefficients always in dual data registers starting with R01, that is { R01 to RN+1 }, using the control word "1,00(n+1)" to define it (in bbb.eee format).

```
01  LBL "DINPT"
02  "N=?"            ; Pol. degree
03  PROMPT
04  E3/E+            ; counter format
05  STO 00           ; used as scratch
06  LBL 00           ; main loop
07  "DN"             ; coeff. value
08  RCL 00
09  E
10  -
11  AINT             ; adds index
12  "/-=?"           ; as question
13  PROMPT
14  DSTO IND L       ; saved in Dual Reg
15  ISG 00           ; next index
16  GTO 00           ; loop till done
17  LASTX            ; counter
18  FRC
19  RTN              ; cntl. word in X
20  LBL "DPVL"       ; Cntl. word in X
21  "DX=?"           ; evaluation point
22  PROMPT           ; input DX
23  LBL A            ; subroutine entry
24  ^RCL Z
25  STO 01           ; keep a backup
26  FRC
27  ENTER^^
28  E3
29  ^*               ; degree
30  STO 00           ; counter to R00
31  ^RDN             ; get it off the way
32  DENTER^          ; push DX to DY
33  CLDX             ; initial value
34  SF 01            ; enable stack lift
35  LBL 01           ; term loop
36  DRCL Y           ; get current value
37  DRCL IND 00      ; coefficient
38  DR*              ; product
39  DR+              ; updated value
40  DSE 00           ; decrease counter
41  GTO 01           ; loop for next
42  DAVIEW           ; show the world
43  ^RCL 01          ; control word
44  RTN              ; ready
45  GTO A            ; run again
46  END
```

Note: DAVIEW is a resource-hungry function, it uses all scratch area, L and ALPHA to do the job. This forces us to use R00 for a backup of the control word.

Step #32 deserves some comments as well. We know that the I/O_SVC does the housekeeping for the LGKT:, ensuring there's no data loss across the double-stack divide (i.e. stack <-> buffer) and keeping UF 01 in sync with CPU F11. However, this only happens in manual mode and not under a running program, which leaves us with a manual refresh of UF 01 so stack lift is enabled again for the instruction DENTER^ right after LBL 01.

Note that we've used Honer's (or Ruffini's) method to write the polynomial taking common factor from right to left – This is the most efficient way to evaluate it as it only does multiplications, avoiding altogether all power operations, more time consuming and less accurate.

Example. Enter the coefficients and evaluate at z=(5+5ε) the polynomial:

P(z) = (-1-1ε) + (1+1ε) z + (2+2ε) z^2 + (3+3ε) z^3

Or, rewritten using Honer's method:

P(z) = (-1-1ε) + z { (1+1ε) + z [(2+2ε) + z (3+3ε)] }

1.004 , XEQ "DPVL"	=>	DX=?
5 , ENTER^^, 5 , R/S	=>	DN0=?
-1, ENTER^^, -1 , R/S	=>	DN 1=?
1 , ENTER^^ , 1 , R/S	=>	DN2=?
2 , ENTER^^ , 2 , R/S	=>	DN3=?
3 , ENTER^^ , 3 , R/S	=>	25+Σ50

Polynomial Derivative Evaluation

A routine to evaluate the derivative at a given point is given below. Notice that this is a stand-alone version but there are common code sections with the *polydualnomial* evaluation, so the proper approach is consolidating both into a single program to leverage from the code reuse.

01	**LBL "dDPVL"**	; Cntl. word in X
02	*"DX=?"*	; eval. point
03	PROMPT	; input DX
04	LBL B	; subroutine entry
05	**^RCL Z**	
06	STO 01	; keep backup
07	FRC	
08	**ENTER^^**	
09	E3	
10	**^***	; degree
11	STO 00	; counter to M
12	**^RDN**	; get it off the way
13	**DENTER^**	; push DX to DY
14	**CLDX**	; initial value
15	**SF 01**	*; enable stack lift*
16	LBL 02	
17	**DRCL Y**	
18	**DRCL IND 00**	; Ck coeff.
19	**DR***	
20	**^RCL 01**	; k+1
21	DSE X	; k
22	NOP	
23	ST* Z	; k*Ck
24	**^***	
25	**DR+**	; add to current
26	DSE 00	; decrease counter
27	GTO 02	; loop for next
28	**DAVIEW**	
29	**^RCL 01**	; control word
30	RTN	
31	GTO B	; for new data point
32	END	

Example: evaluate the derivative of the same *polydualnomial* at the same data point we did its evaluation before.

P'(z) = (1+1ε) + z [2*(2+2ε) + z 3*(3+3ε)]

Since the coefficients are already in memory, we can skip the data entry section and jump directly to the subroutine entry point LBL B – but not without typing the input parameters of course:

1.004 , ENTER^^ , 5 , ENTER^^ , 5 , XEQ B => 70+Σ 140

Regrouping for a moment, we're now equipped to give the root finding solution a good go – since we can calculate both P(z) and P'(z) with the routines above. All that's left is having a sensible driver program asking for the initial guess and orchestrating the iterations till convergence is (hopefully) reached.

Polydualnomial Twice-Roots.

By virtue of the automatic differentiation, finding roots of a *polydualnomial* is equivalent to finding the roots of the real polynomial that are also roots of its derivative polynomial:

Let z= x+yε, with y#0 then:

$$P(z) = P(x) + \varepsilon y\, P'(x)$$
$$\text{If } P(z) = 0 \quad \Rightarrow \quad P(x) = 0 \ \text{ and } \ P'(x) = 0$$

So, it imposes a double condition that makes the search twice as interesting, if not complicated. You'd allow me to coin the term "twice-roots" for them, although I agree all this naming is getting funky.

We know from calculus and polynomial algebra that the roots of the polynomial and its derivative are related by the Rolle's theorem, whereby if d1 is a real root of the derivative then it is placed in-between two real roots of the polynomial, r1 and r2. Let m1 and m2 the multiplicity of said roots, then we have:

d1 = (m1. r1 + m2. r2) / (m1 + m2)

The elephant in the room is that obviously we're going to have non-real roots in many cases, and we don't have any way to handle them. This is not different from the real field, where some roots are complex and therefore escape the Real domain into the Complex plane, right? Likewise, here we'll have some of the polydualnomial roots as Dual Complex numbers, instead of Dual Real. Yes, the thick plottens, as they say…

Dual Complex root: z + i w = (a+bε) + i (c+dε)

Just to touch lightly on the subject we'll attempt Newton's method to find double real roots, combining it with the automatic differentiation and hoping to get assistance from the SandMath prowess on real-variable calculations.

Starting from an initial guess x0, the successive iterations are given by the expression:

$$x_1 = x_0 - \frac{f(x_0)}{f'(x_0)}$$

Using the automatic differentiation, we can re-write it as:

$$z(k+1) = zk - [\ P(zk) / [\ P(xk) + \varepsilon\ yk\ P'(xk)\ ]$$

where: $zk = xk + yk\ \varepsilon$

The numerator P{zk) can be evaluated using the routine **DPVL**, which returns a dual number.

If *restricted it to real coefficients* (not dual numbers) both P(xk) and P'(xk) in the denominator can be obtained using the functions **PVL** and **dPVL** available in the SandMath, so no need to re-write then again. They expect the evaluation point and control word with the information of the location of the coefficients.

However, if we *also allow dual-number coefficients* then the polynomial will have both real and dual parts, even for real variable x. We need to write another routine to evaluate the *polydualnomial* derivative, not a big deal anyway so let's go this route for a general-purpose approach.

The routine below is such a driver program. It assumes the polydualnomial coefficients are saved in data registers R01 to RN+1, and expects the control word in X:

01	**LBL "DP=0"**	; cntl word in X:
02	"bbb.eee=?"	
03	PROMPT	
04	"GUESS=?"	
05	PROMPT	
06	**DSTO 10**	
07	LBL 10	; iterations loop
08	XEQ "B"	; derivative
09	**^RDN**	; P' in DX
10	**DSTO 11**	; P'(zk)
11	**R^^**	; cntlwd. back to X:
12	**DRCL 10**	; zk
13	XEQ "A"	; P(zk)
14	**^RDN**	; get rid of cntl. word
15	**DRCL 11**	; P'(zk)
16	**DR/**	; P/P'
17	**DRCL 10**	; zk
18	**DX<>DY**	
19	**DR-**	; zk – P(zk)/P'(zk)
20	**DSTO 10**	; zk+1
21	**LASTD**	; P/P'
22	**DRND**	; rounded
23	**DX=0?**	
24	GTO 11	; root found
25	**^RCL 01**	; cntl. word
26	**DRCL 10**	; zk
27	GTO 10	; repeat loop
28	LBL 11	
29	**DRCL 10**	; solution
30	**DAVIEW**	; show the world
31	END	; done.

Realize than more often than not this is not going to converge because of the twice-root condition on the real polynomial and its derivative, so without dual-complex support this is quite limited, just an academic exercise without much chances to become a fruitful method.

