My suggested new features for ***Free42***

by Valentín Albillo (2020)

Introduction

These are my well-pondered and researched suggestions of functionalities which could be implemented in Thomas Okken's awesome *HP-42S* simulator, ***Free42*** [1], to very significantly increase its capabilities with minimum effort. All of them are based on the functionalities that I've frequently found necessary to use in my own programs to either simplify the programming effort or to very significantly increase the speed but alas, they weren't currently available at the moment. This might be the time to remedy these omissions for good !

In my educated opinion, many of them are *really very easy* to implement, most are fairly easy, and anyway I'm including a color tag next to each one to indicate my fair estimation of the implementation *difficulty:* *Trivial*, *Very Easy*, *Easy* and *Moderate*. There aren't any really hard cases among the rated ones.
I'm also including an estimation of the general *usefulness* for each one: *Useful*, *Very Useful* and *Most Useful*

For easy reference, I've classified all the instructions and functionalities I'm proposing in the following *Sections*:

1. **Math functions**
   - 1.1 Basic functions
   - 1.2 Advanced functions
   - 1.3 Number Theoretic functions
   - 1.4 Statistics functions
2. **Matrix operations and functions**
3. **Polynomial-related functions**
4. **Constants**
5. **Loop-related functions and functionalities**
6. **Utility keywords**
7. **Additional useful functionalities**
8. **Conclusion**

---

[1] Some relevant links related to *Free42*, the *DM42* (its *physical incarnation*) and the original *HP-42S*:

Main site for *Free42*:	https://thomasokken.com/free42/
Main site for *DM42*:	https://www.swissmicros.com/product/dm42
Description at *MoHPC*:	https://www.hpmuseum.org/hp42s.htm
Owner's Handbook:	https://literature.hpcalc.org/community/hp42s-om-en.pdf
Programming Techniques:	https://literature.hpcalc.org/community/hp42s-prog-en.pdf
My *HP-42S/Free42* articles:	HP Article VA010 - Long Live the HP-42S
	HP Article VA040a - Boldly Going - Mandelbrot Set Area (42S)
	HP Article VA042a - Boldly Going - Outsmarting PROOT (42S)

Detailed examples for the equivalent *HP-71B* functions: HP Article VA044 - HP-71B Math Pac 2 Comments and Proposals

1. Math functions

1.1 Basic functions

CEIL *Trivial* *Useful*

Ceiling: `CEIL(3.4) = 4, CEIL(5) = 5, CEIL(-2.3) = -2, CEIL(-0.1) = 0`

Stack use: as other 1-argument functions, such as `SIN`, i.e. the function value replaces the argument in `ST X`, which gets saved in `ST L (LASTX)`.

Rationale: standard math function present in most programming languages and instruction sets but absent in the *HP-42S*. Emulating it takes several steps and somewhat obscures the code.

FLOOR *Trivial* *Useful*

Floor, just another name for `IP`, should be included for consistency with `CEIL`.

SIGNZ *Trivial* *Very Useful*

Standard *Sign* function: `SIGNZ(PI) = 1, SIGNZ(0) = 0, SIGNZ(-PI) = -1`

- doesn't support alpha data or matrices
- *does* support *complex* numbers: `SIGNZ(3 + i4) = 0.6 + i0.8`

Stack use: as other 1-argument functions, such as `SIN`.

Rationale: this function is necessary, as the built-in `SIGN` function returns ***1*** for an argument of ***0*** for compatibility with the *HP-41C*, but this is utterly non-standard *(i.e.:"plain wrong")* from a mathematical point of view and severely affects the usability of the built-in function for general purposes.

CONJ *Trivial* *Useful*

Complex conjugate: `CONJ(2 + i3) = 2 - i3, CONJ(5) = 5, CONJ(-5) = -5`

Stack use: as other 1-argument functions, such as `SIN`.

Rationale: again, this frequently used basic function is missing from the *HP-42S* instruction set, and emulating it takes several steps, is slower and obscures the code.

2↑X and **LOG2** *Trivial* *Useful*

Binary log and antilog: `2↑10 = 1024, LOG2(33554432) = 25, LOG2(PI) = 1.651496+`

Stack use: as other 1-argument functions, such as `SIN`.

Rationale: this pair of functions, each one the inverse of the other, are also missing. They're quite useful when dealing with anything binary or with exponential duplication, orders of magnitude, etc.

X↑3 and **CURT** *Trivial* *Useful*

Cube and cube root: `X↑3(-17) = -4913, X↑3(-2 + i3) = 46 + i9, CURT(-4913) = -17`

Stack use: as other 1-argument functions, such as `SIN`.

Rationale: again, a missing pair of a function and its inverse, akin to `X↑2` and `SQRT`. They're useful for anything having to do with volumes/radii, take several steps to emulate and are much faster.

