

From The Editors – Issue 31

It is Spring and welcome to the new issue of *HP Solve*. It is new because it is a new issue, and it is new because *HP Solve* has a new direction. The *HP Solve* E-Newsletter is now dedicated to Teachers and STEM education. Advanced technical support for calculators will continue with at least one feature article in each issue to be included here.

Here is the content of this issue

S01 – Classroom Control at the Click of a Mouse HP advances its classroom teacher support with this advanced method of putting the teacher in charge and in control.

S02 – The Past, Present and Future of *HP Solve* by Richard J. Nelson, Jessica Cespedes, & Kevin Regardie. HP has been supporting its calculator users with a newsletter since September 1974. The history of *HP Solve* puts this support in historical perspective and it outlines what expected in the future..

S03 – Quadratics and Rocketry by your editor, Kevin Regardie. Explore the real world applications of the quadratic equation in preventing avalanches, plot a rocket trajectory, and safely launching fireworks. Lesson plans and teacher aids/answers are provided.

S04 – S.T.E.M. Education Moves full-“S.T.E.A.M.” Ahead! by Laura Berlins. STEM is a buz word in the Education World. Most readers know that it is for Science, Technology, Engineering, & Mathematics. Adding an A for the Arts is advocated in this article.

S05 – Navigating the ‘Common Core’ Maze by Kevin Regardie provides insights as to what Common Core State Standards mean.

S06 – STEM is DEAD; Long live STEMx by Jim Vandies who explains how the acronym STEM (Science, Technology, Engineering, and Mathematics) is no longer suitable to describe the focus of a technology driven education program.

S07 – HP Catalyst, Real-time assessment and applied Business Math by HP. HP launched the HP Catalyst Initiative in 2010 as described in this article.

S08 – Regular/assorted Columns

- ◆ From the editors.
- ◆ HP Calculator Tip – Use List Processing.
- ◆ What is RPN?
- ◆ Technical article on HP’s Randomness features. You won’t find this stuff in your Owner’s Manuals.

That is it for this issue. We hope you enjoy it. Write us with your ideas for future topics including being an author yourself at: hpsolve@hp.com

Kevin Regardie – STEM Editor.

Richard J. Nelson – Technical Editor.

HP Calculator Tip – Use List Processing

One of the many very powerful features of HP's scientific/graphing calculators is list processing. List Processing was first offered on the HP28C in June of 1986. Data (numeric or statistical) is entered into a list and the list processed by a wide range of functions. The list feature allows the data to be entered once. Copies may be made or stored for future use. You may then sum the list (most common use), sort the list, reverse the order of the list, calculate the difference between values, multiple each value in the list, etc.

The most common use of List Processing is adding a batch of numbers. Most users will key the value of each number, N values, followed by ENTER to place the values onto the stack. The + key is then pressed N-1 times to get their total. **The Calculator Tip is to train yourself to ALWAYS enter your numerical values into a list.** If the values are equal to the number of levels on the stack, or less, you may save time doing it the old fashioned way. If you have five or more values you should take the time to use List Processing. You will save time and increase the accuracy of your work because you may go back and review/edit your values without having to re-enter them.

What is RPN?

Many teachers may not be familiar with the HP calculator term RPN. RPN is an acronym for Reverse Polish Notation. Any term starting with Reverse isn't an attractive term so RPN students often start out with a negative bias. RPN is a calculator user interface that works with an automatic stack in such a way that an equal key is meaningless to solving problems. Normally/historically an RPN calculator has a double wide ENTER key e.g. HP 35s. In terms of mathematics logic RPN is called postfix. Normal algebraic logic is called infix. These terms describe the order of the data and the operands in calculator problem solving.

In the early days of calculators the explosion of technology continually lowered the cost of the internal electronics and the technical advantages of RPN became less of a cost advantage. Calculators converted infix problems to postfix problems internally for simpler problem solving logic. When other manufacturers joined HP in the scientific/financial calculator business they decided that algebraic logic better matched the solution of problems expressed with infix notation e.g. $1 + 2 \times 3 = ?$.

RPN⁽¹⁾ has many advantages but it is a different way of problem solving that doesn't use parentheses. It is a way of thinking that must be studied/understood. Learning RPN takes but a few minutes and it really makes problem solving much easier. The idea is simple. You enter the data and then you decide what to do with it. If you want to add you press the + key. In the early days there was a "calculator-logic-system war" between HP and other manufacturers. One T-shirt, for example, sported ENTER >=. RPN is simpler and faster. RPN was different and uniquely HP. Serious problem solvers saw the advantages and they have difficulty with algebraic calculators. Algebraic logic users simply can't use an RPN calculator.

There are two points that make the RPN "controversy" especially interesting. The first one is that one system is "better" than the other. Better is a subjective term and in problem solving there is only one solution, the correct one. Is faster "better"? Is fewer keystrokes "better"? Is easy error correction "better"? Remember that we are talking about solving mathematics problems. NO calculator solves problems exactly like they are written on paper. ALL calculators have user interface convenience features that make this statement true.

Let's examine the example problem given above: $1 + 2 \times 3 = ?$ This will illustrate the second less known point regarding RPN. A vital feature of algebraic calculators is that they follow a defined hierarchy of

operation order. If you try this example and you press the corresponding keys on various calculators proceeding left to right you will discover that you could get two answers: 9 or 7. Why is this? Some calculators have no logic and simply perform the operations as they are keyed. Machines that return 9 as an answer might be called arithmetic logic, i.e. no logic. The machines that return 7 as an answer follow an algebraic logic in that multiplication is performed before addition. What was the problem? We all agree that there can only be one correct answer.

Perhaps it is the way the problem is given. It could be $(1 + 2) \times 3 = ?$ OR it could be $1 + (2 \times 3) = ?$. This is where parentheses play an important role in mathematics. Calculators that are truly algebraic will return 7. Algebraic logic Calculators always use parenthesis, but not all calculators that use parentheses are algebraic (using the hierarchy of operations). Calculators that use RPN cannot solve the problem without the parentheses. Since there are no parentheses on an RPN (only) calculator the problem without parentheses cannot be solved because it is unclear. Many calculator users won't even think about this.

Thus far three of the common four calculator user interfaces has been discussed – Arithmetic (no logic), RPN (postfix logic), and algebraic (infix logic). There is a fourth logic system that most readers use on their computers every day. It is called command line logic. The idea is that the user types in the problem with data, operators, and parentheses pretty much as they find it in their text books. The machine parses the command line and if the input follows all of the rules it returns an answer. If not it returns an error. All graphing calculators use Command Line logic.

Here is one more interesting factoid regarding HP calculators. HP has always made calculators of all four user logic interfaces and no other calculator manufacturer does that. In fact many of HP's current models give the user a choice of two or even three user interfaces.

Notes: From the Editors

(1) For a more technical explanation of RPN see [HP Solve](#) issue #4 page 3.

The Randomness of HP

Richard J. Nelson & Namir Shammam

Introduction

Randomness is a term we have all heard because it is frequently used in normal life and mathematics. We flip a coin to “let fate” make a decision. We draw numbered balls from a mixed up container for Bingo and Lotto drawings. We finger snap a spinner, shuffle a deck of cards, or roll a pair of dice for choices to be made for board games. The list is long. Exactly what is meant by, and what are the important qualities of an event or number being truly random? How may we get our own random numbers? This seven page article is supported by two appendices that provide technical details and linked resources.