We can of course try it for a spin building a bespoken polydualnomial from its dual real roots, let's say:

$$P(z) = [z - (1+y\varepsilon)].[(z - (2+2\varepsilon)]^{\wedge}2\ ;$$
$$P(z)= z^{\wedge}3 + z^{\wedge}2\ (1+2\varepsilon) + z\ (2+2\varepsilon) + (4+10\varepsilon)$$

which supposedly has a twice-root in:

$z = [(1+1\varepsilon) + 2(2+2\varepsilon)] /3 = (5 +5\varepsilon) / 3 =$ 1.666666667 ÷ Σ 1.666666667

Left for the reader to complete ;-)

CODA: *What about out trusty quadratic equation?*

Sure enough, we're going to check if this works - at the very least as a programming exercise using the dual number functions.

The equation is Q(z) = A z^2 + B z + C , where A,B,C are dual numbers.

The routine below expects the three coefficients A, B,C in the dual stack levels DZ, DY and DX. It leaves the two roots in DY and DY – or comes back with a DATA ERROR if the discriminant of the square root is, dare I say, "negative" (i.e. the two roots are dual-complex numbers).

```
01  LBL "DQUAD"
02  DRCL Z          ; A to DX
03  DR/             ; C/A
04  DR<> Z          ; A to DX
05  DR/             ; B/A
06  ENTER^^         ; lifts d-stack
07  2
08  ST/ Z
09  ^/              ; B/2A
10  DRNEG           ; -B/2A
11  DENTER^
12  DENTER^
13  DX^2            ; (B/2A^2)
14  DRUP            ; C/A to DX
15  DR-             ; (b/2a)^2 - c/a
16  CF 00           ; default
17  X<0?            ; is Re(D)<0?
18  SF 00           ; yes, set flag
19  ABS             ; absolute value
20  DRSQRT          ; sqrt(D)
21  FS? 00          ; was D<0?
22  RTN             ; Dual Complex result
23  DR-             ; z1
24  DRUP            ; -B/2A to DX
25  LASTD           ; sqrt(D)
26  DR+             ; z2
27  DAVIEW          ; show first root
28  PSE             ; catch a glimpse
29  DX<>DY          ; swap them
30  DAVIEW          ; shows second root
31  END             ; done/
```

Example:

Get the two roots of Q(z) = [z – (1+ε)].[z – (1-2ε)]

First we expand the polydualnomial:, Q(z) = z^2 + (-2+ε) z + (1-ε)

We type (dual part first, remember):

0, ENTER^^, 1, ENTER^^, 1, ENTER^^, -2, ENTER^^, -1, ENTER^^, 1, XEQ "DQUAD"

Resulting:

z1 = 1-Σ0.500000000
z2 = 1-Σ0.500000000

Lo and behold, this is not the expected result! – but certainly *the value (1-Ɛ/2) is a root of Q(z):*

Q(1-Ɛ/2) = 0, and being a double root we can re-write the polydualnomial as follows:

Q(z) = [z – (1 – ε/2)]^2 = z^2 + (1-Ɛ/2)^2 – 2z (1-Ɛ/2) = z^2 + z (-2+Ɛ) + (1-ε)

Therefore, we've found FOUR roots of the quadratic equation, say what??

Interesting and confusing dual numbers to say the least – but nevertheless it's proof that the routine is working right… or is it? Keep reading to know more…

CODA2.- *Heresy or Paradox? Understanding this mess.*

The issue we've run into can be explained by going up one level to the actual polydualnomials we're trying to get the roots of. The vagaries of dual numbers are playing the trick on us because several polydualnomials reduce to the same polynomial when expanding their terms – sometimes they even reduce to "standard" polynomials of real variable.

Take for instance the polydualnomial formed from the roots $(1+\varepsilon)$ and $-(1+\varepsilon)$, i.e:

$P(z) = [\, z - (1+\varepsilon)]\ [\, z + (1+\varepsilon)]$,

Expanding it:

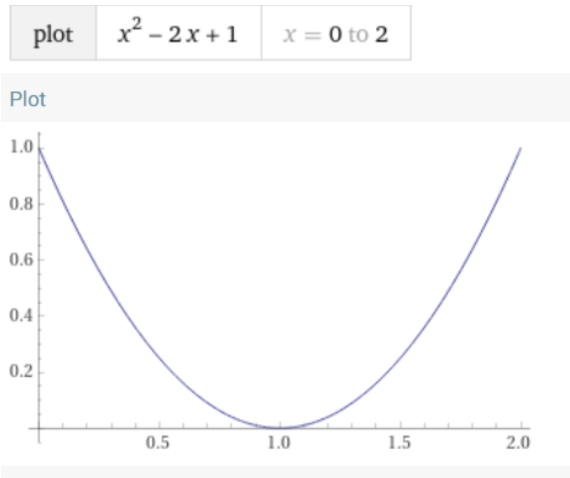
$P(z) = z^2 = z\,(1+\varepsilon) - z(1+\varepsilon) = z^2$, with a double root in $z=0$

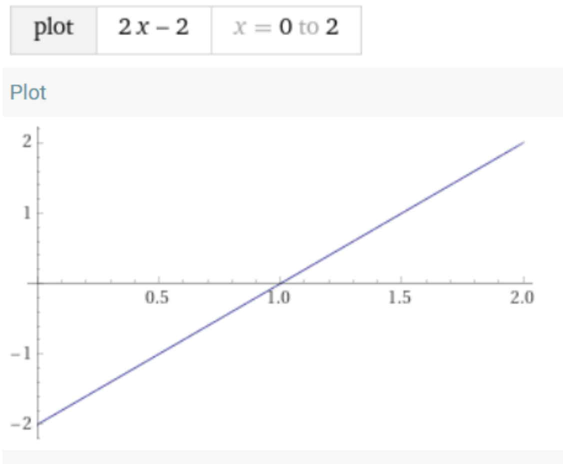
Another example showing the same dichotomy would be using the roots $(1+\varepsilon)$ and $(1-\varepsilon)$:

$Q(z) = [\, z - (1+\varepsilon)\,]\ [\, z - (1-\varepsilon)\,] = z^2 - z\,(1-\varepsilon) - z\,(1+\varepsilon) + 1 = z^2 - 2\,z + 1$

With double root $z = 1$

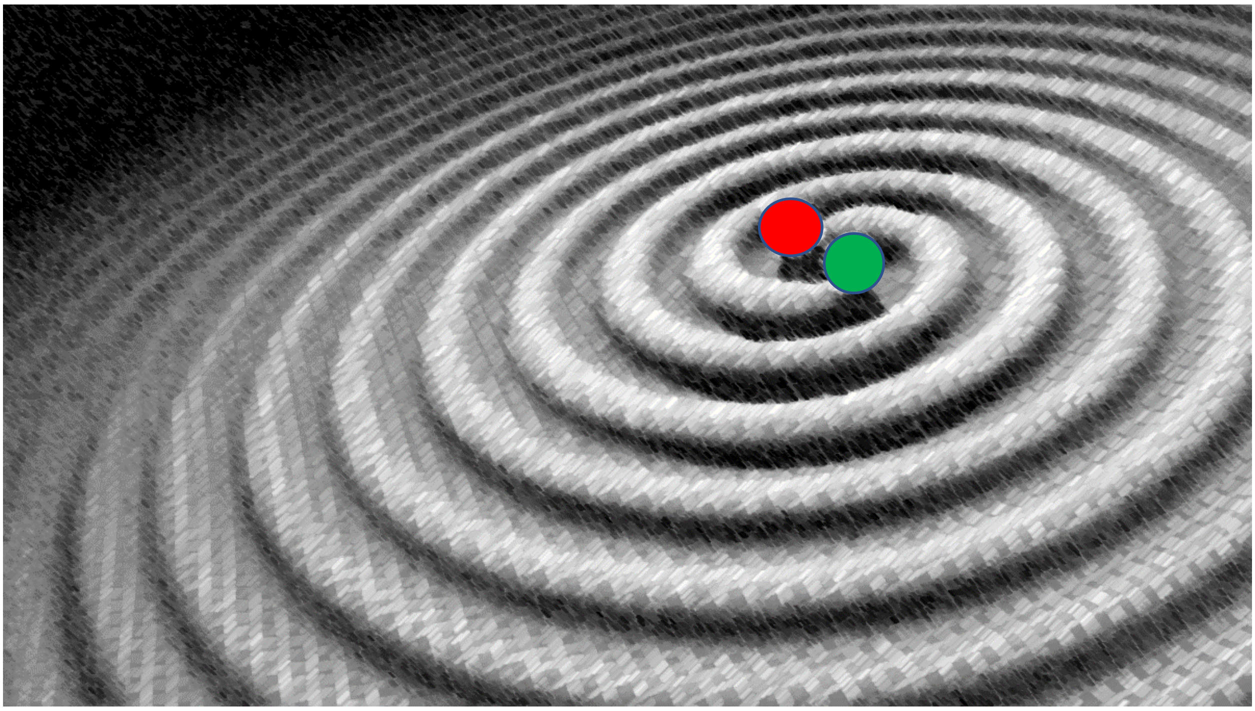
Note how they're twice-roots of both the polynomial and its derivative, so at least that part holds water-tight not creating more paradoxes:





PS. Somehow this duality brings to mind the quantum mechanics field paradoxes. I wonder if that's one reason why dual numbers are also applied in that field or if all this is just rubish to the square…

Part III – MCODE Listings



I/O_SVC Monitoring Routine.