→FRC *Easy* *Very Useful*

Convert to Fraction: `FRC(PI,1E-6) → X:355, Y:113, FRC(PI,1E-9) → X: 103993, Y: 33102`

```
T:  T                T:  T
Z:  Z                Z:  Z
Y:  tol              Y:  numerator
X:  X    →FRC →      X:  denominator
L:  L                L:  X
```

Rationale: in many scientific and engineering applications it's frequently necessary to substitute a real number by a suitable equivalent fraction within a specified tolerance (for instance, when designing gears), and also to rationalize the coefficients of a LSQ functional fit or approximation.

NSPLIT *Trivial* *Useful*

Splits a number into mantissa and exponent: `NSPLIT(5.23E-34) → X: 5.23, Y: -34`

```
T:  T                T:  Z
Z:  Z                Z:  Y
Y:  Y                Y:  exponent of N
X:  N    NSPLIT →    X:  mantissa of N
L:  L                L:  N
```

Rationale: when dealing with really large numbers, exceeding even the ±*6,144* exponent range of *Free42 Decimal* (such as, say, *1,000,000 ! ~ 8.264…E5565708*), which appear in *Statistics*, simulations and research), working separately with mantissas and exponents is but mandatory.

NJOIN *Trivial* *Useful*

Joins mantissa and exponent into a number: `NJOIN(5.23,-34) = 5.23E-34`

```
T:  T                  T:  T
Z:  Z                  Z:  T
Y:  exponent           Y:  Z
X:  mantissa   NJOIN → X:  N = mantissa*10^exponent
L:  L                  L:  mantissa
```

Rationale: this is the inverse function of `NSPLIT`, see its *Rationale* above.

1.2 Advanced Functions

LAMW *Moderate* *Very Useful*

Lambert's ***W*** *function*: `LAMW(-0.2) = -0.259171101818, LAMW(1) = 0.567143290411`

Stack use: as other 1-argument functions, such as `SIN`.

Rationale: *Lambert W* function, recently popularized, has awesome mathematical properties and an ever-growing number of important applications in all fields, to the point that it's been proposed as a new *"elementary"* function to join the ranks of exponentials, logarithms and trigs. Every worthy math package includes it but few calculators do, so it would be highly desirable to implement it in *Free42*.

AGM *Easy* *Very Useful*

Arithmetic-Geometric Mean: `AGM(1,3)= 1.86361678324, AGM(5,7)= 5.9579660133`

Stack use: as other 2-argument functions, such as `COMB`.

Rationale: the `AGM` function has a plethora of important applications, e.g.: to quickly and efficiently compute many essential functions in engineering applications (such as the *elliptic* functions), and even to compute π itself at unsurpassed speeds. As with the *Lambert W* function, few or no calculators compute the *AGM* out of the box, so it would make a fine addition to the *Free42* instruction set.

1.3 Number Theoretic functions

Rationale: the *HP-42S* completely lacks any *Number Theoretic* functions, even such *"kindergarten"* ones as *GCD (Greatest Common Divisor)* and *LCM (Least Common Multiple)*, let alone higher-level ones, which are available in the instruction sets of most advanced calculators and pocket computers. All are very useful in various fields, including anything having to do with cryptography, checksums, analysis of algorithms and data structures, any research on *Number Theory* subjects, and even working *on US$ 1,000,000* prize problems such as the *Riemann's Hypothesis.*

GCD *Very easy* *Useful*

Greatest Common Divisor: `GCD(28702,117181) = 113, GCD(113,317) = 1`

Stack use: as other 2-argument functions, such as `COMB`.

LCM *Very easy* *Useful*

Least Common Multiple: `LCM(28702,117181) = 29763974, LCM(5,7) = 35`

Stack use: as other 2-argument functions, such as `COMB`.

MMUL *Easy* *Very Useful*

*Modular Multiplication (A*B) mod M*: `MMUL(123456789,987654321,3141592654) = 2910639175`

`MMUL(10123465234878998,65746311545646431,10005412336548794) = 4652135769797794`

```
T:  T                T:  T
Z:  A                Z:  T
Y:  B                Y:  T
X:  M     MMUL  →    X:  (A*B) MOD M
L:  L                L:  M
```

MEXP *Easy* *Most Useful*

Modular Exponentiation (A^B) mod M: `MEXP(6,5,10001) = 7776, MEXP(23,391,55) = 12`
`MEXP(113, 2305843009213693950, 2305843009213693951) = 1` *(the modulus M is prime)*

```
T:  T                T:  T
Z:  A                Z:  T
Y:  B                Y:  T
X:  M     MEXP  →    X:(A^B) MOD M
L:  L                L:  M
```

PRIM *Moderate* *Useful*

N^th Prime: `PRIM(1) = 2, PRIM(5) = 11, PRIM(1E4) = 104729, PRIM(1E5) = 1299709`

Stack use: as other 1-argument functions, such as `SIN`.

