HP35s - Storing Lotsa Lotsa Numbers

Valentín Albillo (HPCC #1075, PPC #4747)

Among many other excellent **HP35s** materials, the previous issue of ***Datafile*** (V26N4) included a routine by **Gene Wright** demonstrating how to pack *three* full-precision real numbers into a *single* variable thus making it possible to store and recall in excess of 2300 floating point values in **HP35s**' *indirect* variables. Gene stated that his program was intended as a starting point.

The idea caught my fancy and as soon as I got an **HP35s**, instead of trying to optimize Gene's routine I decided to try instead a *novel* approach to accomplish the same task by means of a new, *completely different* implementation to get myself familiarized with **HP35s**' programming. As you'll see, I partially succeeded but not quite: my resulting routine has both advantages *and* disadvantages when compared to Gene's but anyway I've had my fun creating it !

My version is *16 lines, 185 bytes long* (vs. Gene's *77 lines, 338 bytes*) and consist of three main, externally callable routines which implement ***Initialization***, ***Store***, and ***Recall*** functionalities within the address space of a single label, **P**:

STORE routine

```
P001 LBL P                                          { LN=185, CK=E7F3 }
P002 R↓
P003 XEQ P009
P004 ((J)-K*((J)*K)+K*REGT▷(J))*[0,0,0]+REGT        { LN=39, CK=E332 }
P005 RTN
```

RECALL routine

```
P006 XEQ P009
P007 K*(J)                                          { LN=5, CK=FEBA }
P008 RTN
```

Internal auxiliary subroutine

```
P009 (IP(I/3)▷J+(I-3*J-1▷K)+ABS([K*(K-1)/2,1-K*K,K*(K+1)/2]▷K))*0+REGT
P010 R↓
P011 RTN
```

INITIALIZATION routine

```
P012 [0▷J,0,0]▷(J)*(IP(REGX/3)▷J)                   { LN=28, CK=620D }
P013 STO (J)
P014 DSE J
P015 GTO P013
P016 GTO P010
```

Note: Line *P009's* Length & Checksum: *{ LN=65, CK=F717 }*

Notes:

- Lines *P004*, *P007*, *P009*, and *P012* hold *equations*, so you should press **EQN** prior to keying them in. Most include *store* operations (the "▷" symbol) which are entered by the **STO** key sequence, as well as *Indirect Address* **(J)**, which *must* be entered by pressing **RCL (J)** in **EQN** mode.
- All "*****" and "/" symbols are the multiply/divide operation, respectively.
- Unlike Gene's routine, *no flags are used*. Variables **I**, **J**, and **K** are used but variable **I** is unchanged, freely available to the user for indexing purposes.

Usage instructions

1. This program is to be run in **RPN** mode so make sure the correct mode is active. Also, keep flag **10** cleared so that equations are *evaluated*, not *displayed*.

2. Before using the routines for the first time, and just *this one time*, you must *initialize* the indirect variables which will host our *virtual variables* by specifying the number M of the highest virtual variable you want to use:

   M, XEQ P012 → M

   You can now use M+1 virtual variables numbered from 0 to M, where M is limited by the number of available indirect variables (beginning with indirect variable **0**), at the rate of one indirect variable per 3 virtual variables. Notice that unlike Gene's routine, virtual variable **0** *is* supported.

3. Once initialized, to store/recall a real (*not* complex, *not* vector) value into/from virtual variable N, you must proceed *in the exact same way as you do when working with standard numbered indirect variables*, i.e., you must address them indirectly by first storing their number in variable **I**, then performing the store or recall operation, which for numbered indirect variables is **STO(I)** or **RCL(I)** and for virtual variables is a call to the ***Store*** or ***Recall*** routine.

   This *essentially differs* from Gene's routine, which implemented the **RPL**-like convention of having the address of the virtual variable *in the X stack register* prior to performing the operation. I don't think this is convenient because:

   - it forces virtual variables to be used *differently* than standard indirect variables, so there's a new access method to learn, and worse
   - you can't easily *convert* a program using *indirect* variables to use *virtual* variables instead without major rewriting and even so, it might need extra care as having the number of the virtual variable in the stack essentially means *losing one stack register*, which can be troublesome in some programs and will always require time-consuming, careful scrutiny just to make sure no *side effects* are introduced by the *reduced* stack availability.