Being Random

Technically randomness means different things in various fields. When used as described above it means a lack of a pattern and an equal chance of each possible outcome. It means that a series of random events occur in such a way that the next event is not predictable. Depending on how the random event is caused, however, there are probabilities that a certain value will occur. For example, if we flip a properly made coin⁽¹⁾ most people will agree that when the coin lands there will be an equal chance of it showing heads or tails. It is not the purpose of this article to delve into the very deep mathematical techniques of defining and evaluating all aspects of randomness. Rather it is the purpose of this article to explore the seldom documented random features of HP calculators. Just like other features such as SOLVE, MOD, and a Calendar, HP has implemented random features that are exceptional and mathematically rigorous. The ability to produce high quality random options suitable for all of the situations described in the Introduction is a typical hall mark of HP calculators.

Sources of Randomness

There are two sources of randomness. (1) A physical process such as rolling die or, (2) a mathematical operation such as that implemented in a computer or HP calculator. In its most basic form a series of numbers are generated mathematically by various processes such as those previously mentioned. When computers are involved the processing of numbers is very fast and vast quantities of data may be used for a particular application. In certain modeling situations billions of random numbers may be required and that is beyond the capability of our hand held calculators. The discussion of random numbers is a vast subject and the available articles, books, programs and research papers would keep any teacher or student busy for many years. When vast numbers of high quality true random numbers are required computers are teamed up with electronic devices such as a noisy resistor, noisy semiconductor, atmospheric noise, or detecting a radioactive decay process⁽²⁾.

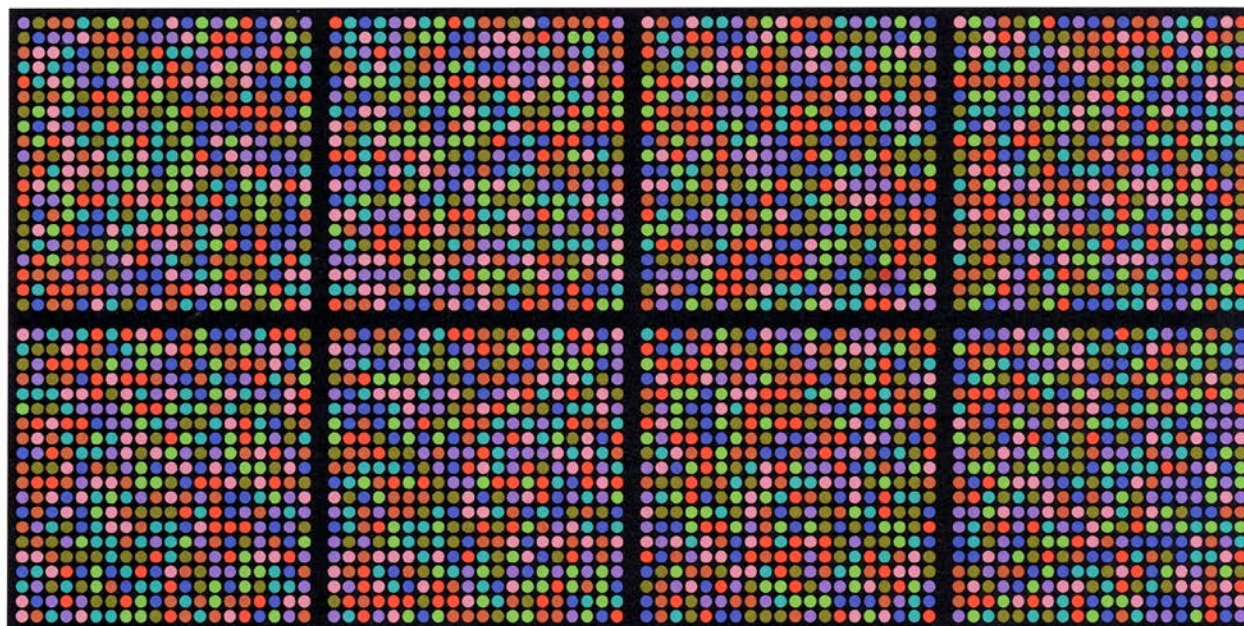
Randomness Testing

Testing for true randomness is a challenging task⁽³⁾ and is usually a college level topic that will not be covered here. One of the simplest tests is a tally of all the numbers – usually digits – and noting their frequency. For true randomness the distribution of the digits, and their odd-evenness should be uniform⁽⁴⁾. There are dozens of (statistical) tests that may be performed and looking for repeating sequences is but one of them. While these tests involve advanced mathematics and a computer there is one simple test that a graphing calculator may easily perform to test for randomness. Use the generated numbers two at a time as x-y coordinate values, scale them, and plot a point on the calculator screen. The screen should look similar to Fig. 1 which shows a 131 x 80 pixel plot (HP50g display) of random points. It is a simple task to examine the screen pattern as having no pattern.



Fig. 1 – 10,480 random pairs plotted.
<http://www.random.org/>

If the numbers are not random vertical, horizontal, or diagonal streaks or other patterns will be obvious. Fig. 2 is from the first link in note 2. It illustrates a more complex graphical way of comparing a batch of 1,200 random numbers. Adding a third number to represent color greatly increases the information that may be derived in terms of the desired lack of patterns.



American Scientist, July-August 2001, Volume 89, p300

Figure 2. Eight specimens of randomness come from very different sources but yield similar patterns. The 400 colored dots in each panel represent 1,200 random bits taken three at a time. The sources of randomness are, from left to right and top to bottom: a pseudo-random generator, the 1955 Rand Corporation table, the HG202 random-bit generator (faust.irb.hr/~stippy/), the Pennsylvania Daily Number lottery (www.palottery.com), 1,200 coin flips, rapid pounding on a keyboard, an electrocardiogram of atrial fibrillation (www.physionet.org), and six Lava Lites (lavarand.sgi.com).

Random or Pseudorandom?

Using a computer or calculator program to generate random numbers is by definition not considered possible because the same numbers may be generated by another machine using the same program and therefore is predictable. A fundamental part of being random is the inability to predict the next values in a series. It is possible to program a very well tested⁽³⁾ algorithm that will produce a very long series of random numbers. The “trick” is to jump into the series at some known point - a value which may be kept secret or itself being random. These programs are called pseudorandom number generators (PRNG) compared to true random number generators⁽³⁾ (TRNG) based on a physical process. All HP calculators use a PRNG. The point where you start the series is called the seed. See additional details in the next section and Appendix A. Fig. 3 shows examples of TRNG’s. Opposite side spot counts equal seven.



Fig. 3 – L to R Conventional dice, rounded corners in various colors, game dice with special markings, and assorted non-standard sided dice which are often 8, 10, 16, and 20 sided for various applications of a TRNG.

HP Calculator PRNGs

Table 1 shows the current calculators offered by HP. Models that feature randomness are indicated by *.

Circa late 1980's and with the HP 28C on June 1st 1986, HP revised their PRNG algorithm to improve it. All current machines will have the values shown in table 2 using the latest version of the algorithm except for the remanufactured HP-15C which uses a different (earlier) algorithm.

As an illustration of the limited information related to this algorithm and its application, the text of the HP33s "user's manual" is reproduced below. This example is as complete as it gets in terms of the descriptions of the two randomness functions typically called RAND and SEED. The indication of its quality is given with its Donald Knuth reference.

Table 1 – 2013 HP Calculator Products – 24 Total

| # | Financial Calculators | Scientific & Graphing | Home & Office |
|----|---|-----------------------------------|--------------------------|
| 1 | HP 10bII | HP 10s | HP CalcPad 100 |
| 2 | HP 10bII+* | HP-15C Limited Edition* | HP CalcPad 200 |
| 3 | <u>HP 20b</u> * | HP 33s* | OfficeCalc 100 |
| 4 | <u>HP 30b</u> * | HP 35s* | OfficeCalc 200 |
| 5 | HP-12C | HP 39gs* | OfficeCalc 300 |
| 6 | HP-12C 30th Anniversary | HP 39gII* | PrintCalc 100 |
| 7 | HP-17bII+ | HP 40gs* | QuickCalc |
| 8 | | HP 48gII* | |
| 9 | | HP 50g* | |
| 10 | | SmartCalc HP 300s | |
| | 5 of 7 = 71% programmable | 8 of 10 = 80% programmable | None programmable |

Notes: Programmable calculators are in blue. Underlined models are at or near end of life.