Rationale: having this function available makes it unnecessary to generate on the fly a table or range of primes or having them coded inline for programs which need a supply of them.

PRIMQ *Moderate* *Useful*

Returns whether the argument is *Prime* or *Composite*: `PRIMQ(100) = 0, PRIMQ(113) = 1`
where *0 = False* (the number is *Composite*) and *1 = True* (the number is a *Prime / Probable Prime*).

Stack use: as other 1-argument functions, such as `SIN`.

PRIMF *Moderate* *Useful*

Smallest Prime Factor: `PRIMF(117181) = 17, PRIMF(117181/17) = 61`

Stack use: as other 1-argument functions, such as `SIN`.

Rationale: repeatedly using this function makes it easy to obtain the complete factorization of any integer numbers, even very large ones (say up to 30 digits in *Free42 Decimal*)

FPRIM *Easy* *Useful*

Find Next Prime: `FPRIM(113) = 113, FPRIM(114) = 127, FPRIM(1E6) = 1000003`

Stack use: as other 1-argument functions, such as `SIN`.

P! (*aka* **N#**) *Easy* *Useful*

Primorial: `P!(5) = 2*3*5*7*11 = 2310, P!(19) = 7858321551080267055879090`

Stack use: as other 1-argument functions, such as `SIN`.

PHI *Moderate* *Useful*

Euler's Totient: `PHI(113) = 112, PHI(2520) = 576, PHI(5000) = 2000`

Stack use: as other 1-argument functions, such as `SIN`.

MOEB *Moderate* *Useful*

Moebius function: `MOEB(8) = 0, MOEB(13) = -1, MOEB(14) = 1`

Stack use: as other 1-argument functions, such as `SIN`.

1.4 Statistics functions

RANG *Easy* *Very Useful*

Random Number generation with standard *Gaussian Distribution* of mean ***0*** and standard deviation ***1***:

Stack use: the stacks lifts and the value in placed in `ST X`, as with `RCL`. *LAST x* `(ST L)` is unaffected.

Rationale: The built-in **RAN** function only produces (pseudo-)random numbers *uniformly* distributed, but there are various important applications that do require *Gaussian* distributions because they fit many real-life processes in all areas from biology to finance. No calculator that I know of includes a *Gaussian random number generator* so **RANG** will be a welcome and pretty useful first.

NCDF *Moderate* *Useful*

Cumulative normal distribution: `NCDF(0, 1, 2) = 0.97724986805`

Stack use: as other 3-argument functions, i.e. the stack drops and the function value replaces the argument in `ST X`, which gets saved in `ST L` *(LAST x)*.

NICDF *Moderate* *Useful*

Inverse cumulative normal distribution: `NICDF(0, 1, 0.841344746069) = 1`

Stack use: as other 3-argument functions like `NCDF`.

NORMD *Moderate* *Useful*

Normal probability density: `NORMD(0, 2, 0.5) = 0.193334058402`

Stack use: as other 3-argument functions like `NCDF`.

ERF and **IERF** *Moderate* *Useful*

Error function and *Inverse Error function*: `ERF(1) = 0.842700793, IERF(0.95228512) = 1.4`

Stack use: as other 1-argument functions, such as `SIN`.

2. Matrix operations and functions

MIDN *Very easy* *Very Useful*

Creates in ST X a real NxN square *Identity Matrix*:

```
T:  T                T:  T
Z:  Z                Z:  Z
Y:  Y                Y:  Y
X:  N    MIDN   →    X:  [NxN Identity Matrix]
L:  L                L:  N
```

Rationale: A very basic function that is inexplicably absent from the *HP-42S* instruction set, all the more as creating the *Identity* matrix from scratch requires a number of steps in a loop. It's useful for initializing many matrix procedures, such as evaluating matrix series (exponential of a matrix), etc.

MRAN *Very easy* *Most Useful*

Creates in ST X a *Random Matrix* filled up with random values: $minV \leq V < maxV$:

```
T:  T                      T:  T
Z:  Z                      Z:  T
Y:  {M,N}                  Y:  Z
X:  {minV, maxV}  MRAN  →  X:  [MxN Random Matrix]
L:  L                      L:  (minV, maxV)
```

Rationale: when doing simulations or testing algorithms there's frequently the need to use some random data as initial starting values to see if the algorithm behaves as expected or to seed the first stage of the simulation, and in general random matrices find frequent use in all sorts of tasks. Creating and filling up a matrix with random values in *RPN* requires loops and is rather clumsy and slow, so having **MRAN** to do it in a single step at compiled high-level language speeds is extremely convenient.