This is the heart of the double length stack, and the best example of how MCODE facilitates things when really pushing the envelope. The routine filters the events by the following criteria:

1. Checks that ALPHA is not ON and that there's not a program running
2. Rejects calls from ROMs other than ROM #0
3. Rejects calls from routines different from [PARSE], at 0x0C93
4. Rejects keypresses not from the numeric keypad

If any of the above is false the routine sets UF 01 if CPU F11 is set, and then returns the polling vector back to the O/S to resume with the bus enumeration.

The first digit finds UF 01 clear, so it immediately sets it and proceeds to copy the contents of the T: register into the LGKT location, "last Good Known T" in buffer register G: It makes an audible sound to let the user know the backup has been made.

If however, UF 01 is found set that indicates a repeat digit event, and since we don't want to restore the LGTK value to the T: register with each digit entered the code exits as explained before.

See below the actual code with the details:

UPDATE	UPDATE	A226	00C	?FSET 3	PRGM mode on?
UPDATE		A227	09F	*JC +19d*	*yes, never mind!*
UPDATE		A228	28C	?FSET 7	*is ALPHA mode on??*
		A229	08F	*JC +17d*	*yes, never mind!*
this is the soul of the machine,		A22A	1B0	POPADR	*OK, grab the calling address*
First we discard calls not made		A22B	170	PUSHADR	*put it back*
from ROM#0		A22C	15C	PT= 6	
		A22D	262	C=C-1 @PT	*calling from ROM-0?*
UPDATE		A22E	063	*JNC +12d*	*no, ignore -> [LB_AB2B]*
UPDATE		A22F	03C	RCR 3	yes, only three digits
		A230	106	A=C S&X	*put in A for compares*
then we check whio's calling,		A231	130	LDI S&X	
discarding if not [PARSE]		A232	393	*[PARSE] address*	*for digit entry*
		A233	1F6	C=C+C XS	*"693"*
UPDATE		A234	1F6	C=C+C XS	*"C93"*
UPDATE		A235	366	?A#C S&X	got a match?
		A236	043	*JNC +08*	*yes, take care of it*
this catches all other functions		A237	18C	?FSET 11	stack lift enable?
even the native ones !		A238	019	*?C XQ*	*yes, Sets Uflag 01*
		A239	105	*->4106*	*[SF01]*
UPDATE	IGNORE	A23A	198	C=M ALL	*restore polling vector*
UPDATE		A23B	358	ST=C XP	*restore status bits*
UPDATE		A23C	3CD	*?NC GO*	*resume polling process*
UPDATE		A23D	09E	*->27F3*	*[RMCK10]*
UPDATE	DIGITS?	A23E	130	LDI S&X	
UPDATE		A23F	01B	*CON:*	*keycode limit*
		A240	106	A=C S&X	
third, which key was pressed?		A241	046	C=0 S&X	***valid keys are:***
only care for numeric pad		A242	39C	PT= 0	{ 10 - 19 } 0-9 digits,
		A243	098	C=G @PT,+	{ 1A, 1B } EEX and RADIX
UPDATE		A244	31C	PT= 1	
UPDATE		A245	362	?A#C @PT	*is it "1"*
UPDATE		A246	38F	*JC -15d*	*no, ignore -*
UPDATE		A247	306	?A<C S&X	*is KY > "1B"*
UPDATE		A248	37F	*JC -17d*	*yes, ignore*
UPDATE	*DIGITS*	A249	3B8	**READ 14(d)**	
UPDATE		A24A	1FE	C=C+C MS	
UPDATE		A24B	1FE	C=C+C MS	*sets carry if UF1 is SET*
		A24C	373	*JNC -18d*	*it's clear, no need to update!*

will only be done once	A24D	*02D*	*?NC XQ*	*Clears Uflag 01 to ignore next digits*
to account for the first digit	A24E	*104*	*->410B*	*[CF01]*
for the stack update	A24F	*169*	*?NC XQ*	*make a sound*
	A250	*10C*	*->435A*	*[TONE7]*
UPDATE	A251	*369*	*?NC XQ*	*Check buffer id#7 - > header in C*
UPDATE	A252	*124*	*->49DA*	*[CHKBF#7] - returns addr in A.X*
UPDATE	A253	*379*	*PORT DEP:*	*Lift buffer regs - always!*
UPDATE	A254	*03C*	*XQ*	*Expects header addr in A.X*
UPDATE	A255	*096*	*->A096*	*[BLIFT] - Ends w/ Chip0 Sel*
UPDATE	A256	1A6	A=A-1 S&X	*header addr in A.X*
UPDATE	A257	130	LDI S&X	
UPDATE	A258	***007***	***CON: 7***	*offset to G: reg*
UPDATE	A259	206	C=A+C S&X	*points at G::*
UPDATE	A25A	270	RAMSLCT	selects G:
UPDATE	A25B	**038**	**READATA**	***LGKT in C***
UPDATE	A25C	0AE	A<>C ALL	header
UPDATE	A25D	226	C=C+1 S&X	pointer to A:
UPDATE	A25E	270	RAMSLCT	selects A:
UPDATE	A25F	0AE	A<>C ALL	A: addr to A.X
UPDATE	A260	1A6	A=A-1 S&X	*header addr in A.X*
UPDATE	A261	*035*	*?NC XQ*	*puts G: in A:*
UPDATE	A262	*124*	*->490D*	*[WRTSEL] - selects Chip0*
UPDATE	A263	038	READATA	current T
UPDATE	A264	0EE	B<>C ALL	park it in B
UPDATE	A265	130	LDI S&X	
UPDATE	A266	***007***	***CON: 7***	*offset to G: reg*
UPDATE	A267	206	C=A+C S&X	*points at G:*
UPDATE	A268	270	RAMSLCT	selects G:
UPDATE	A269	0CE	C=B ALL	
UPDATE	A26A	*035*	*?NC XQ*	*puts T in LGKT*
UPDATE	A26B	*124*	*->490D*	*[WRTSEL] - selects Chip0*
UPDATE	A26C	*273*	*JNC -50d*	

It's important to realize the the I/O_SVC event is received by the module *after* the action has occurred, so it's not a true interrupt because we cannot intercept it and thus prevent the event from happening. In other words, when we react to it the T: register has already been deleted and the lower stack lifted to push the new digit into the X: register.

That's why we need to have a method to keep a backup copy of the T: register always up to date, in case there's a data entry event at any given moment. This backup is made as a follow-up task by all functions that alter the double-stack arrangement, either by directly handling their registers (like ENTER^^, R^^, ^RDN, ^LASTX, etc.) or indirectly as part of the automated stack drop and register duplication after a two-number function execution.

Finally, the last touch still needed is keeping UF 01 status always in sync with CPU F11 – as a proxy that can be interrogated at our discretion even *after* the O/S has dealt with the stack lift condition and thus it has re-set F11. Remember that our code is receiving the message *after the fact*, so checking F11 at that point is useless but UF 01 is still a valid marker for or purposes.

UF 01 is first set when the calculator is switched ON using the CALC_ON polling point. From that moment on, it's always refreshed by those functions that clear F11 (such as ^CLX, ENTER^^, CLDX and DENTER^), and by the I/O_SVC routine itself – remember it's the last thing it does upon a false event execution.

That's in a nutshell all there's to it – not rocket science but clever just the same, really a lot of fun to put together in such an interesting way.

Buffer Drop routine

Used to drop the upper-stack registers as part of the complete Stack drop, for instance in ^RDN, and the two-number Math functions.