** Machines that have a PRNG function.*

Table 2 – Random Sequences for Current Models (Seed = $\sqrt{5}$)

| # | Machine | Random #1 | Random #2 | Random #3 |
|----|----------------------------------|--------------------|--------------------------------|---------------------|
| 1 | HP 17bII+ ^(a) | 0.521548989463 | 0.0593946804209 | 0.666602695109 |
| 2 | HP 10bII+ ^(a) | 0.521548989463 | 0.0593946804209 | 0.666602695109 |
| | HP 20b ^(b) | 0.521548989463 | 0.0593946804209 | 0.666602695109 |
| 3 | HP30b ^(a) | 0.521548989463 | 0.0593946804209 | 0.666602695109 |
| 4 | HP-15C LE^(a,c) | 0.635762643 | 0.5681838663 | 0.6749247476 |
| 5 | HP33s | 0.521548989463 | 0.0593946804210 ^(d) | 0.666602695109 |
| 6 | HP35s | 0.521548989463 | 0.0593946804210 ^(d) | 0.666602695109 |
| 7 | HP39gs | 0.521548989463 | 0.0593946804209 | 0.666602695109 |
| 8 | HP39gII | 0.521548989463 | 0.0593946804209 | 0.666602695109 |
| 9 | HP40gs | 0.521548989463 | 0.0593946804209 | 0.666602695109 |
| 10 | HP48gII | 0.521548989463 | 0.0593946804209 | 0.666602695109 |
| | HP50g | 0.521548989463 | 0.0593946804209 | 0.666602695109 |

Notes: (a) The seed is stored using the keys STO RAND.


(b) The HP 20b does not have the ability for the user to store a seed.

(c) Based on original ROMs of HP-15C announced on July 1st 1982.


(d) Last digit error because of a bug in the implementation of the algorithm wherein the rounded⁽⁷⁾ value was used (as with most normal functions) instead of the truncated value.

The HP33s Calculator Random number usage instructions below will serve as an example of the brevity of the typical RAND information.

Seed

To store the number in **x** as a new seed for the random number generator, press  **SEED**.

Random number generator

To generate a random number in the range $0 \leq x < 1$, press  **RAND**. (The number is part of a uniformly-distributed pseudo-random number sequence. It passes the spectral test of D. Knuth, *The Art of Computer Programming*, vol. 2, *Seminumerical Algorithms*, London: Addison Wesley, 1981.)

The RANDOM function uses a seed to generate a random number. Each random number generated becomes the seed for the next random number. Therefore, a sequence of random numbers can be repeated by starting with the same seed. You can store a new seed with the SEED function. If memory is cleared, the seed is reset to zero. A seed of zero will result in the calculator generating its own seed.

For most applications, especially documented ones, it is most desirable to start with a specified seed for testing and so others may repeat and validate your results. Additional details of the current models RAND function may be found in Appendix A.

PRNG and TRNG Applications

The more obvious calculator applications of RAND (as it is usually notated on an HP calculator keyboard) are described in the Introduction. Selected example RAND programs for various HP “languages” e.g. Classical RPN: HP-15C, HP35s; ENTRY RPN: HP30b; RPL: HP48gII, HP50g; and Pascal like: HP39gII may be found in Appendix B. Cryptology requires the highest quality random numbers and a TRNG is essential. Bank and Internet transactions are examples. Simulations and research applications that use Monte Carlo⁽⁵⁾ methods require billions of high quality random numbers. In certain simulations a PRNG may be suitable.

Scaling

HP Calculators produce pseudorandom numbers from 0 to 0.999 999 999 999. If you want other values such as 6 for a die or 52 (both integers) for a deck of cards you must scale the 0.nnn nnn nnn nnn PRNG output range to another range. The general formula to use is:

$$\text{rand integer } (x,y) = (\text{round}(\text{rand}() * (y-x))) + x$$

(x represents the minimum and y the maximum integer value).

An example is 1 to 6 for a standard die.

$$\text{Random } 0,0.\text{nnn nnn nnn nnn} = \text{ROUND}(\text{RAND} * (6-0) + 1) = \text{ROUND } 0.\text{nnn nnn nnn nnn}$$

An RPL program to do this is: `<< RAND 6 * CEIL >>` where CEIL returns the smallest integer greater than or equal to the argument. Each execution returns a random digit from 1 to 6. Using $\sqrt{5}$ the result of 102 throws of the die are: 1=17, 2=21, 3=10, 4=15, 5=20, 6=19. A uniform distribution would be 17 of each.

Non-uniform Rand distributions

The output of most random number generators has a uniform distribution but there are applications that require that the numbers be distributed non-uniformly⁽⁶⁾. The basic uniformly-distributed random number generators play a vital role in generating non-uniformly distributed random numbers. Mapping the (0,1] range of a basic uniform random number generator to any other range (A,B] is very easy and uses the following equation:

$$X = A + (B-A) U$$

The notation (x,y) signifies a range that includes the value of x but very closely approaches y. The variable U is the uniform random number in the range (0,1). The range of 0 to 1 is useful in generating non-uniformly distributed random numbers by applying that range to cumulative distribution functions.

Since any cumulative distribution function has values that range from 0 to 1, we generate a random number in that range and then calculate the value in a non-uniform distribution that has the matching cumulative value. Here is a simple example that illustrates the concept. Consider the exponential probability distribution function (PDF):

$$f(x) = \lambda e^{-\lambda x} \text{ for } x > 0 \text{ and } \lambda > 0$$

The cumulative distribution function (CDF) $F(x)$ is:

$$F(x) = 1 - e^{-\lambda x}$$

To generate an exponentially distributed random number we use the inverse of the CDF:

$$x = F^{-1}(U) = \ln(1/U) / \lambda$$

This method is called the *inversion method*. It is a popular one, but not the only method. Table 3 shows a set of probability distributions to which we can easily apply the inversion method.

Table 3 – Examples of Simple Probability Distributions Where the Inversion Method Can Be Easily Applied

| Probability Distribution | Probability Distribution Function | Cumulative Distribution Function | Inverse CDF |
|---------------------------------|--|---|--------------------|
| Weibull | $a x^{a-1} \exp(-x^a); x > 0, a > 0$ | $1 - \exp(-x^a)$ | $(\ln(1/U))^{1/a}$ |
| Logistic | $1/(2+e^x+e^{-x})$ | $1/(1+e^{-x})$ | $\ln(U/(1-U))$ |
| Cauchy | $1/(\pi(1+x^2))$ | $1/2 + 1/\pi \arctan(x)$ | $\tan(\pi U)$ |
| Pareto | a/x^{a+1} for $a > 0$ and $x > 1$ | $1 - 1/x^a$ | $1/U^{1/a}$ |

Unfortunately Table 3 shows the minority of probability distributions to which the inversion method can be easily applied. Other distributions, like the normal distribution, Student-t, Chi-Square, and the Fisher F distributions, all require extensive and/or iterative calculations to use the inversion method. For example,

The PDF for the normal distribution is:

$$\text{PDF}(x) = \exp(-(x-\mu)^2/2\sigma^2)/(\sigma\sqrt{2\pi})$$

The CDF for the normal distribution is:

$$\text{CDF}(x) = [1 + \text{erf}((x-\mu) / \sqrt{2\sigma^2})]/2$$

Where $\text{erf}(x)$ is the error function. Calculating the values for this function typically involves a summation series. Solving for x in the above equation, given a uniform random number (as the value for $\text{CDF}(x)$) requires an iterative root-seeking process in which each iteration has to calculate one or more values for the error function.