MPOW *Very easy* *Useful*

Returns to ST X the N^{th} *power of a square matrix*:

```
T:  T                    T:  T
Z:  Z                    Z:  T
Y:  [matrix]             Y:  Z
X:  N        MPOW   →    X:  [matrix]^N
L:  L                    L:  N
```

where *N* is an integer. The computation goes as follows:

- If $N > 1$ then *Matrix^N = Matrix * Matrix * ... * Matrix*
- If $N = 1$ then *Matrix^N = Matrix*
- if $N = 0$ then *Matrix^N = Identity Matrix*
- if $N = -1$ then *Matrix^N = Inverse Matrix*
- if $N < -1$ then *Matrix^N = Inverse Matrix * Inverse Matrix * ... * Inverse Matrix*

Rationale: another matrix function that's very easy to implement (repeated matrix multiplications using a binary decomposition) and conveniently freeing the user from having to code it as a slow *RPN* loop. Possible applications are many, for instance to help evaluate more sophisticated transcendental matrix functions such as *exp(**A**)* or *sin(**A**)*, where ***A*** is a matrix, checking if a matrix is *nilpotent*, etc.

MSORT *Easy* or *Moderate*, depending on the sorting algorithm used. *Most Useful*

Sorts the elements of a matrix in ascending order.

```
T:  T                        T:  T
Z:  Z                        Z:  Z
Y:  Y                        Y:  Y
X:  {range}   MSORT var  →   X:  [sorted matrix]
L:  L                        L:  {range}
```

where *range = { first element, last element }* specifies the range of elements to sort. If *range = 0* then all the elements are sorted. The sort is in ascending order but if descending order is required then simply use MREV (see below) immediately afterwards to reverse the order.

Rationale: sorting data is a most frequent operation and doing it in *RPN* is a very slow affair. For large datasets one would go for complex, fast algorithms such as *Quicksort* or *Heapsort*, but when using the *HP-42S* one usually needs to sort datasets having under 100 elements or so, in which case the best sorting algorithms aren't really needed and using ***Insert***- or ***Shell Sort*** is perfectly adequate.

MREV *Trivial* *Useful*

Reverses the order of the elements of a matrix (useful for sorted matrices and *Taylor* series, etc.)

```
T:  T                        T:  T
Z:  Z                        Z:  Z
Y:  Y                        Y:  Y
X:  {range}   MREV var  →    X:  [reversed matrix]
L:  L                        L:  {range}
```

where *range = {first element, last element}* specifies the range of elements to reverse. If *range = 0* then all the elements are reversed.

Rationale: another basic array function which works well with other functions (such as with MSORT to change the sorting order), while being trivial to implement.

TRACE *Very easy* *Useful*

Returns to ST X the *Trace* (sum of diagonal elements) of a real/complex square matrix:

```
T:  T                      T:  T
Z:  Z                      Z:  Z
Y:  Y                      Y:  Y
X:  [matrix]  TRACE  →     X:  Trace
L:  L                      L:  [matrix]
```

Rationale: the *trace* (called ***Tr*** in the literature) is a basic scalar-valued matrix function, which finds many uses, e.g.: computing the coefficients of the *Characteristic Polynomial* of a square matrix (while also computing accurate inverses and determinants as byproducts, as seen in my PCHAR subprogram.)

3. Polynomial-related functions

PEVAL *Very easy* *Most Useful+*

Evaluates a polynomial (whose coefficients are stored in a matrix) for either a single value or for all the values stored in another matrix, returning the single value or all the computed values in a matrix created in `ST X`. The coefficients and the arguments can be real and/or complex.

```
T:  T                              T:  T
Z:  Z                              Z:  Z
Y:  Y                              Y:  Y
X:  argument    PEVAL var  →       X:  P(argument)
L:  L                              L:  argument
```

where:

var is a real/complex matrix which holds the coefficients of the polynomial to evaluate, stored in decreasing order of powers of *x* (i.e: $2x^3-5x+7$ is stored as *[2, 0, -5, 7]*).

argument is either a single real/complex value or a real/complex matrix.

P(argument) is either a real/complex value or a matrix containing the values of *P(argument)*

Rationale: Polynomial evaluation is a most frequent task in any number of disciplines, from curve fitting to interpolation to root finding to evaluating *Taylor* expansions to whatever, so doing it as quickly and conveniently as possible will greatly simplify and speed up many programs.

For instance, let's consider my **PZER** rootfinder *RPN* subprogram which, for the 50 iterations needed to find all 100 complex roots of the sample 100th-degree polynomial, has to evaluate it some *5,000 times* in all, each evaluation requiring a tight 10-step loop which is executed 100 times per evaluation, so replacing this loop by a *single* **PEVAL** instruction will *greatly* speed up the root-finding process !

Implementing this instruction is just a matter of using *Horner's Scheme* to evaluate an Nth-degree polynomial, which can be done very fast and accurately using just N multiplications and additions.