BDROP	*Header*	A18C	090	"P"	*A -> CPU(B)*
BDROP	*Header*	A18D	00F	"O"	*A= B*
BDROP	*Header*	A18E	012	"R"	*B = C*
BDROP	*Header*	A18F	004	"D"	*C = D*
BDROP	*Header*	A190	002	"B"	*D stays put !*
BDROP	**BDROP**	**A191**	*369*	*?NC XQ*	*Check buffer id#7 - > header in C*
BDROP		A192	*124*	*->49DA*	*[CHKBF#7] - returns addr in A.X*
BDROP		A193	*379*	*PORT DEP:*	*Drop buffer regs, Saves b1 in B*
BDROP		A194	*03C*	*XQ*	*Expects header addr in A.X*
BDROP		A195	*198*	*->A198*	*[BDROP]*
BDROP		A196	*3C1*	*?NC GO*	
BDROP		A197	*002*	*->00F0*	*[NFRPU]*
BDROP	**BDROP**	**A198**	**0A6**	**A<>C S&X**	
BDROP		A199	226	C=C+1 S&X	*pointer to A*
BDROP		A19A	270	RAMSLCT	*selects A*
BDROP		A19B	106	A=C S&X	*park pt in A.X*
BDROP		A19C	038	READATA	A contents
BDROP		A19D	**0EE**	**C<>B ALL**	***saves b1(A) in B***
BDROP		A19E	1A6	A=A-1 S&X	*points at header*
BDROP		A19F	01C	PT= 3	*repeat three times*
BDROP	LOOP3	A1A0	0A6	A<>C S&X	*points to bR-1*
BDROP		A1A1	226	C=C+1 S&X	*points at bR*
BDROP		A1A2	226	C=C+1 S&X	*points at bR+1*
BDROP		A1A3	270	RAMSLCT	*selects bR+1*
BDROP		A1A4	106	A=C S&X	*park pt in A.X*
BDROP		A1A5	038	READATA	*bR+1 contents*
BDROP		A1A6	0AE	A<>C ALL	*points at bR+1*
BDROP		A1A7	266	C=C-1 S&X	*points at bR*
BDROP		A1A8	270	RAMSLCT	*selects bR*
BDROP		A1A9	0AE	A<>C ALL	
BDROP		A1AA	2F0	**WRTDATA**	***puts bR+1 in bR***
BDROP		A1AB	3D4	PT=PT-1	
BDROP		A1AC	394	?PT= 0	
BDROP		A1AD	*39B*	*JNC -13d*	
BDROP		A1AE	3E0	RTN	*why stop here?*

This routine saves A: into the CPU register [B], and moves { DCB } down to { CBA }. It leaves D unchanged and ends with the register A: selected, *not the Status regs* (!).

Note that a complete stack drop requires additional code to deal with the lower-stack registers and (critically) to "stitch" the upper and level parts adequately, so data moves up and down seamlessly.

In this case the lower stack will need to be drop as well, and the backup copy of A: saved in the CPU register [B] will need to be copied into T:

Buffer Lift routine

Used to lift the upper-stack registers as part of the complete Stack drop, for instance in R^^, ^LASTX, ^PI and ENTER^^.

0	BLIFT	*Header*	A089	*094*	**"T"**	*D = C*
0	BLIFT	*Header*	A08A	*006*	**"F"**	*C = B*
0	BLIFT	*Header*	A08B	*009*	**"I"**	*B = A*
0	BLIFT	*Header*	A08C	*00C*	**"L"**	*A = T*
0	BLIFT	*Header*	A08D	*002*	**"B"**	*T -> A -> B -> C -> D*
0	BLIFT	BLIFT	A08E	*369*	*?NC XQ*	*Check buffer id#7 - > header in C*
1	BLIFT		A08F	*124*	*->49DA*	*[CHKBF#7] - returns addr in A.X*
2	BLIFT	BLIFT?	A090	046	C=0 S&X	
3			A091	270	RAMSLCT	
4	*contingent to UF 01 status:*		A092	3B8	READ 14(d)	*buffer lift is conditioned*
5	*won't do if UF 01 is clear*		A093	1FE	C=C+C MS	*to the stack-lift being SET*
6			A094	1FE	C=C+C MS	*sets carry if UF1 is SET*
7	BLIFT		A095	**3A0**	**?NC RTN**	
8	BLIFT	BLIFT	A096	130	LDI S&X	
9	BLIFT		A097	*005*	***CON: 5***	*offset to E*
10	BLIFT		A098	146	A=A+C S&X	*points at E*
11	BLIFT		A099	01C	PT= 3	*will do three times*
12	BLIFT	REPEAT	A09A	0A6	A<>C S&X	*pointer to bR+2*
13	BLIFT		A09B	266	C=C-1 S&X	points to bR+1
14	BLIFT		A09C	266	C=C-1 S&X	points to bR
15	BLIFT		A09D	270	RAMSLCT	selects bR
16	BLIFT		A09E	106	A=C S&X	bR adr to A.X
17	BLIFT		A09F	**038**	**READATA**	reads bR contents
18	BLIFT		A0A0	0AE	A<>C ALL	*pointer to bR*
19	BLIFT		A0A1	226	C=C+1 S&X	pints to bR+1
20	BLIFT		A0A2	270	RAMSLCT	selects bR+1
21	BLIFT		A0A3	0AE	A<>C ALL	bR+1 adr to A.X
22	BLIFT		A0A4	2F0	**WRTDATA**	***puts bR in bR+1***
23	BLIFT		A0A5	3D4	PT=PT-1	
24	BLIFT		A0A6	394	?PT= 0	
25	BLIFT		A0A7	*39B*	*JNC -13d*	
26	BLIFT		A0A8	046	C=0 S&X	
27	BLIFT		A0A9	270	RAMSLCT	selects T
28	BLIFT		A0AA	038	READATA	T contents
29	BLIFT		A0AB	0AE	A<>C ALL	pointer to B: in C.X
30	BLIFT		A0AC	266	C=C-1 S&X	points to A:
31	BLIFT		A0AD	270	RAMSLCT	selects A:
32	BLIFT		A0AE	0AE	A<>C ALL	A: adr to A.X; T value to C
33	BLIFT		A0AF	*035*	*?NC GO*	*save C in selected RG*
34	BLIFT		A0B0	*126*	*->490D*	*[WRTSEL] - selects Chip0*

This routine first moves { ABC } into { BCD }, and then copies the contents of T: in the A: register prepare for further actions. The reoutine ends with the status registers selected.

Note that a complete stack lift requires additional code to deal with the lower-stack registers and (critically) to "stitch" the upper and level parts adequately, so data moves up and down seamlessly.

In this case the lower stack will be lifted for ENTER^ and ^LASTX, and possibly a copy of D: should be copied into X if we're performing R^^.

Stack Roll Up , ^PI and ^LASTX routines.

Here's an example that demonstrates the utilization of [BLIFT]. Note how in this case we rely on CPU F11 to divert the execution to the O/S in those instances when that's possible.

1	R^^	*Header*	A0C8	*09E*	**"^"**	
2	R^^	*Header*	A0C9	*01E*	**"^"**	
3	R^^	*Header*	A0CA	*012*	**"R"**	*Ángel Martin*
4	R^^	**R^^**	**A0CB**	*369*	*?NC XQ*	*Check buffer id#7 - > header in C*
7	R^^		A0CC	*124*	*->49DA*	*[CHKBF#7] - returns addr in A.X*
8	R^^		A0CD	130	LDI S&X	
9	R^^		A0CE	***004***	***CON: 4***	*points to D*
10	R^^		A0CF	206	C=A+C S&X	*points at D*
11	R^^		A0D0	270	RAMSLCT	*selects D*
12	R^^		A0D1	038	READATA	*contents of D*
13	R^^		A0D2	070	N=C ALL	*save for the end*
14	R^^	RUP*	A0D3	*379*	*PORT DEP:*	*Lift buffer regs - always!*
15	R^^	*T -> A -> B -> C -> D*	A0D4	*03C*	*XQ*	*Expects header addr in A.X*
16	R^^		A0D5	*096*	*->A096*	*[BLIFT] - Ends w. Chip0 Sel*
13	R^^	WRPUP2	A0D6	*3B5*	*?NC XQ*	*X-> Y-> Z-> T-> X*
14	R^^		A0D7	*050*	*->14ED*	*[R^SUB]*
15	R^^	WRPUP3	A0D8	0B0	C=N ALL	
16	R^^		A0D9	0E8	WRIT 3(X)	
17	R^^		A0DA	*379*	*PORT DEP:*	*Saves current T as LGKT*
18	R^^		A0DB	*03C*	*XQ*	*leaves buf addr in A.X*
19	R^^		A0DC	*272*	*->A272*	*[PSTFC#] - Ends w/ Chip0 Enabled*
20	R^^		A0DD	*3B9*	*?NC GO*	
21	R^^		A0DE	*002*	*->00EE*	*[NFRPR]*
1	LASTX^	*Header*	A0DF	*09E*	**"^"**	
2	LASTX^	*Header*	A0E0	*018*	**"X"**	
3	LASTX^	*Header*	A0E1	*014*	**"T"**	
4	LASTX^	*Header*	A0E2	*013*	**"S"**	
5	LASTX^	*Header*	A0E3	*001*	**"A"**	
6	LASTX^	*Header*	A0E4	*00C*	**"L"**	*Ángel Martin*
7	LASTX^	**LASTX^**	**A0E5**	**18C**	**?FSET 11**	
8	LASTX^		A0E6	*0A1*	*?NC GO*	*the OS will ddo the job*
9	LASTX^		A0E7	*04A*	*->1228*	*[LASTX]*
10	LASTX^		A0E8	138	READ 4(L)	
11	LASTX^		A0E9	070	N=C ALL	
12	LASTX^	WRAPUP	A0EA	*369*	*?NC XQ*	*Check buffer id#7 - > header in C*
13	LASTX^		A0EB	*124*	*->49DA*	*[CHKBF#7] - returns addr in A.X*
14	LASTX^		A0EC	*379*	*PORT DEP:*	*Lift buffer regs *if* F1 is set*
15	LASTX^		A0ED	*03C*	*XQ*	*Expects header addr in A.X*
16	LASTX^		A0EE	*090*	*->A090*	*[BLIFT?] - Ends w/ Chip0 Sel*
17	LASTX^		A0EF	*33B*	*JNC -25d*	*[WRPUP2]*
1	PI^	*Header*	A0F0	*089*	**"I"**	
2	PI^	*Header*	A0F1	*010*	**"P"**	
3	PI^	*Header*	A0F2	*01E*	**"^"**	*Ángel Martin*
4	PI^	**PI^**	**A0F3**	**18C**	**?FSET 11**	
5	PI^		A0F4	*109*	*?NC GO*	*the OS will ddo the job*
6	PI^		A0F5	*04A*	*->1242*	*[PI]*
7	PI^		A0F6	2A0	SETDEC	
8	PI^		A0F7	*269*	*?NC XQ*	
9	PI^		A0F8	*064*	*->199A*	*[PI/2]*
10	PI^		A0F9	1EE	C=C+C ALL	
11	PI^		A0FA	23A	C=C+1 M	*rounding?*
12	PI^		A0FB	046	C=0 S&X	*truncation to 10-digit*
13	PI^		A0FC	070	N=C ALL	*save for the end*
14	PI^		A0FD	260	SETHEX	
15	PI^		A0FE	*363*	*JNC -20d*	*[WRAPUP]*

Clear Dual/Double Stack routines.