In the case of a simple normal distribution (with a mean of 0 and a standard deviation of 1), the PDF is:

$$\text{PDF}(x) = \exp(-x^2/2)/\sqrt{2\pi}$$

Statisticians have developed a much simpler and non-iterative algorithm to generate standard normally distributed random numbers. The method, which generates pairs of random numbers, uses the following algorithm:

1. Generate uniform random numbers U_1 and U_2 .
2. Calculate $\text{Rand1} = \sqrt{\ln(1/U_1)} \cos(2\pi U_2)$
3. Calculate $\text{Rand2} = \sqrt{\ln(1/U_1)} \sin(2\pi U_2)$

You can store the value of Rand2 and use it when you want the next normally distributed number. Alternately, you can obtain a single random number using the expression (Rand1 + Rand2) mod 1. The book *Computation Methods in Statistics and Econometrics*, by Hishashi Tanizaki offers excellent short and easy to read FORTRAN subroutines that generate non-uniform random numbers. Tanizaki uses a very clever approach by generating non-uniform random numbers based on the values of other, simpler-to-calculate, non-uniform random numbers. Thus, the author avoids complicated and CPU-intense calculations.

Another approach uses the following rational polynomial approximation to calculate x, the normally distributed random number:

$$x = t - (c_0 + c_1 t + c_2 t^2) / (1 + d_1 t + d_2 t^2 + d_3 t^3)$$

Where t is defined as:

$$t = \sqrt{\ln(1/U^2)}$$

And the constants are $c_0= 2.515517$, $c_1= 0.802853$, $c_2= 0.010328$, $d_1=1.432788$, $d_2=0.189269$, and $d_3=0.001308$.

Once you have the standard normal random number x, you can map it onto a general normal distribution with a mean μ and standard deviation σ using the following equation to get y:

$$y = (\sigma x + \mu)$$

Statisticians have created similar approximations for the Student-t, Chi-Square, and Fisher F distributions.

It is worth mentioning that the HP 39gII implements functions that generate non-uniformly distributed random number for the normal, binomial, Chi-Square, Student-t, Poisson, and Fisher F distributions. Tables 4 shows these functions with sample calls. The last argument in each sample call represents a uniformly distributed random number in the range of (0,1]. These functions make generating random numbers for the above distribution a breeze!

Table 4. The list of functions in the HP 39gII that can generate popular non-uniform random numbers

| Distribution | Example of Function Call |
|---------------------|---|
| Normal | normald_icdf(0, 1, 0.841344746069) returns 1 |
| Binomial | binomial_icdf(4, 0.5, 0.6875) returns 2 |
| Chi-Square | chisquare_icdf(2, 0.952641075609) returns 6.1 |
| Fisher F | fisher_icdf(5, 5, 0.76748868087) returns 2 |
| Poisson | poisson_icdf(4, 0.238103305554) returns 2 |
| Student-t | student_icdf(3, 0.0246659214813) returns 3.2 |

Observations and Conclusion

The use of randomness is extensive in our high tech society. Randomness is used for bingo games, lotto drawings, coin flipping, dice rolling (for board games) and gambling in places like Las Vegas. There are two sources of randomness. (1) A physical process such as rolling dice, or (2) a mathematical operation, such as that implemented on a computer or HP calculator. Many HP calculators provide randomness features in the form of a pseudorandom number generator, PRNG, – vs. a true random generator, TRNG, such as the throwing of dice - and a SEED function. Unfortunately most calculator manuals do not provide very much information on the RAND and SEED/RDZ randomness functions and it is the purpose of this article to expand on this information. All HP calculators with these features made since the late

1980's use the same algorithm as detailed in Appendix A. HP's PRND will generate more than enough quality random numbers for any calculator program you can run in your lifetime.

Notes for: The Randomness of HP

- (1). *The primary assumption is that the coin is homogenous, uniform, throughout. Another assumption is that the coin is symmetrical about a center slice through the coin between the head and the tail sides. This is not true in reality but the raised portions of each side are quite close. Yet another assumption is that the nervous system of a human being is not consistently repeatable in terms of the force and direction of the thumb and fingers when making the flip and the distance traveled while turning. For an additional resource see: <http://en.wikipedia.org/wiki/Randomness>*
- (2). *For a very readable overview of randomness see an article titled Randomness As A Resource by Brian Hayes in American Scientist, July-August 2001, Volume 89, p300. See and download the article at: <http://www.americanscientist.org/issues/pub/randomness-as-a-resource> For an excellent source of true random numbers (based on atmospheric noise) and other randomness information see: <http://www.random.org/> If you only visit one randomness website this should be the one.*
- (3). *The critical issue here is being well tested and documented by multiple team efforts using the best testing tools available. The constants used in the algorithms are critical and once a good algorithm is developed it should not be altered because the results will no longer be valid. HP uses a good quality low output algorithm. Documentation (especially in the User's Manual) is sparse and this article will provide additional information and resources. See <http://www.random.org/> Especially see: <http://www.random.org/analysis/> Additional test descriptions may be found at: http://en.wikipedia.org/wiki/Diehard_tests Also see: <http://csrc.nist.gov/groups/ST/toolkit/rng/index.html>*
- (4). *A discussion of using a computer to download 10,000 digits of π and counting the distribution of the digits is described and illustrated in **HP Solve** # 25 page 69, Table 5. Each digit is very well distributed with an average of 1,000 occurrences of each digit 1 through 0. The Odd/Even distribution is not quite so uniform with occurrences being 1004/994. To read the article See: http://h20331.www2.hp.com/hpsub/downloads/HP_Calculator_eNL_09_September_2011.pdf*
- (5). *An outline of the Monte Carlo process may be found in the Introduction at: http://en.wikipedia.org/wiki/Monte_Carlo_method.*
- (6). *For an excellent discussion of how some of these distributions may be implemented on a calculator see an HHC 2011 presentation by Richard Schwartz. The Power Point version is titled To Deviate Normally (714 KB). The Conference paper, 11 pp, is titled Generating Normal Deviates (145 KB). The HHC proceedings may be obtained at: <http://www.pahhc.org/ppccdrom.htm> You may also request copies from Richard J. Nelson: rjnelsoncf@cox.net*
- (7) *Bug Hunter Joseph K. Horn explains. "Bottom Line: HP 33s and 35s programs that use random numbers cannot be relied upon to obtain precisely the same results as the other HP models that use the same RNG engine. This is critically important if the RNG is re-seeded with one of these different random numbers; in that case, the random number sequence in the 33s and 35s will diverge from the other models."*

Example of critical divergence: 11 RDZ RAND RDZ RAND RAND RAND --> (Note: RDZ is used to store the seed)
RPL models: 0.975035362027, 0.529463266203, 0.783522353434
33s & 35s: 0.826166290494, 0.626934236292, 0.213277976998
As you can see, the obtained sequence is totally different.

Appendix A – HP’s Pseudorandom Number Generator

HP 48 Random Number Generator

Richard J. Nelson

The following discussion of the HP 48 Pseudorandom number generator is based on postings made by John H. Meyers on May 28, 1997 to the Newsgroups: comp.sys.hp48^(A1). John disassembled the ROM code to better understand how the algorithm works. **This PRNG is used on all HP calculators since the introduction of the RPL machines in the late 1980’s.** See Table 2 in the article text and Note (7) above.

The process used by the HP 48 to generate “random” numbers is one that has been tested by some of the best minds in the field of mathematics and it is well known for its mathematical properties of randomness. These programs are known as pseudorandom number generators and they have many uses in games, simulations, and modeling. An important aspect of pseudorandom number generators is being able to *repeat* the sequence of random numbers.