PDER *Easy* *Very Useful*

Returns a matrix with the coefficients of the N^{th} *derivative of a polynomial* whose coefficients are stored as a matrix in a variable. The result matrix has the same type real/complex as the coeffs. matrix.

```
T:  T                              T:  T
Z:  Z                              Z:  Z
Y:  Y                              Y:  Y
X:  N     PDER var  →              X:  [result]
L:  L                              L:  N
```

where:

var stores a real/complex matrix which holds the coefficients of the polynomial,
result is a real/complex matrix which holds the coeffs. of the N^{th}-derivative of the pol., and is already in the format required for `PEVAL` to evaluate the derivative.

Example: $P(x) = 225\,x^4 - 425\,x^3 + 170\,x^2 + 370\,x + 100$, stored as matrix *[225,-425,170,370,100]* in variable *MYPOL*, and we want the coefficients of its first and second derivatives:

```
1, PDER "MYPOL"  →  matrix[900, -1275, 340, 370]  (coefs of the first derivative)
2, PDER "MYPOL"  →  matrix[2700, -2550, 340]      (coefs of the 2nd derivative)
```

Rationale: finding the derivatives of a polynomial in *exact* form is an essential task and **PDER** makes it as fast, accurate and simple as possible. Working together with **PEVAL** the applications are endless, e.g.: finding the extrema and points of inflection of a polynomial (which might be a *Taylor Series* or a least-squares fit to experimental scientific and engineering data), etc.

PZER *Moderate+* *Most Useful*

Returns all real/complex roots of a polynomial whose coefficients are stored in a matrix. The roots are returned in another matrix. See my subprogram PZER in the article mentioned below.

```
T:  T                          T:  T
Z:  Z                          Z:  Z
Y:  Y                          Y:  Y
X:  {opt.pars}    PZER var  →  X:  [roots]
L:  L                          L:  {opt.pars}
```

where:

var stores a real/complex matrix which holds the coefficients of the polynomial.

roots is a *complex* matrix which holds the real/complex computed roots.

opt.pars is either ***0*** or a complex value *{tol, maxiter}* where *0* specifies default parameterss for *tol* (tolerance) and *maxiter* (max. number of iterations).

See my article *"HP Article VA042a - Boldly Going - Outsmarting PROOT (42S)"* for details about **PZER** implemented as an *RPN* subprogram for *HP-42S/Free42*. Implementing **PZER** as a built-in function in *Free42* instead of a user *RPN* program will greatly increase its speed.

PCHAR *Easy* *Very Useful*

Returns in a real/complex matrix the real/complex coefficients of the *Characteristic Polynomial* of a real/complex NxN square matrix, that can then be passed to **PZER** to compute all its real/complex roots, which are the matrix ***eigenvalues***.

```
T:  T                  T:  Z
Z:  Z                  Z:  Y
Y:  Y                  Y:  x
X:  X   PCHAR var  →   X:  [CP coeffs.]
L:  L                  L:  L
```

Rationale: computing *eigenvalues* of a matrix is essential in many important areas of science and engineering as well as analysis of algorithms, etc. This instruction simplifies it by computing the coefficients of the polynomial whose roots are the eigenvalues, which are then found quickly and accurately by **PZER**, i.e., using just *two* instructions, another great example of the synergy between the proposed **PEVAL**, **PDER**, **PCHAR** and **PZER** instructions. Further, **PCHAR** can also compute with enhanced accuracy and return the *Inverse* matrix and the *Determinant* as byproducts at no cost.

See my article *"HP Article VA047 - Boldly Going - Eigenvalues and Friends"* for details about PCHAR implemented as a subprogram for the *HP-71B.*

4. Constants

All of them: *Trivial* *Useful*

`MAXREAL` = *Maximum positive number* that *Free42* can represent (depends on version, *Decimal* or *Binary*).

`MINREAL` = *ditto* but *Smallest positive number* instead.

> *Rationale:* these two values have many uses in programs, e.g.: to check for limits or to detect abnormal termination conditions, similar to the uses of *NaNs* and *Infinities*, which the *HP-42S* number system also lacks. Another useful case would be for running programs to be able to determine programmatically whether they're being run under *Free42 Decimal* or under *Free42 Binary*, as the values for `MAXREAL` and `MINREAL` will be different for each and can be checked to see what's the case.

Also, the table below includes the **8** proposed ***constants*** that I find most useful. Additional ones are perfectly possible, to be included in the free rows at the bottom:

Proposed constants

Name	*Description*	*Steps replaced*	*Value*	*Rationale*
kPHI	*Golden Ratio*	`5, SQRT, 1, +, 2, ÷`	`1.618033984…`	*cleaner, fast, shorter, saves 1 level*
kPI2	*π / 2*	`PI, 2, ÷`	`1.570796326…`	*ditto*
k2PI	*2 π*	`PI, STO+ ST X`	`6.283185307…`	*cleaner, faster, shorter*
kE	*e*	`1, EXP`	`2.718281828…`	*ditto*
kGAMMA	*Euler's Gamma*	`.57721566…`	`0.577215664…`	*cleaner, faster, much shorter*
kCPLX0	*Complex Zero*	`0, ENTER, COMPLEX`	`0 + i0`	*cleaner, fast, shorter, saves 1 level*
kCPLXI	*Complex Unity*	`0, ENTER, 1, COMPLEX`	`0 + i1`	*ditto*
kLSPEED	*Lightspeed (m/s)*	`299792458`	`299,792,458`	*cleaner, fast, shorter, useful for astronomical distances and times*
?				
?				
?				
?				