Here's the combination of ^CLST and CLDST routines, controlled by CPU F7. Very straight forward and not really challenging, so we do it nice and clean.

1	CLDST	*Header*	A1B0	*094*	**"T"**	
2	CLDST	*Header*	A1B1	*013*	**"S"**	
3	CLDST	*Header*	A1B2	*004*	**"D"**	
4	CLDST	*Header*	A1B3	*00C*	**"L"**	
5	CLDST	*Header*	A1B4	*003*	**"C"**	*Ángel Martin*
6	CLDST	**CLDST**	**A1B5**	**288**	**SETF 7**	
7	CLDST		A1B6	*369*	*?NC XQ*	*Check buffer id#7 - > header in C*
8	CLDST		A1B7	*124*	*->49DA*	*[CHKBF#7] - returns addr in A.X*
9	CLDST		A1B8	04E	C=0 ALL	
10	CLDST		A1B9	15C	PT= 6	*will do 6 times*
11	CLDST		A1BA	**05B**	***JNC +11d***	
1	CLST^	*Header*	A1BB	*094*	**"T"**	
2	CLST^	*Header*	A1BC	*013*	**"S"**	
3	CLST^	*Header*	A1BD	*00C*	**"L"**	
4	CLST^	*Header*	A1BE	*003*	**"C"**	
5	CLST^	*Header*	A1BF	*01E*	**"^"**	*Ángel Martin*
6	CLST^	**CLST^**	**A1C0**	**284**	**CLRF 7**	
7	CLST^		A1C1	*369*	*?NC XQ*	*Check buffer id#7 - > header in C*
8	CLST^		A1C2	*124*	*->49DA*	*[CHKBF#7] - returns addr in A.X*
9	CLST^		A1C3	04E	C=0 ALL	
10	CLST^		A1C4	05C	PT= 4	*will do 4 times*
11	CLST^	NXTBRG	A1C5	0AE	A<>C ALL	
12	CLST^		A1C6	226	C=C+1 S&X	*points to next bR*
13	CLST^		A1C7	270	RAMSLCT	*selects bR*
14	CLST^		A1C8	0AE	A<>C ALL	*zero to C*
15	CLST^		A1C9	2F0	**WRTDATA**	*clears bR*
16	CLST^		A1CA	3D4	PT=PT-1	
17	CLST^		A1CB	394	?PT= 0	
18	CLST^		A1CC	*3CB*	*JNC -07*	*[NXTBRG]*
19	CLST^		A1CD	04E	C=0 ALL	
20	CLST^		A1CE	270	RAMSLCT	
21	CLST^		A1CF	028	WRIT 0(T)	
22	CLST^		A1D0	068	WRIT 1(Z)	
23	CLST^		A1D1	0A8	WRIT 2(Y)	
24	CLST^		A1D2	0E8	WRIT 3(X)	
25	CLST^		A1D3	*02D*	*?NC XQ*	*Clears Uflag 01 !*
26	CLST^		A1D4	*104*	*->410B*	*[CF01]*
27	CLST^		A1D5	*379*	*PORT DEP:*	*Saves current T as LGKT*
28	CLST^		A1D6	*03C*	*XQ*	*leaves buf addr in A.X*
29	CLST^		A1D7	*272*	*->A272*	*[PSTFC#] - Ends w/ Chip0 Enabled*
30	CLST^		A1D8	28C	**?FSET 7**	*"DR" case?*
31	CLST^		A1D9	*309*	*?NC GO*	*no, to the O/S*
32	CLST^		A1DA	*002*	*->00C2*	*[NFRSIG]*
31	CLST^	*DUALREL*	A1DB	*369*	*PORT DEP:*	*Show result*
32	CLST^		A1DC	*03C*	***GO***	
33	CLST^		A1DD	*2A8*	*->A2A8*	*[DVIEW?]*

Double Stack Math routines.

MATH	Header	A10D	0AA	"*"	
MATH	Header	A10E	01E	"^"	Ángel Martin
MATH	**MULT^**	**A10F**	**084**	**CLRF 5**	
MATH		A110	023	JNC +04	
MATH	Header	A111	0AF	"/"	
MATH	Header	A112	01E	"^"	Ángel Martin
MATH	**DIV^**	**A113**	**088**	**SETF 5**	
MATH	BOTH1	A114	148	SETF 6	
MATH		A115	04B	JNC +09	
MATH	Header	A116	0AB	"+"	
MATH	Header	A117	01E	"^"	Ángel Martin
MATH	**PLUS^**	**A118**	**084**	**CLRF 5**	
MATH		A119	023	JNC +04	
MATH	Header	A11A	0AD	"-"	
MATH	Header	A11B	01E	"^"	Ángel Martin
MATH	**MINUS^**	**A11C**	**088**	**SETF 5**	
MATH	BOTH2	A11D	144	CLRF 6	
MATH	MERGE	A11E	1A5	?NC XQ	Check for valid entries
MATH		A11F	100	->4069	[CHKST2] - sets DEC mode
MATH		A120	14C	?FSET 6	DIV/MUTL?
MATH		A121	043	JNC +08	no, skip over
MATH		A122	08C	?FSET 5	
MATH	DIV/MLT	A123	261	?C XQ	
MATH		A124	061	->1898	[DV2_10]
MATH		A125	08C	?FSET 5	
MATH		A126	135	?NC XQ	
MATH		A127	060	->184D	[MP2-10]
MATH		A128	03B	JNC +07	
MATH	PLUS/MIN	A129	08C	?FSET 5	
MATH		A12A	013	JNC +02	
MATH		A12B	2BE	C=-C-1 MS	
MATH	PLUS	A12C	000	NOP	let carry settle
MATH		A12D	01D	?NC XQ	
MATH		A12E	060	->1807	[AD2-10]
MATH	CODA2	A12F	**070**	**N=C ALL**	the result
MATH		A130	0A5	?NC XQ	
MATH		A131	050	->1429	[OVFL10]
MATH		A132	0D4	?PT= 10	
MATH		A133	289	?C GO	bad boy
MATH		A134	003	->00A2	[ERROF]
MATH	REPL_D	A135	260	SETHEX	
MATH		A136	369	?NC XQ	Check buffer id#7 - > header in C
MATH		A137	124	->49DA	[CHKBF#7] - returns addr in A.X
MATH		A138	379	PORT DEP:	Drop buffer regs, Saves b1 in B
MATH		A139	03C	XQ	Expects header addr in A.X
MATH		A13A	198	->A198	[BDROP]
MATH		A13B	046	C=0 S&X	
MATH		A13C	270	RAMSLCT	
MATH		A13D	038	READATA	T contents
MATH		A13E	10E	A=C ALL	save it in A.ALL
MATH		A13F	078	READ 1(Z)	
MATH		A140	0A8	**WRIT 2(Y)**	puts Z in Y
MATH		A141	0AE	A<>C ALL	
MATH		A142	068	**WRIT 1(Z)**	puts T in Z
MATH		A143	0CE	C=B ALL	recovers b1 contents
MATH		A144	028	**WRIT 0(T)**	puts A: in T
MATH		A145	379	PORT DEP:	Saves current T as LGKT
MATH		A146	03C	XQ	leaves buf addr in A.X
MATH		A147	272	->A272	[PSTFC#] - Ends w/ Chip0 Enabled
MATH		A148	0B0	C=N ALL	
MATH		A149	331	**?NC GO**	
MATH		A14A	002	->00CC	[NFRX]

Termination of DRCL, DSTO, DR<> and DVIEW

This was one of the more finicky routines to write, mostly because it's common to the four dual-number memory actions: recalling, storing, exchanging and viewing dual number values in RAM.

Starting with SWAP and VIEW, the first think we need to determine is where in RAM is the target dual number: either in data registers, in lower-stack location (the native stack) or in upper-stack location (i.e. buffer #7).