Most random number generators start with a number called the seed and apply a mathematical process to the seed value. The result is the generator output which also serves as the next value for the seed. Each random number is the result of the previous number. If 100 numbers are generated with a given seed the same sequence will be generated if the same seed is used again at the start. This aspect of random number generators is important for experiments, which must be repeated by others for verification.

A major concern is the non-repeating length of the random number sequence — cycle length. If the same value of the initial seed is used by the random number generator, the sequence will repeat. The non-repeating sequence length must be many times longer than the number of RNs needed to complete the task at hand. The HP 48 random number generator has a sequence length of 50,000,000,000,000 (5E13).

The following program generates 200 random numbers on the stack in one second. << 1 200 START RAND NEXT >>. This program will run on a fast (3.7 Mhz.) HP 48 for more than 7,922 years before the cycle repeats. This is certainly more than adequate for any HP handheld calculator program. Today’s much faster models (HP38gII) are 20 times faster and you will only have 396 years of continuous non-repeating random numbers.

The second command related to random numbers on the HP 48 is RDZ. This command accepts the level one value and stores it as the RAND seed. A common seed is generated by taking the reciprocal of ‘e’ with the key sequence: 1 e^x 1/x. Add these three commands at the beginning of the program for repeatable results. The seed used for testing the machines in Table 2 of the current machines is $\sqrt{5} = 2.2360679775$. (2.23606797749979)

If the machine is powered up for the first time or you use zero as the seed the system sets the seed value to 999,500,333,083,533. The HP 48 uses 15 digit arithmetic internally and truncates the answer to 12 digits. If you are interested in random number generators here are four good sources.

1. “Semi-Numerical Algorithms”, Volume 1, The Art of Computer Programming, Addison Wesley, 1969, by Donald E. Knuth. See Volume 2 page 9.
2. Numerical Recipes in C by Press; Teukolsky, Vetterling, & Flamery; Cambridge University Press. See Chapter seven.
3. HP 48 Goodies Disk No. 9 (#GD9) See Postings Directory, text file RAND.DOC.
4. PPC ROM User’s Manual . See RN routine on page 380 and GN routine on page 176.

Technically the HP random number generator is described as a multiplicative linear congruential generator. The equation (used by the HP 28C and all following models) is:

$$X_{n+1} = (a * X_n + c) \pmod m \quad \text{Where:}$$

- X_{n+1} = next random number
- $a = 2,851,130,928,467.$
- X_n = seed or previous random number.
- $c = 0$
- mod = modulo function.
- $m = 1E15.$

Choosing a seed

The more digits you use for the seed the better. Rather than keying in a 12 digit number it is more convenient to use a function to generate the digits. A second consideration for the longest cycle is a seed that is *not* divisible by two (it should be odd) or five. The output is *not* “ruined” but it will not strictly conform to the randomness tests if it is not an “odd” seed for a period of 5E13 instead of 5E15. One way to verify that these conditions are met is to test the number. ‘**TS1**’, **T**est **S**eed version **1** tests an integer number not to be divisible by 2 or 5. It returns a one if the number is a “good” number.

‘**TS1**’ << DUP 2 MOD SWAP 5 MOD AND >>

27.5 Bytes, #EA6h. Timing: 123456789012 ⇒ 0 in 9.01_ms., 123456789011 ⇒ 1 in 8.89_ms.

Here is how ‘**TS1**’ works. ‘**TS1**’ is similar to the built-in tests in that it returns a one for pass, or a zero for fail. If the input integer is NOT divisible by 2 or 5 it is a “good” number and the result is a one. A copy of the input number, n, is made with DUP. If n is evenly divisible by 2 the result of 2 MOD is 0, otherwise it is 1. The result of the first MOD “test” is Swapped with n and a similar “test” is made with 5 MOD. A zero results if n is evenly divisible by five and one through four otherwise. At this point level two may be either zero or one (non-zero) and level one may be either zero or non-zero. The logic operator AND compares level two with level one according to the “rules” in the table below.

AND Truth Table

| Input | Input | Output |
|---------|---------|--------|
| Level 2 | Level 1 | AND |
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

The one in the truth table is any non-zero real according to the rules defined in the Advanced User’s Reference Manual, **AUR**. ‘**TS1**’ is great for integers, but fails for decimal numbers. 0.88 is divisible by two yet it passes the ‘**TS1**’ test. The reason for this is MOD only works for integers. Let’s solve this problem by converting any number, decimal or mixed, into an integer with ‘**TS2**’.

‘**TS2**’ << ABS MANT DUP →STR 3 OVER SIZE SUB SIZE ALOG * >>

37.5 Bytes, #58C0h. Timing: 0.123456789012 ⇒1234567890012 in 24.0_ms.

Here is how ‘**TS2**’ works. ABS insures that only positive numbers are used. Negative numbers may result from some math functions. MANT returns the mantissa of the number in the form N.N ...N. The test input 0.123456789012 becomes 1.23456789012. The goal is to determine how many places, digits, there are after the decimal point. A copy is made with DUP and the level one number is made a string with →STR. The first position in the string is followed by a decimal point. That means that we want the third through the total, SIZE, number of characters and OVER makes a copy of the string, SIZE gives the

total, and SUB returns the desired characters. Another SIZE provides their number and ALOG makes a power of ten equal to the SIZE value. The last command multiplies the MANT number by this number effectively moving the decimal point to the far right converting the decimal number into an integer.

Any input to 'TS2' is converted into an integer. 'TS1' tests it for meeting the non-divisible by 2 or 5 rule. The last task is to convert a "good" number back to a decimal number to use as a seed. 'TS3' performs this task. Any number may be used as a seed. It does not have to be a decimal number. Let's explore the inverse of 'TS2' for "educational/programming" purposes.

'TS3' << MANT 10 / >> 25.5 Bytes, #4DEAh. 123456789011 \Rightarrow 0.123456789011 in 6.5_ms.

Does RAND generate numbers that are not recommended as *initial* seeds? We now have the tools to test numbers so let's test the RAND output just for fun.

'TS4' << DO RAND DUP 'SEED' STO TS2 1200 .05 BEEP UNTIL TS1 END >>

74.0 Bytes, #32beh. NOTE: Commands not native to the HP 48 are in **bold**.

'TS4' has great entertainment value. It produces a 'chirp' for each RAND number generated. Our 'TS1' test is applied and if it is a good seed the program ends with the RAND value stored in a variable called 'SEED'. The program generates another random number if the first one is not acceptable. It will continue to 'chirp' until a good one is found. I have heard (actually the program was modified to count them) eleven consecutive 'chirps' on one occasion. The chirps is NOT an indication of a PRNG failure.

Here is how 'TS4' works. A DO ...UNTIL ... END loop structure is used. The DO clause is the commands between DO and UNTIL. The commands between UNTIL and END is the test clause. The first command in the DO clause is RAND. A copy is made with DUP with one of the RAND numbers stored in 'SEED'. TS2 converts the decimal number to an integer and a 1,200 hertz tone of 0.05 seconds duration is sounded with BEEP. The UNTIL clause is simply our test program TS1. It returns a one if the number is acceptable. If this is the case the END is executed and the program is finished. If TS1 produces a zero, the DO clause is executed again. The DO loop repeats the DO clause until the UNTIL clause tests true.

I mentioned that I "counted" the number of 'chirps'. This is done with 'TS5' with the addition of the three underlined commands at the beginning. The operation of the program is similar to 'TS4'.

'TS5' << 0 DO 1 + RAND DUP 'SEED' STO TS2 1200 .05 BEEP UNTIL TS1 END >>

81.5 Bytes, #145Ch. NOTE: Commands not native to the HP 48 are in **bold**.

You may leave out 0, 1 + 1200 .05 and BEEP and use the modified program,

'TS6', as a create-a-seed-and-record-it program. Add an RDZ command after the END if you call this program as part of your main program.