5. Loop-related functions and functionalities

INCR / INCR nnn *Trivial* *Very Useful*

Increments a numeric value in `ST X` by one or *nnn* (*1* to *999*). Doesn't test anything or skip any steps, just increments the value as fast as possible:

```
T:  T                T:  T
Z:  Z                Z:  Z
Y:  Y                Y:  Y
X:  X  INCR nnn →    X:  X+nnn
L:  L                L:  X
```

DECR / DECR nnn *Trivial* *Very Useful*

Decrements a numeric value in `ST X` by one or *nnn* (*1* to *999*). Doesn't test anything or skip any steps, just decrements the value as fast as possible:

```
T:  T                T:  T
Z:  Z                Z:  Z
Y:  Y                Y:  Y
X:  X  DECR nnn →    X:  X-nnn
L:  L                L:  X
```

XISG / XDSE *Easy* *Most Useful*

Extended `ISG/DSE`, work like the built-in ones but using an *extended loop control number* like this,

iiiii.fffffdd (if 12-digit) or better, instead of: ***iiiiiii.fffdd***

which was kept that way for compatibility with *HP-41C* programs (for a 10-digit calculator) but it's unnecessarily limiting for a 12-digit calculator like the *HP-42S* (let alone *Free42*), and makes the existing `ISG` useless if the final counter value has more than 3 digits, which frequently is the case.

IRCL, IRCL+, IRCL-, IRCLx, IRCL÷ *Trivial* *Most Useful*

Integer Part Recall: useful to recall (from within a loop, for instance) just the *integer part* of the current value of an index stored in a register/variable in the usual *ccccccc.fffii* format (or the extended format proposed above). This way, we can save much time and have cleaner and shorter code by just doing, e.g.: `IRCLx nn` *(1 step)* instead of `RCL nn, IP, x` *(3 steps)*, resulting in significant runtime savings for time-consuming loops.

6. Utility keywords:

X=NN?, X>NN?, X>=NN? , etc. *Easy* *Most Useful*

The six *HP-41CX* tests comparing ST X with register NN. Quite useful to, for example, check a value in ST X versus some limit (stored in a register or variable) without disturbing the ST X register.

FS?T, FC?T *Trivial* *Most Useful*

Flag Set/Clear test and Toggle: works exactly like the existing FS?S/FS?C/FC?S/FC?C flag tests but instead of setting/clearing the flag after the test it now *toggles* it: if the flag was *set* then it's *cleared* and vice versa.

EVEN?, ODD? *Trivial* *Most Useful*

Parity Tests: work like any other tests, following the *do-if-true rule*, i.e.: executing the next step if the value in ST X is even/odd respectively, and skipping it otherwise. Allows for easy *"every other"* programming, as well as integer programming and *Number Theory* applications.

TF *flag* *Trivial* *Most Useful*

Toggle flag: like SF *(Set Flag)* and CF *(Clear Flag)*, this isn't a flag test as it doesn't test the status of the flag or skips steps, but simply *toggles* the flag: if the flag was *set* then it's *cleared* and vice versa.

POP / **POP N** *Trivial* *Very Useful*

Pop Return level(s): deletes one/N pending returns. Frequently used to transfer execution back to a return level other than the previous one because of some condition, like an error which needs special handling.

STOST *Trivial* *Most Useful*

Store Stack: stores internally the 4 stack registers ST X,Y,Z,T. **ST L** is ***not*** stored. Doesn't affect the stack. Really useful in many situations. As **ST L** is not stored (nor replaced when executing RCLST afterwards), it can be used to pass a value of any type between the current and restored stacks.

RCLST *Trivial* *Most Useful*

Recall Stack: recalls a stored stack (ST X,Y,Z,T) which was previously stored with **STOST**. The register **ST L** is unaffected (see STOST above). If the stack wasn't previously stored, it does nothing.

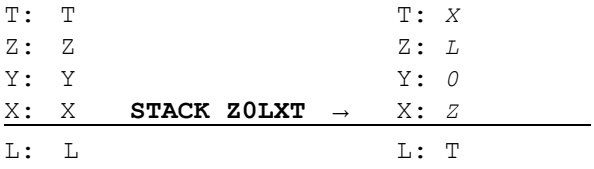
```
T:  T                T:  T prev
Z:  Z                Z:  Z prev
Y:  Y                Y:  Y prev
X:  X    RCLST   →   X:  X prev
L:  L                L:  L (unaffected)
```

RSEED *Trivial* *Very Useful*

Recall Seed: recalls the *current* seed for the built-in random number generator, so that a sequence of RANs can be continued exactly from where it was interrupted by using SEED with this value.