1	SWPSTO	ENDSWP	A7E4	0B8	READ 2(Y)	get X,Y in B,M
2			A7E5	158	M=C ALL	so they can be accessed
3	common entry point for		A7E6	0F8	READ 3(X)	while Chip0 is not active
4	both SWAP and STO		A7E7	0EE	B<>C ALL	
5			A7E8	130	LDI S&X	STACK RG#
			A7E9	004	DSTK levels left	value limit
ZSTO	Y	ZRCL	A7EA	106	A=C S&X	
	X		A7EB	0B0	C=N ALL	recall REG# adr
			A7EC	306	?A<C S&X	is C > 4 ?
	R(nn+1)		A7ED	1E7	JC +60d	yes, recall REG# adr
	Rnn		A7EE	266	C=C-1 S&X	L = 0?
			A7EF	0AF	JC +21d	yes, do nothing -> [DAVIEW]
13	SWPSTO		A7F0	266	C=C-1 S&X	L = 1?
14	SWPSTO		A7F1	18B	JNC +49d	no, must be 2 or 3 -> [DX<>BF]
15	SWPSTO	L=1	A7F2	38C	?FSET 0	SWAP case?
16	SWPSTO		A7F3	023	JNC +04	no, we're STORING
17	SWPSTO		A7F4	341	PORT DEP:	yes, do DX<>DY
18	SWPSTO		A7F5	08C	GO	
19	SWPSTO		A7F6	3AC	->A3AC	[DX<>DY]
20	SWPSTO	DYUPDT	A7F7	198	C=M ALL	
21	SWPSTO		A7F8	028	WRIT 0(T)	
22	SWPSTO		A7F9	0CE	C=B ALL	
23	SWPSTO		A7FA	068	WRIT 1(Z)	
24	SWPSTO		A7FB	04B	JNC +09	[DAVIEW]
25	ENDRCL	ENDVEW	A7FC	130	LDI S&X	STACK RG#
26	ENDRCL		A7FD	004	DSTK levels left	value limit
27	ENDRCL		A7FE	106	A=C S&X	
28	ENDRCL		A7FF	0B0	C=N ALL	recall REG# adr
29	ENDRCL		A800	306	?A<C S&X	is C > 4 ?
30	ENDRCL		A801	147	JC +40d	yes, recall REG# adr
31	ENDRCL		A802	266	C=C-1 S&X	L = 0?
32	ENDRCL		A803	023	JNC +04	no, skip over
33	ENDRCL	LEVL=0	A804	341	PORT DEP:	Show result
34	ENDRCL		A805	08C	GO	
35	ENDRCL		A806	2A8	->A2A8	[DVIEW?]
36	ENDRCL	NOTZER	A807	266	C=C-1 S&X	LEVL = 1?
37	ENDRCL		A808	0D3	JNC +26d	no, recall STKBUF adr
38	ENDRCL	LEVL=1	A809	341	PORT DEP:	Show result
39	ENDRCL		A80A	08C	GO	
40	ENDRCL		A80B	2AE	->A2AE	[DYVIEW]
41	ENDRCL	ENDRCL	A80C	130	LDI S&X	STACK RG#
42	ENDRCL		A80D	004	DSTK levels left	value limit
43	ENDRCL		A80E	106	A=C S&X	
44	ENDRCL		A80F	0B0	C=N ALL	recall REG# adr
			A810	306	?A<C S&X	is C > 4 ?
ZSTO	Y	ZRCL	A811	0C7	JC +24d	yes, recall REG# adr
	X		A812	266	C=C-1 S&X	Level = 0?
			A813	033	JNC +06	no, -> [NOTZER]
	R(nn+1)		A814	18C	?FSET 11	stack lift enabled?
	Rnn		A815	1DB	JNC +59d	no, just show result
			A816	341	PORT DEP:	yes, divert to DENTER
52	ENDRCL		A817	08C	GO	i.e. RCL ST X
53	ENDRCL		A818	359	->A359	[DENTER^]

The RCL part uses the status of F11 to determine whether to lift the stack before copying the target dual number into the DX stack level. We can do it this way because F11 is sync'd back with UF 01 by the DRCL main function code, not shown here as it occurs prior to this section.

54	ENDRCL	NOTZER	A819	266	C=C-1 S&X	*Level = 1?*
55	ENDRCL		A81A	043	*JNC +08*	*no, recall STKBUF adr*
56	ENDRCL	*Level=1*	A81B	046	C=0 S&X	
57	ENDRCL		A81C	270	RAMSLCT	
58	ENDRCL		A81D	038	READATA	
59	ENDRCL		A81E	158	M=C ALL	*puts T in M*
60	ENDRCL		A81F	078	READ 1(Z)	
61	ENDRCL		A820	0EE	B<>C ALL	*puts Z in B*
62	ENDRCL		A821	**103**	***JNC +32d***	*[MERGE]*
63	SWPSTO	DX<>BF	A822	*369*	*?NC XQ*	*Check buffer id#7 - > header in C*
64	SWPSTO		A823	*124*	*->49DA*	*[CHKBF#7] - returns addr in A.X*
65	SWPSTO		A824	0B0	C=N ALL	*level#: 2 or 3*
66	SWPSTO		A825	266	C=C-1 S&X	either 1 or 2
67	SWPSTO		A826	1E6	C=C+C S&X	either 2 or 4
68	SWPSTO		A827	266	C=C-1 S&X	either 1 or 3
69	SWPSTO		A828	206	C=A+C S&X	*bR addr*
70	SWPSTO	**DX<>RG**	**A829**	**270**	**RAMSLCT**	*select Br / Reg#*
71	SWPSTO		A82A	106	A=C S&X	*save addr in A.X for later*
72	SWPSTO	*unused if STO*	A82B	038	READATA	*bR contents*
73	SWPSTO		A82C	0EE	B<>C ALL	*bring X value to C*
74	SWPSTO		A82D	*20C*	*?FSET 2*	***RCL / VIEW cases?***
75	SWPSTO		A82E	***017***	***JC +02***	*yes, skip*
76	SWPSTO		A82F	**2F0**	**WRTDATA**	*puts X In bR*
77	SWPSTO	RCL1	A830	0A6	A<>C S&X	*pointer to bR in C.X*
78	SWPSTO	*unused if STO*	A831	226	C=C+1 S&X	*points at bR+1*
79	SWPSTO		A832	270	RAMSLCT	*selects bR+1*
80	SWPSTO		A833	038	READATA	*bR+1 contents*
81	SWPSTO		A834	1D8	C<>M ALL	*bring X value to C*
82	SWPSTO		A835	*28C*	*?FSET 7*	***VIEW case?***
83	SWPSTO		A836	***05B***	***JNC +11d***	no, skip
84	SWPSTO	*SHOWRG*	A837	046	C=0 S&X	
85	SWPSTO		A838	270	RAMSLCT	selects Chip0
86	SWPSTO		A839	0CE	C=B ALL	
87	SWPSTO		A83A	128	WRIT 4(L)	*real part*
88	SWPSTO		A83B	198	C=M ALL	
89	SWPSTO		A83C	070	N=C ALL	*dual part*
90	SWPSTO		A83D	**244**	**CLRF 9**	RECT mode
91	SWPSTO		A83E	*341*	*PORT DEP:*	*Show result*
92	SWPSTO		A83F	*08C*	***GO***	*even if running PRGM*
93	SWPSTO		A840	*2C1*	*->A2C1*	*[DAVEW#]*
94	SWPSTO	NOVIEW	A841	*20C*	*?FSET 2*	*RCL case?*
95	SWPSTO		A842	*017*	*JC +02*	*yes, skip*
96	SWPSTO		A843	**2F0**	**WRTDATA**	*puts Y in bR+1*
97	SWPSTO	RCLMRG	A844	046	C=0 S&X	
98	SWPSTO		A845	270	RAMSLCT	selects Chip0
99	SWPSTO		A846	*20C*	*?FSET 2*	*RCL case?*
100	SWPSTO		A847	*043*	*JNC + 08*	*no, skip Stack Lift*
101	SWPSTO		A848	18C	?FSET 11	*stack lift enabled?*
102	SWPSTO		A849	*043*	*JNC +08*	*no, skip DS lift*
103	SWPSTO		A84A	088	SETF 5	*subroutine mode*
104	SWPSTO		A84B	*349*	*PORT DEP:*	*Lift complete D-Stack*
105	SWPSTO		A84C	*08C*	*XQ*	*Buffer and Stack*
106	SWPSTO		A84D	*35E*	*->A35E*	*[DSTLFT] -uses N*

The View routine pointed at here is shared by the **DAVIEW** and the **DVIEW** functions. It's also the ending part of every dual-number function when used in manual mode, so the display presents the combination of but real and dual parts in the proper format.

Some instructions are not needed for all cases but keeping them makes the routine compatible for the four actions, so as long as they don't mangle the source data they're run even if not necessary.

On the other hand, some other instructions are only executed when needed, as controlled by the appropriate case-telling flags.