'TS6' << DO RAND DUP 'SEED' STO TS2 UNTIL TS1 END (RDZ optional) >>

50.5 Bytes, #E57Fh. NOTE: Commands not native to the HP 48 are in **bold**.

It should be kept in mind that the basis for RAND is 15 digits internally and that the 12 digits we see are

truncated from the internal 15 and that they are NOT necessarily an accurate indication that the PRNG is not working properly. These test programs are intended as educational programming/test examples.

We now have the basic tools to create and store a seed for repeatability purposes. The above programs illustrate the modularity nature of the HP 48 programming language. It is easy to write and test short small modules and call them as needed. This also makes modifications easy. You may use a module just as you do a built-in command as illustrated above. Of course, you may key in the respective code into the program to make it a complete stand-alone program if you wish. The interactive stack makes assembling programs this way easy, and the amount of re-keying is kept to a minimum. These programs are provided as an educational exercise for those readers wanting to explore programming and RAND.

After many users have researched the performance of the HP random number generator it is not surprising to find that HP has done an excellent job in the programming of the PRNG. **The ONLY requirement the user has for using RDZ and selecting a seed is that it be as many decimal digits as possible.** All other requirements are automatically taken care of by the algorithm. ANY 12 digit number will suffice with an exponent up to 99 maximum. The not divisible by 2 or 5 is taken care of by internally shifting the exponent so it always ends in one. e.g. 362 becomes 621, 539 becomes 391, 045 becomes 451. Of course you cannot know this unless you examine the internal ROM programming.

Another aspect of the HP 48 RNG described by John H. Meyers is the details of storing zero as a seed. The HP 48G Series User's Guide mentions that 0 RDZ sets the seed according to the system time. The TICKS counter uses five nibbles as a real time counter and the over flow is "counted" to an eight nibble register. The rightmost five HEX digits are taken from the real time clock with the leftmost eight digits taken from memory. The sequence, **12 →TIME 0 RDZ RAND**, however, will not produce consistent results. The reasons for this are unclear. The intentional system clock jitter^(A2) is likely the major cause. If you run the program 100 times putting the results into a list and sorting them you will find that there are a limited number of variations - usually less than a dozen with the majority of them being one or two values.

Notes for The Randomness of HP Appendix A

(A1) See <http://www.hpcalc.org/search.php?query=John+Meyers+random+Numbers> for additional details.

(A2) *The HP48 system clock is not a smooth running clock. In order to ensure that the calculator's digital signals do not cause radio frequency interference the system clock has a small amount of jitter introduced. This jitter causes a small ambiguity in precise time measurements based on it and the resolution of TICKS (8192 TICKS per second, 122 microseconds) is enough to detect it.*

Appendix B – Selected PRNG Programs

These programs illustrate the flipping of a coin, rolling a pair of dice, dealing a deck of cards (selection without replacement) the use of scaling, and other assorted RAND applications.

Selected Random Number Generators for the HP-41 (Classical RPN^{B1})

Also see these at: Jean-Marc Baillard's excellent website at <http://hp41programs.yolasite.com> which contains a sizeable collection of classical RPN programs for Analytical Methods, Arithmetic, Astronomy, Calendars, Complex & Hypercomplex Numbers, Differentiation & Integration, The Earth, Geometry, Matrices, Polynomials & Rational Functions, special Functions, Spectral Analysis, Functions of Several Variables, Statistics (*random numbers*), Physics, Games, Miscellaneous, and Links to other program collections.

Five pseudo random number generators are listed here. "RNG1" "RNG2" "RNG3" work on every HP-41. "RNG4" requires a Time-Module. Finally, the last program is an attempt to play (win?) the lottery.

Program#1

A well known RNG is given by the formula: $x_{n+1} = \text{FRC}(9821 x_n + 0.211327)$ which provides 1 million random numbers. The following program gives 1,000,000 random numbers r ($0 \leq r < 1$). The formula $x_{n+1} = \text{FRC}(9^8 x_n + 0.236067977)$ is used.

The coefficient $9^8 = 43,046,721$ may be replaced by a where $a = 1 \pmod{20}$

and 0.236067977 may be replaced by b where $b \cdot 10^9$ is not divisible by 2 or 5.

| | | | | | |
|---------------|--------|--------|--------|---------|--------|
| 01 LBL "RNG1" | 06 * | 11 FRC | 16 * | 21 FRC | 26 END |
| 02 9 | 07 FRC | 12 * | 17 FRC | 22 5 | |
| 03 ENTER^ | 08 * | 13 FRC | 18 * | 23 SQRT | |
| 04 ENTER^ | 09 FRC | 14 * | 19 FRC | 24 + | |
| 05 R^ | 10 * | 15 FRC | 20 * | 25 FRC | |

(35 bytes / SIZE 001)

| | | |
|-------|--------|---------|
| STACK | INPUTS | OUTPUTS |
| X | xn | xn+1 |

Example:

0.2 XEQ "RNG1" yields 0.436067977
 R/S 0.779021394 ... etc ...

Program#2

"RNG2" provides 9,999,999,996 random numbers with the formula: $x_{n+1} = (10^{59} x_n) \text{MOD } p$ where $p = 9,999,999,967$ is the greatest prime $< 10^{10}$. x_n are integers between 0 and p (exclusive) which are then divided by p to be reduced to a number between 0 and 1. This routine works well because the MOD function gives exact results even when the operands are greater than 10^{10} . Actually, the exponent 59 may be replaced by any integer m provided m is relatively prime to $p-1 = 2 \cdot 3 \cdot 11 \cdot 457 \cdot 331543$, but I don't know what is the best choice. Unlike "RNG1" and other routines based upon the same type of formulae, the least significant digits don't go through any cycle of ten, one hundred and so on. Register R00 is used to store the different x_n integers.

| | | | | |
|---------------|---------|--------|-----------|--------|
| 01 LBL "RNG2" | 04 * | 07 33 | 10 STO 00 | 13 END |
| 02 RCL 00 | 05 10 | 08 - | 11 LASTX | |
| 03 E59 | 06 10^X | 09 MOD | 12 / | |

(26 bytes / SIZE 001)

| STACK | INPUTS | OUTPUTS |
|-------|--------|-------------|
| X | / | $0 < r < 1$ |

Example:

1 STO 00

XEQ "RNG2" gives 0.3129146797 (and R00 = 3129146787 = 1059 (mod p))

R/S gives 0.6904570204 (R00 = 690457018) ... etc ...

Actually if p is a prime, $(\mathbb{Z}/p\mathbb{Z} - \{0\}; *)$ is a group and if a is an integer, the number of distinct elements in the subset $\{ 1; a; a^2; \dots; a^k; \dots \} \pmod p$ divides p-1

If p-1 is the smallest positive integer q such that $a^q = 1 \pmod p$, then the sequence $a; a^2; \dots; a^k; \dots; a^{p-1} \pmod p$ is a permutation of $1; 2; \dots; p-1$

In particular, if $p = 2p' + 1$ where p' is also a prime, and if $a^{p'}$ is not equal to $1 \pmod p$ then a satisfies the required property.

For instance, $p = 7,841,296,787 = 2 * 3,920,648,393 + 1$ 7,841,296,787 and 3,920,648,393 are primes and $-1024 = 4,851,307,369 \pmod p$ satisfies $(-1024)^{p'} = -1$ therefore the routine:

E24

*

CHS

7841296787

MOD

gives 7,841,296,786 random integers. These ideas may be used to create your own RNG.