STACK *stack-arrangement arg* *Easy* *Most Useful*

Arranges the stack as specified by *arg*, which is exactly 5-char long (among X,Y,Z,T,L or a digit ***0-9***, duplicates allowed), representing the new contents of X,Y,Z,T,L . E.g. **STACK Z0LXT** would result in:



Other more useful examples would be, for instance (the possibilities are almost endless):

STACK XXXXL	fills up the 4 levels with the contents of X while leaving L unaffected,
STACK TZYXL	reverses the order of the stack registers's contents, L unaffected,
STACK 00000	clears the whole stack, including L (CLST *doesn't* clear L),
STACK 0019X	clears X and Y, initializes to 1 an increasing counter in Z, to 9 a decreasing counter in T, and stores the previous value of X in L,
STACK XZYTL	is equivalent to ST Y <> ST Z, any other exchanges/copies are possible

and so on. From the keyboard, the function shows a 5-char prompt which only admits X,Y,Z,T,L or ***0-9***.

TMBEG *Trivial* *Most Useful*

Timing Begins: marks internally the beginning of the timing. Doesn't affect the stack.

Rationale: Immensely useful for timing specific sections of code or whole programs, for instance to help in optimization, and also for documentation purposes.

TMEND *Trivial* *Most Useful*

Timing Ends: recalls to ST X the time elapsed since the most recent execution of TMBEG, in seconds and hundredths of a second. Doesn't reset the mark, so as to provide subsequent timings if desired.

TRY ... *steps* ... **CATCH** ... *steps* ... **ENDTRY** *Moderate+* *Very Useful*

Error Trapping and Recovery: executes the block of steps between **TRY** and **CATCH**, and:

- If none of them results in an error, upon reaching **CATCH** it skips the entire **CATCH** block and immediately resumes execution at the first step after **ENDTRY**.
- If one of them results in an error, execution immediately jumps to the first step after **CATCH** and the block of steps between **CATCH** and **ENDTRY** is executed. Within this block you can use the also proposed **ERRN** *(Error Number)* and **ERRS** *(Error Step)* instructions to find out which error took place and where, and take appropriate action. Upon reaching **ENDTRY,** execution simply continues at the first step after **ENDTRY**.
- **TRY/CATCH/ENDTRY** constructs can't be nested: if another error happens within the **CATCH** block the error is immediately reported to the user and program execution halts.

IF*test* ... *steps* ...**ELSE**... *steps* ... **ENDIF**, where *test* is one of **X**?**Y** or **X**?**0** (12 in all) *Moderate+* *Very Useful*

Structured Conditionals: if the text (X<Y?, say) comes out *True*, first it executes the steps between **IFX<Y** and **ELSE**, then resumes execution at the first step after **ENDIF**. If the test comes out *False*, then it executes the steps between **ELSE** and **ENDIF** instead. Saves **GTO**s and **LBL**s and looks cleaner.

7. Additional useful functionalities *(not rated)*:

1. The top two rows of keys should work as in the *HP-41C* to quickly enter addresses *01-10* for all instructions which take an NN address (`STO/RCL`, `STO+/−/…`, `GTO/XEQ`, etc) so that just pressing the first key of row 1 at the NN prompt will enter the address as *01*, etc. This appreciably speeds up using these instructions and is second-nature for *HP-41C* users, thus it's sorely missed in the *HP-42S* (and *Free42*).

2. ***Extended `STO`, `RCL`****, etc*: at the NN prompt, pressing the **E** (exponent) key should *lengthen* the prompt to **1 _ _** so that *R100* to *R199* can be addressed directly. Other lengthenings and ranges are possible.

3. Possibility of having the *"printer tape"* appear *next* to the simulated keyboard/display so that printed output can be seen simultaneously as it is produced. Extremely useful also for the new *Stack Trace Mode*.

4. Computing some kind of program ***checksum***, which would appear at line *00* next to the byte length (i.e.: `333 bytes, ck FA079`). To ensure that the checksum doesn't vary with internal representations of the program or machine status, it must be based *exclusively* in the **text** representation of the program (i.e.: the program text listing) so that *identical* listings are guaranteed to always result in *identical* checksums.

   This way, the checksum would be useful to be 100% certain that a program has been typed in correctly, unlike what happens with the *HP-35S*, where checksums are *utterly useless* because they take into account internal factors and thus the ones obtained by different users/machines usually *don't* match). Checksums may be computed/updated only when line *00* is visible, to avoid wasting time while entering/editing long programs.

5. Implementing ***Equations***, exactly as in the *HP-35S* (or a subset of the functionality), no need to reinvent the wheel. I've been using them very extensively in the *HP-35S* and I've realized that they're a very powerful and convenient way to greatly increase computing capabilities in programs, most especially as they allow for evaluating quite complex expressions *without* affecting the stack at all if so desired, while offering the possibility of storing and recalling intermediate values on the fly, including to/from the stack registers.