On the other hand, the fact that my routines use the *same* convention to address virtual variables as the one classic **RPN** uses to adress indirect variables, i.e.: taking their addresses from register **I**, means:

- you don't need to remember anything new: programs written to use indirect variables *can be converted* to use virtual variables instead by simply *replacing* **STO(I)** by **XEQ P001** and **RCL(I)** by **XEQ P006**, plus

- you can use standard *looping instructions* **ISG I** and **DSE I** to loop through virtual variables exactly as you normally do when working with indirect variables, no changes at all; and last but not least,

- you don't need to worry about *stack use*, which remains *unaltered* from that of your original program.

Thus, I've opted to adopt the classic **RPN** style instead of Gene's **RPL**ish one.

4. So, to **store** some value in virtual variable N (assuming its address (N) is already stored in index variable **I**), simply place your value in stack register **X** and call the ***Store*** (*P001 or simply P*) routine:

   value, XEQ P → value

   This operation works *exactly* like the built-in **STO** operation, leaving the stack absolutely *undisturbed*, including **LASTX**.

5. To **recall** the contents of virtual variable N (assuming its address (N) is already stored in index variable **I**), simply call the ***Recall*** (*P006*) routine:

   XEQ P006 → contents of virtual variable (N)

   Once again, this operation works *exactly* like the built-in **RCL** operation, recalling the contents of virtual variable N to the **X** stack register and lifting up the stack, again leaving **LASTX** undisturbed as well.

Examples of use

Let's perform some basic operations to demonstrate their use. First we'll *initialize* virtual variables 0 to 100 (assume **FIX 2**):

100, XEQ P012 → 100.00

Now, to use virtual variable **97** we store its address in index variable **I**:

97, STO I → 97.00

Let's put some contents in the stack (and **LASTX**) to check they're preserved:

5.55, ABS, 1.11, ENTER, 2.22, ENTER, 3.33, ENTER, 4.44

Now we'll store the value in the display (**4.44**) in virtual variable **97**:

XEQ P → 4.44

Let's check that the stack is indeed *undisturbed* after the store operation:

R↓ → 3.33, R↓ → 2.22, R↓ → 1.11, LASTX → 5.55

Let's recall the value from the virtual variable and store there **-3.42E-35**:

XEQ P006 → 4.44 *{ former contents of virtual v.97 }*

-3.42E-35, XEQ P → -3.42E-35 *{ new contents of virtual var. 97 }*

Now we'll store **2.71E85** in virtual variable **96**:

DSE I, 2.71E85, XEQ P → 2.71E85

Finally clear the stack and recall the contents of both virtual variables **96** and **97**:

CLSTK, XEQ P006 → 2.71E85 *{ contents of virtual variable 96 }*

ISG I, XEQ P006 → -3.42E-35 *{ contents of virtual variable 97 }*

The following test routine shows how to use the routines from a program: it generates and stores N values in virtual variables, computes their sum and stops. Upong continuing, the values are recalled and the sum is recomputed. Both sums must agree and you can time the routines: *{LN=65, CK=96CA}*

```
A001 LBL A          A008 RANDOM          A015 STO I
A002 STO A          A009 XEQ P001 {STO}  A016  0
A003 STO I          A010  +              A017 XEQ P006 {RCL}
A004 SEED           A011 DSE I           A018  +
A005 XEQ P012 {INIT} A012 GTO A008       A019 DSE I
A006 STOP           A013 STOP            A020 GTO A017
A007  0             A014 RCL A           A021 RTN
```

100, XEQ A → 100.00 *{ 1.5 seconds to initialize 101 virtual variab.}*

R/S → 53.85 *{ 175 seconds to STO & add 100 values = 1.75 s/v }*

R/S → 53.85 *{ 128 seconds to RCL & add 100 values = 1.28 s/v }*

The times for Gene's routines are 1.15 sec/value for **Store** and 0.62 sec/value for **Recall**, i.e.: significantly *faster* (34% and 52% respectively).

Conclusion

So that's why my version didn't *fully* meet my expectations despite being much *shorter* than G's: the **HP35s**' parsing/evaluation of equations is *so unexpectedly slow and inefficient* compared to **RPN** code that it ultimately hindered my efforts.

Still, its other advantages and reduced memory and resource consumption (no flags, for instance) may prove worthwhile, specially when *converting already existing programs* which use indirect variables to use virtual variables instead, such as the typical stack-tricky, multiprecision *p* - computing program.