94	SWPSTO	NOVIEW	A841	20C	?FSET 2	RCL case?
95	SWPSTO		A842	017	JC +02	yes, skip
96	SWPSTO		A843	2F0	WRTDATA	puts Y in bR+1
97	SWPSTO	RCLMRG	A844	046	C=0 S&X	
98	SWPSTO		A845	270	RAMSLCT	selects Chip0
99	SWPSTO		A846	20C	?FSET 2	RCL case?
100	SWPSTO		A847	043	JNC + 08	no, skip Stack Lift
101	SWPSTO		A848	18C	?FSET 11	stack lift enabled?
102	SWPSTO		A849	043	JNC +08	no, skip DS lift
103	SWPSTO		A84A	088	SETF 5	subroutine mode
104	SWPSTO		A84B	349	PORT DEP:	Lift complete D-Stack
105	SWPSTO		A84C	08C	XQ	Buffer and Stack
106	SWPSTO		A84D	35E	->A35E	[DSTLFT] -uses N
107	SWPSTO		A84E	01B	JNC +03	
108	SWPSTO	DXWRT?	A84F	38C	?FSET 0	SWAP case?
109	SWPSTO		A850	02B	JNC +05	no, skip DX update
110	SWPSTO	DCUPDT	A851	0CE	C=B ALL	bR contents
111	SWPSTO		A852	0E8	WRIT 3(X)	puts bR in X
112	SWPSTO		A853	198	C=M ALL	bR+1 contents
113	SWPSTO		A854	0A8	WRIT 2(Y)	puts bR+1 in Y
114	SWPSTO	DSHOW	A855	188	SETF 11	enable stack lift!
			A856	341	PORT DEP:	Save LAST & Show result
115	SWPSTO		A857	08C	GO	
116	SWPSTO		A858	2A5	->A2A5	[LSTSHW?]

From here the execution moves to write the T: register into the LGKT location (setting UP 01 if needed), and then to show the result in the display if we're in manual mode:

1	LGKT	POSTFC	A270	019	?NC XQ	Sets Uflag 01
2	LGKT		A271	104	->4106	[SF01]
3	LGKT	PSTFC#	A272	046	C=0 S&X	
4	LGKT		A273	270	RAMSLCT	
5	LGKT		A274	038	READATA	current T
6	LGKT		A275	0EE	B<>C ALL	park it in B
7	LGKT		A276	369	?NC XQ	Check buffer id#7 - > header in C
8	LGKT		A277	124	->49DA	[CHKBF#7] - returns addr in A.X
9	LGKT		A278	130	LDI S&X	
10	LGKT		A279	007	CON: 7	offset to G: reg
11	LGKT		A27A	206	C=A+C S&X	points at G:
12	LGKT		A27B	270	RAMSLCT	selects G:
13	LGKT		A27C	0CE	C=B ALL	
14	LGKT		A27D	035	?NC GO	puts T in LGKT
15	LGKT		A27E	126	->490D	[WRTSEL] - selects Chip0

Dual-Number Stack Roll Up

Here's the complete DRUP routine code. Note the two calls to the buffer drop routine, and to the [R^SUB] routine to deal with the regular stack – as well as the stitching components to keep things in good shape.

DRUP	*Header*	A2EB	090	"P"	
DRUP	*Header*	A2EC	015	"U""	***DR Roll Up***
DRUP	*Header*	A2ED	012	"R"	
DRUP	*Header*	A2EE	004	"D"	*Ángel Martin*
DRUP	**DRUP**	**A2EF**	369	?NC XQ	*Check buffer id#7 - > header in C*
DRUP		A2F0	124	->49DA	*[CHKBF#7] - returns addr in A.X*
DRUP		A2F1	0A6	A<>C S&X	*buffer header*
DRUP		A2F2	106	A=C S&X	*keep in A.X*
		A2F3	158	M=C ALL	*save in M for later*
first backup {C,D} in {B,N}		A2F4	130	LDI S&X	
		A2F5	004	**CON: 4**	*offset to D*
DRUP		A2F6	206	C=A+C S&X	*points at D*
DRUP		A2F7	270	RAMSLCT	*selects D*
DRUP		A2F8	106	A=C S&X	*D addr to A.X*
DRUP		A2F9	038	READATA	*D contents*
DRUP		A2FA	070	**N=C ALL**	*puts D in N*
DRUP		A2FB	0A6	A<>C S&X	
DRUP		A2FC	266	C=C-1 S&X	points at C
DRUP		A2FD	270	RAMSLCT	
DRUP		A2FE	038	READATA	
DRUP		A2FF	**0EE**	**B<>C ALL**	*puts C in CPU-B*
DRUP	*2XLIFT*	A300	198	C=M ALL	*header addr to C.X*
		A301	106	A=C S&X	*needed by [BLIFT]*
D B		A302	379	PORT DEP:	*Lift buffer regs*
C A		A303	03C	XQ	*Expects header addr in A.X*
B T		A304	096	->A096	*[BLIFT] - Ends w/ Chip0 Sel*
A Z		A305	3B5	?NC XQ	*X-> Y-> Z-> T-> X*
T Y		A306	050	->14ED	*[R^SUB]*
Z X		A307	198	C=M ALL	*header addr to C.X*
Y D		A308	106	A=C S&X	*needed by [BLIFT]*
X C		A309	379	PORT DEP:	*Lift buffer regs*
		A30A	03C	XQ	*Expects header addr in A.X*
DRUP		A30B	096	->A096	*[BLIFT] - Ends w/ Chip0 Sel*
DRUP		A30C	3B5	?NC XQ	*X-> Y-> Z-> T-> X*
DRUP		A30D	050	->14ED	*[R^SUB]*
DRUP		A30E	0B0	C=N ALL	
DRUP		A30F	0A8	WRIT 2(Y)	*puts D in Y*
DRUP		A310	0CE	C=B ALL	
DRUP		A311	0E8	WRIT 3(X)	*puts C in X*
DRUP		A312	369	PORT DEP:	*Save LAST & Show result*
DRUP		A313	03C	**GO**	
DRUP		A314	2A5	->A2A5	*[LSTSHW?]*

Dual-Number Stack Roll Down

Here's the complete DRDN routine code, a bit more involved that the previous case. Note how we deal with the stack registers first, and afterwards we take care of the buffer registers with the two calls to the buffer drop routine as well. The final part copies the original {X,Y} registers to the buffer top registers {C,D}, notice the numerous RAMSLCT/READATA instructions making the usual mess to move between different RAM areas.

1	DRDN	*Header*	A316	*08E*	**"N"**	
2	DRDN	*Header*	A317	*004*	**"D"**	***DR Roll Down***
3	DRDN	*Header*	A318	*012*	**"R"**	
4	DRDN	*Header*	A319	*004*	**"D"**	*Ángel Martin*
5	DRDN	**DRDN**	**A31A**	**0F8**	**READ 3(X)**	
6	DRDN		A31B	268	WRIT 9(Q)	*first we save X,Y in N,Q*
7	DRDN		A31C	0B8	READ 2(Y)	*needed for later*
8	DRDN		A31D	070	N=C ALL	
9	DRDN	*XY<>ZT*	A31E	*3B5*	*?NC XQ*	
10	DRDN		A31F	*050*	*->14ED*	*[R^SUB]*
11	DRDN		A320	*3B5*	*?NC XQ*	
12	DRDN		A321	*050*	*->14ED*	*[R^SUB]*
13	DRDN	*2XDROP*	A322	*369*	*?NC XQ*	*Check buffer id#7 - > header in C*
14	DRDN		A323	*124*	*->49DA*	*[CHKBF#7] - returns addr in A.X*
15	DRDN		A324	0A6	A<>C S&X	*buffer header*
16	DRDN		A325	106	A=C S&X	*keep in A.X*
17	DRDN		A326	158	M=C ALL	*save in M for later*
18	DRDN		A327	*379*	*PORT DEP:*	*Drop buffer regs, Saves [A] in B*
19	DRDN		A328	*03C*	*XQ*	*Expects header addr in A.X*
20	DRDN		A329	*198*	*->A198*	*[BDROP]*
21			A32A	046	C=0 S&X	
22	**D Y**		A32B	270	RAMSLCT	*select chip0*
23	**C X**		A32C	0CE	C=B ALL	*A contents*
24	**B D**		A32D	**068**	**WRIT 1(Z)**	***puts A IN Z***
25	**A C**		A32E	198	C=M ALL	*buffer header*
26	**T B**		A32F	106	A=C S&X	*needed by [BDROP]*
27	**Z A**		A330	*379*	*PORT DEP:*	*Drop buffer regs, Saves [A] in B*
28	**Y T**		A331	*03C*	*XQ*	*Expects header addr in A.X*
29	**X Z**		A332	*198*	*->A198*	*[BDROP]*
30			A333	046	C=0 S&X	
31	DRDN		A334	270	RAMSLCT	*select chip0*
32	DRDN		A335	0CE	C=B ALL	*B contents*
33	DRDN		A336	**028**	**WRIT 0(T)**	***puts B in T***
34	DRDN	*XY -> CD*	A337	198	C=M ALL	
35	DRDN		A338	106	A=C S&X	*buffer header*
36	DRDN		A339	278	READ 9(Q)	*original X*
37	DRDN		A33A	158	M=C ALL	*X saved in M*
38	DRDN		A33B	*130*	LDI S&X	
39	DRDN		A33C	***004***	***CON: 4***	*offset to D*
40	DRDN		A33D	206	C=A+C S&X	*points at D*
41	DRDN		A33E	270	RAMSLCT	*selects D*
42	DRDN		A33F	106	A=C S&X	*addr to A.X*
43	DRDN		A340	0B0	C=N ALL	
44	DRDN		A341	2F0	**WRTDATA**	***puts Y in D***
45	DRDN		A342	0A6	A<>C S&X	
46	DRDN		A343	266	C=C-1 S&X	
47	DRDN		A344	270	RAMSLCT	
48	DRDN		A345	198	C=M ALL	
49	DRDN		A346	2F0	**WRTDATA**	***puts X in C***
50	DRDN		A347	046	C=0 S&X	
51	DRDN		A348	270	RAMSLCT	
52	DRDN		A349	*369*	*PORT DEP:*	*Save LAST & Show result*
53	DRDN		A34A	*03C*	***GO***	
54	DRDN		A34B	*2A5*	*->A2A5*	*[LSTSHW?]*

Dual Number Math routines.