Program#3

The following algorithm is given by Clifford Pickover in "Keys to Infinity" (John Wiley & Sons) ISBN 0-471-11857-5

| | | | |
|---------------|-------|--------|--------|
| 01 LBL "RNG3" | 03 E2 | 05 1 | 07 END |
| 02 LN | 04 * | 06 MOD | |

(17 bytes / SIZE 001)

1st 3 Ex .25x 50%

| STACK | INPUTS | OUTPUTS |
|-------|--------|---------|
| X | xn | xn+1 |

Example:

0.1 XEQ "RNG3" produces 0.74149070

R/S gives 0.09073404 ... etc ...

Program#4

| | | | |
|---------------|--------|----------|--------|
| 01 LBL "RNG4" | 04 + | 07 PI | 10 R-D |
| 02 DATE | 05 E49 | 08 MOD | 11 FRC |
| 03 TIME | 06 * | 09 LN1+X | 12 END |

(25 bytes / SIZE 000)

| STACK | INPUTS | OUTPUTS |
|-------|--------|-------------|
| X | / | $0 < r < 1$ |

I cannot give any example since the result depends on the instant you press R/S

Program#5 - Winning the Lottery?

If you need, for instance, 7 integers between 1 and 49 , you can use 7 random numbers between 0 and 1, multiply them by 49 , add 1 and take the integer part of the result. The small routine hereafter is another possibility, if you accept to calculate the integer part in your mind:

```
01 LBL "$$$"    03 49      05 1      07 END
02 R-D          04 MOD     06 +
```

(16 bytes / SIZE 000)

| STACK | INPUTS | OUTPUTS |
|-------|--------|--------------|
| X | r | $0 < N < 50$ |

Example:

```
41 XEQ "$$$" yields 47.1270
R/S 6.1759, R/S 1.8535, R/S 43.1567, R/S 23.6963, R/S 35.6962 R/S 37.2418
suggesting 47-06-11-43-23-35-37
```

Notes:

- 1- I'm not a statistician and I can't assure all these RNGs would stand up to sophisticated tests, but one may use his imagination in devising variations.
- 2- If you win one million dollars thanks to one of these programs, I accept to share the jackpot...

Selected RPL RAND Application Programs (HP48/49/50)

More meaningful random passwords (Wlodek Meir-Jedrzejowicz)

Many people like to have a random password generator rather than make up their own. The best advice is to combine upper case, lower case, digits and special symbols, but that can make for very unmemorable

passwords. It is often enough to use a string of 7 or more lower case letters - at least those make up something that can be related as a foreign word! The following program generates a string of 7 lower case characters - change 7 to another number if you wish. It uses $RAND\ 26 * 96.5$ to generate the random letter between a and z. 96.5 is used instead of 97 because CHR rounds to the nearest number. Totally random letter combinations contain too few vowels, and too many letters from the end of the alphabet, so I add SQ after RAND to increase the likelihood of the early letters, which contain a higher proportion of vowels. This makes for a higher proportion of readable words, though with too many "a"s in them. Run the program repeatedly until you find a password you like!

```
'RPAS' << "" 1 7 START RAND SQ 26 * 96.5 + CHR + NEXT >>
```

13 commands, 61 bytes, F0B2h. Timing: with πRDZ , \Rightarrow "htabbst" in 176ms. Following: "aaekkg", "sjatauz".

How many random numbers are required (summed) to be \geq to n? (Detlef Müller)

This is an unusual application of START...STEP. Key n and execute 'NRN'.

```
'NRN' << 0 0 ROT START 1 + RAND STEP >>
```

8 commands, 30.0 Bytes, # 43DCh.

Reminder. In order to get the same results you must first store a value in RDZ. If RDZ starts with the $\sqrt{5}$ the results for n = 7 are. 17, 14, 13, 15, 16, 15, 11 . . .

Dealing a deck of cards

Drawing a BINGO^(B2) number or dealing a deck of cards^(B3) are examples of using random numbers as selection without replacement. A fixed number of random numbers ordered randomly and used. This problem may be solved in many ways depending on how it will be used. Breaking the program into parts allows the reader/student to better understand how the programs work. Four HP 48 programs are used as described in Table B0.

Table B0 – RPL Card Dealing Program Statistics

| Name | Description | Size | Calls | Chk Sum ^[a] |
|--------------|---|--------|--------------------------|------------------------|
| DEAL | Deals a card. | 59.5 B | DECK, & CRDID | # 6004h |
| CRDID | Identifies/displays the card with value & suit as a text string. | 219 B | — | # 31BDh |
| DECK | A list of randomly ordered card numbers (1 – 13 is Spades, 14 – 26 is Hearts, 27 – 39 is Diamonds, 40 – 51 is clubs). | 479 B | — | #3FA3h |
| MAKE | Creates a list of randomly ordered card numbers 1 to 52. | 77.5 B | — | # 47B8h |
| SEED | Makes repeatable deck, not required, only used for testing. | 17.5 B | — | # 7CF0h |

Note: Check sum is for program only. The deck is based on a seed of $\sqrt{5}$. Variables not native to the HP 48 are in bold.

```
'DEAL' << DECK HEAD LASTARG TAIL 'DECK' STO CRDID MEM DROP >>
```

Note MEM DROP is not required unless memory is inadequate. This sequence clears LASTARG, etc.

```
'CRDID' << { ACD DUCE TREY FOUR FIVE SIX SEVEN EIGHT NINE TEN JACK QUEEN KING } OVER 1 – 13 MOD 1 + GET " of " + { Spades Hearts Diamonds Clubs } ROT 13 / CEIL GET + >>
```

Note that the text is entered as a name which is automatically converted to a string when it is concatenated with the " of " text string.

```
'DECK' { 17 46 37 24 36 21 41 47 23 52 44 49 32 39 7 19 1 22 12 28 9 5 26 40 34 14 10 38 25 13 2 48 11 29 29 31 42 43 8 27 35 45 15 18 39 33 6 34 51 16 59 } Note: SEED, (RDZ) IS  $\sqrt{5}$ .
```

```
'MAKE' << 1 52 FOR n n DUP RAND * CEIL ROLLD NEXT 52 →LIST 'DECK' STO >>
```

```
'SEED' << 5  $\sqrt{RDZ}$  >>
```

If all 52 cards are delt and the output is put into a list of text strings the list will be 1,006 bytes with a check sum of # 51C2h,

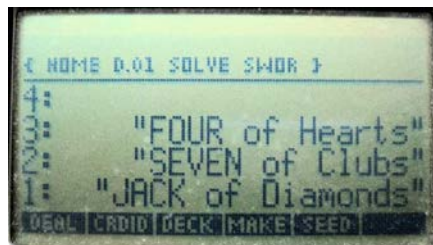


Fig B1 – 1st three cards dealt. Seed = $\sqrt{5}$.

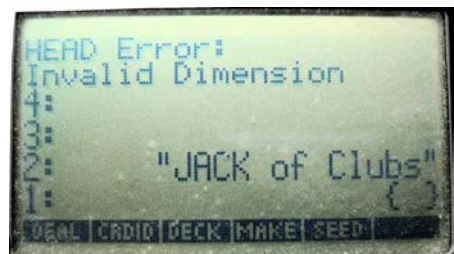


Fig. B2 – Error with 53rd card dealt.

Selected Pascal like **RAND** programs for the HP 39gII

Namir Shammas

Three HP 39gII listings for:

1. A program that simulates tossing a fair coin.
2. A program that simulates tossing a pair of dice.
3. A program that simulates shuffling a deck of cards and drawing cards from that deck.

The Coin Tossing Program

Table B1 shows the listing for the coin tossing program. The program calls the function **RANDOM(0,1)** to generate a random number between 0 and 1. The program passes the random number to function **ROUND** to round that value to either 0 or 1. Based on the result, the program **COIN** returns the text **HEADS** or **TAILS**.