   In *Free42* they should be *pre-parsed* when entered/edited, so that they execute at full speed at run time, without further parsing, unlike the *HP-35S* where they're re-parsed every time they are executed in a running program, making them unbearably *slow*, to the point that they should be avoided within a loop because it will take *much* longer than if the computations were done as pure *RPN* code instead.

6. Better handling of *"**bignums**"*, i.e.: adding some functionalities to make life much easier when using numbers longer than 12 digits *("bignums"*, up to 34 digits in *Free42 Decimal*), for instance:

   - Right now, using `VIEW` with *bignums* shows only up to 12-digit mantissas (e.g.: `VIEW 00` would show something like `R00=2.30..21E18`). It should allow for viewing the whole 34 digits because else you're forced to stop the program and manually do something like `X<> 00, SHOW, X<>00, R/S`, which is quite cumbersome and can't be done within a running program which works with *bignums*.

   - It should be possible to input a *bignum* or recall it and see it whole in the stack regs (at least in `ST X`).
   - Possibility of printing whole a *bignum* in the printer, copy it to the *"clipboard"*, etc.
   - Possibility of `ARCL`-ing it whole to the `ALPHA` register.
   - Expand `FIX/SCI/ENG` modes to work with more than 12-digit mantissas (e.g.: `FIX 20` for *bignums*).

   Perhaps this last functionality (or all of them) can be implemented as a *new Mode*, for instance:

   `FIX/SCI/ENG/ALL/`**`BIG`** or Mode **`BIGN`** , which can be checked *On* or *Off*

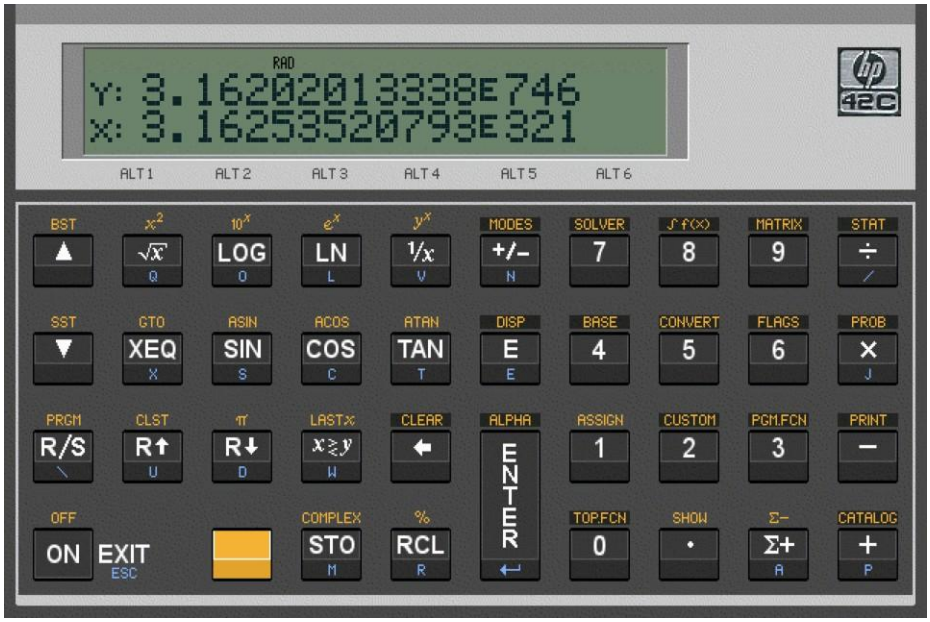
7. *(Speculative)* Implementing a kind of *"**compilation**"* of *RPN* programs to some faster-executing code. The *"compiled"* program wouldn't be editable, just *"runnable"*, and as at runtime it wouldn't need to perform some ancillary operations or checks that regular non-compiled programs need to do at every step, and all label searches would be pre-resolved, it might be significantly faster, similar to using a kind of *"System-RPN"*, akin to the existing *"SysRPL"* instruction set used with *RPL* programming.

   Admittedly speculative, but perhaps this isn't too difficult to implement (no conversion to *assembler* or anything of the sort) and would probably speed programs significantly. Some food for thought.

8. Conclusion

To wit, this is a worthwhile, carefully-selected set of new functions/operations which will very significantly augment *Free42* math and programming capabilities and general usability, even for real-life scientific or engineering uses on the go, at the workplace or at home.

As I see it, *Free42* really has the potential to be used profitably even compared with modern computing systems, and implementing the proposed functionalities is a step towards achieving that goal. I honestly feel most are not that hard at all to *implement* (though ergonomically *fitting* them in the interface will require careful consideration, I know.)



Free42 with a *Voyager Series*-inspired skin running on an *Android* device