The structure for all math functions is the same: there is a main program that orchestrates the admin tasks and calls the specific math subroutine that does the actual legwork, using the O/S 13-digit routines as often as possible. This brings a lot of consistency to the code and makes editing and maintenance much easier. See below the Trigonometric functions example:

47	DTRIG	Header	A888	08E	"N"	
48	DTRIG	Header	A889	009	"I"	**Dual Sine**
49	DTRIG	Header	A88A	013	"S"	
50	DTRIG	Header	A88B	004	"D"	Ángel Martin
51	DTRIG	DSIN	A88C	1A5	?NC XQ	Check {X, Y} data
52	DTRIG		A88D	100	->4069	[CHKST2] - sets DEC
53	DTRIG		A88E	260	SETHEX	
54	DTRIG		A88F	36D	PORT DEP:	Save DX in LastDX
55	DTRIG		A890	08C	XQ	
56	DTRIG		A891	000	->A400	[D2LAST]
57	DTRIG		A892	379	PORT DEP:	Calculate Sin z
58	DTRIG		A893	03C	XQ	writes result in X.Y
59	DTRIG		A894	05A	->A85A	[DSIN#]
60	DTRIG	DSHOW	A895	341	PORT DEP:	Show result
61	DTRIG		A896	08C	GO	
62	DTRIG		A897	2A8	->A2A8	[DVIEW?]
63	DTRIG	Header	A898	093	"S"	
64	DTRIG	Header	A899	00F	"O"	**Dual Cosine**
65	DTRIG	Header	A89A	003	"C"	
66	DTRIG	Header	A89B	004	"D"	Ángel Martin
67	DTRIG	DCOS	A89C	1A5	?NC XQ	Check {X, Y} data
68	DTRIG		A89D	100	->4069	[CHKST2] - sets DEC
69	DTRIG		A89E	260	SETHEX	
70	DTRIG		A89F	36D	PORT DEP:	Save DX in LastDX
71	DTRIG		A8A0	08C	XQ	
72	DTRIG		A8A1	000	->A400	[D2LAST]
73	DTRIG		A8A2	379	PORT DEP:	Calculates Cos {X,Y}
74	DTRIG		A8A3	03C	XQ	
75	DTRIG		A8A4	06E	->A86E	[DCOS#]
76	DTRIG		A8A5	383	JNC -16d	
77	DTRIG	Header	A8A6	08E	"N"	**Dual Tangent**
78	DTRIG	Header	A8A7	001	"A"	Dtan z = Dsin z / Dcos z
79	DTRIG	Header	A8A8	014	"T"	
80	DTRIG	Header	A8A9	004	"D"	Ángel Martin
81	DTRIG	DTAN	A8AA	1A5	?NC XQ	Check {X, Y} data
82	DTRIG		A8AB	100	->4069	[CHKST2] - sets DEC
83	DTRIG		A8AC	260	SETHEX	
84	DTRIG		A8AD	36D	PORT DEP:	Save DX in LastDX
85	DTRIG		A8AE	08C	XQ	
86	DTRIG		A8AF	000	->A400	[D2LAST]
87	DTRIG		A8B0	379	PORT DEP:	Calculate Sin z
88	DTRIG		A8B1	03C	XQ	writes result in X.Y
89	DTRIG		A8B2	05A	->A85A	[DSIN#]
90	DTRIG		A8B3	048	SETF 4	silent mode
91	DTRIG		A8B4	084	CLRF 5	no truncation
92	DTRIG		A8B5	104	CLRF 8	LASTD option
93	DTRIG		A8B6	349	PORT DEP:	Subroutine entry for DBLIFT
94	DTRIG		A8B7	08C	XQ	Buffer and Stack
95	DTRIG		A8B8	35E	->A35E	[DSTLFT] - uses N
96	DTRIG		A8B9	379	PORT DEP:	Calculates Cos {X,Y}
97	DTRIG		A8BA	03C	XQ	writes result in X.Y
98	DTRIG		A8BB	06E	->A86E	[DCOS#]
99	DTRIG		A8BC	36D	PORT DEP:	Calcculates {Z,T} / {X,Y}
100	DTRIG		A8BD	08C	XQ	and checks overflow
101	DTRIG		A8BE	048	->A448	[DRDIV#]
102	DTRIG		A8BF	333	JNC -26d	

In fact this example makes an exception in the DTAN sace, which is calculated based on the [DSIN#] and [DCOS#] subroutines instead of having its own dedicated one.

Common to the three functions you see the initial check for ALPHA data and saving of the argument in the DL stack level (call to [D2LAST]}. In the ending part is the final diversion to the [OVFL2] code to check the integrity of the calculated results. The final values are not written into XY if any of the two the overflow tests fail.

DTRIG	DSIN#	A85A	2A0	SETDEC	
DTRIG		A85B	3C4	ST=0	skips [TRGSET] $\sin(u + v\varepsilon$
DTRIG		A85C	048	SETF 4	result in RAD
DTRIG		A85D	0F8	READ 3(X)	x
DTRIG		A85E	070	N=C ALL	required by [TRG100]
DTRIG		A85F	22D	?NC XQ	Cos(y) - skipping [TRGSET]
DTRIG		A860	048	->128B	[COS1]
DTRIG		A861	11E	A=C MS	bug or what??
DTRIG		A862	0B8	READ 2(Y)	y
DTRIG		A863	13D	?NC XQ	
DTRIG		A864	060	->184F	[MP1-10]
DTRIG		A865	128	WRIT 4(L)	dual part
DTRIG		A866	3C4	ST=0	skips [TRGSET]
DTRIG		A867	048	SETF 4	result in RAD
DTRIG		A868	0F8	READ 3(X)	x
DTRIG		A869	070	N=C ALL	required by [TRG100]
DTRIG		A86A	2EE	?C#0 ALL	bug when zero!
DTRIG		A86B	229	?C XQ	Sin(y) - skipping [TRGSET]
DTRIG		A86C	049	->128A	[SIN1]
DTRIG		A86D	0A3	JNC +20d	[WRAPUP]
DTRIG	DCOS#	A86E	2A0	SETDEC	
DTRIG		A86F	3C4	ST=0	skips [TRGSET]
DTRIG		A870	048	SETF 4	result in RAD
DTRIG		A871	0F8	READ 3(X)	y
DTRIG		A872	070	N=C ALL	required by [TRG100]
DTRIG		A873	229	?NC XQ	Sin(y) - skipping [TRGSET]
DTRIG		A874	048	->128A	[SIN1]
DTRIG		A875	2BE	C=-C-1 MS	
DTRIG		A876	11E	A=C MS	$\cos z = \cos x - ($
DTRIG		A877	0B8	READ 2(Y)	y
DTRIG		A878	13D	?NC XQ	
DTRIG		A879	060	->184F	[MP1-10]
DTRIG		A87A	128	WRIT 4(L)	
DTRIG		A87B	3C4	ST=0	skips [TRGSET]
DTRIG		A87C	048	SETF 4	result in RAD
DTRIG		A87D	0F8	READ 3(X)	
DTRIG		A87E	070	N=C ALL	required by [TRG100]
DTRIG		A87F	22D	?NC XQ	Cos(y) - skipping [TRGSET]
DTRIG		A880	048	->128B	[COS1]
DTRIG	WRAPUP	A881	070	N=C ALL	real part to N
DTRIG		A882	138	READ 4(L)	
DTRIG		A883	0EE	B<>C ALL	dual part to B
DTRIG		A884	260	SETHEX	
DTRIG		A885	365	PORT DEP:	puts result in {X,Y}
DTRIG		A886	08C	GO	and checks overflow
DTRIG		A887	161	->A561	[DOVFL2]

Appendix.- Valentín Albillo's STKN FOCAL Program

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

Program characteristics. –

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

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

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

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

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

Instructions.

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

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

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

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

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

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

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