Table B1 – HP 39gII Program for Flipping a Coin

| Statement | Comment |
|---------------------------------|---|
| EXPORT COIN() | |
| BEGIN | |
| IF ROUND(RANDOM(0,1),0)==0 THEN | Generate a random number between 0 and 1, and round it to either 0 or 1. Test if the result is 0. |
| RETURN "TAILS"; | If the result is zero, return Tails |
| ELSE | |
| RETURN "HEADS"; | Otherwise, return Heads. |
| END; | |
| END; | |

Figure B1 shows sample sessions with the function **COIN**. The output is the result of executing the function **COIN** multiple times.

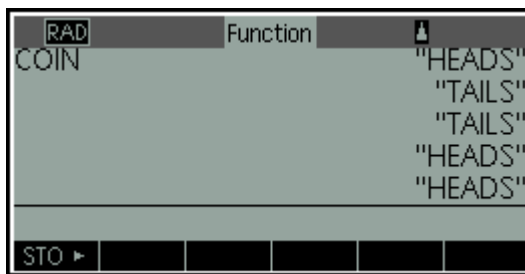


Figure B1. Sample sessions with program COIN.

The Dice Tossing Program

Table B2 shows the listing of a program that generates the result of rolling a pair of dice. To calculate the value of each die, the program uses the expression **RANDOM(1,6)** rounded to 0 decimals. If you want to change the program to emulate playing with eight-face dice, then use **RANDOM(1,8)** in each of the two assignment statements.

Table B2 – HP 39gII Program for Throwing Dice

| Statement | Comment |
|---------------------------|---------------------------------|
| EXPORT DICE() | |
| BEGIN | |
| LOCAL d1,d2; | |
| d1:=ROUND(RANDOM(1,6),0); | Simulate rolling the first die. |

| Statement | Comment |
|---------------------------|----------------------------------|
| d2:=ROUND(RANDOM(1,6),0); | Simulate rolling the second die. |
| RETURN CONCAT(d1,d2); | Return a list of the two dice. |
| END; | |

Figure B2 shows sample sessions with the function **DICE**. The output is the result of executing the function **DICE** multiple times.



Figure B2. Sample sessions with program DICE.

Now let's look at another version of the dice throwing program—one that throws loaded dice. The next program **DICE2** is a function that heavily favors the number 6. Table B3 shows the source code for function **DICE2**. The program generates values for the dice using **RANDOM(1,8)** and stores them in variables **d1** and **d2**. The function **DICE2** examines the values in **d1** and **d2** to determine if they exceed 6. If so, the function assigns 6 to either variables. This scheme gives each of the faces 1 through 5 a 12.5% chance (down from 16.67% for a fair die), while handing the face 6 a whopping 37.5% chance! If you replace **RANDOM(1,8)** with **RANDOM(1,10)** in the program, the face 6 will have a 50% chance against a 10% chance for each of the other die faces.

Table B3 – The cheater's Dice program

| Statement | Comment |
|---------------------------|--|
| EXPORT DICE2() | |
| BEGIN | |
| LOCAL d1,d2; | |
| d1:=ROUND(RANDOM(1,8),0); | Simulate rolling the first die. |
| IF d1>6 THEN | If the value of d1 exceeds 6, set it to 6. |
| d1:=6; | |
| END; | |
| d2:=ROUND(RANDOM(1,8),0); | Simulate rolling the second die. |
| IF d2>6 THEN | If the value of d2 exceeds 6, set it to 6. |
| d2:=6; | |
| END; | |
| RETURN CONCAT(d1,d2); | Return a list of the two dice. |
| END; | |

Figure B3 shows a sample output from the function **DICE2**. Notice how many 6s appear!



Figure B3. Sample sessions with program DICE2.

The Deck of Cards Program

Unlike the first three programs that are short and simple, the next set of programs are more elaborate. They perform the following tasks:

- Initialize a deck of cards.
- Shuffle the deck of cards.
- Draw a card out of the deck.

Initializing the Deck

Table B4 shows the listing for the **InitDOC** program that initializes the deck of cards. The first few statements in the listing export the following variables:

- The matrix **cards** which has one row and 52 columns. This matrix stores the numeric codes that represents the cards in the deck.
- The variable **numCards** stores the number of cards in the deck.
- The variable **setNames** stores a list of card face names.
- The variable **cardNames** stores a list of card names.
- The variable **stackHeight** stores the current number of available cards to draw.

The exported matrix, **cards**, is a single-row matrix with 53 columns. To examine its contents at any time between function calls, copy the data in that matrix into one of the ten global matrices. Then use the

Table B4 – The Card Initialization Program

| Statement | Comment |
|--|---|
| EXPORT cards,numCards; | Export variables used by the other related functions. |
| EXPORT setNames, cardNames; | |
| EXPORT stackHeight; | |
| EXPORT InitDOC() | |
| BEGIN | |
| LOCAL i,j,k; | |
| numCards:=4*13; | Calculate the number of cards in the deck. |
| cards:=MAKEMAT(0,1,numCards); | Create the row of data that stores the numeric codes for the cards. |
| k:=0; | |
| FOR i FROM 1 TO 4 DO | |
| FOR j FROM 1 TO 13 DO | |
| k:=k+1; | |
| cards(1,k):=100*i+j; | Store the numeric codes of the cards in matrix cards. |
| END; | |
| END; | |
| setNames:=CONCAT("Diamonds","Clubs","Hearts","Spades"); | Initialize the names of the sets of cards. |
| cardNames:=CONCAT("Ace","2","3","4","5","6","7"); | Initialize the names of the cards. |
| cardNames:=CONCAT(cardNames,"8","9","10","Jack","Queen","King"); | |
| RETURN "DECK INITIALIZED";- | Return affirmation message |
| END; | |

matrix editor to view the value in the global matrix. Table B5 shows the ranges of values used with each face.

Table B5– The Numeric Codes for the Cards

| <i>Range</i> | <i>Meaning</i> |
|--------------|---|
| 101 to 113 | Ace of diamonds, 2 of diamonds, ..., queen of diamonds, and king of diamonds. |
| 201 to 213 | Ace of clubs, 2 of clubs, ..., queen of clubs, and king of clubs. |
| 301 to 313 | Ace of hearts, 2 of hearts, ..., queen of hearts, and king of hearts. |
| 401 to 413 | Ace of spades, 2 of spades, ..., queen of spades, and king of spades. |

Shuffling the Cards

This section presents two functions that differently shuffle the deck of cards. The first function, SHUFCARDS, shuffles the cards, computer style, as if they were an array of integers. The basic algorithm performs an in-place shuffling of cards (treated as array of integers). The leading array elements (or rows of the matrix cards) represent the cards *TO BE* shuffled. The trailing array elements represent the cards *THAT WERE* shuffled. Initially, the number of the cards to be shuffled is 52 and the number of shuffled cards is 0. The function selects an element in the range of 1 to 52 and swaps that element with the

last card. Now there are 51 cards to shuffle and one shuffle card. The function selects an element in the range of 1 to 51 and swaps that element with the second-from-last card (at index 51). The third pass selects a card in the range of 1 to 50, and so on.

Table B6 shows the listing of function SHUFCARDS.

Table B6– The listing for the SHUFCARDS Program

| Statement | Comment |
|---------------------------|--|
| EXPORT SHUFCARDS() | |
| BEGIN | |
| LOCAL nc,nc2,i,j,k,temp; | |
| nc:=numCards; | Set nc to be the initial number of cards to be shuffled. |
| nc2:=0; | Set nc2 to be the initial number of shuffled cards. |
| WHILE nc>1 DO | Iterate while there are 2 or more cards to shuffle. |
| i:=ROUND(RANDOM(1,nc),0); | Select a card in the range of 1 to nc. |
| j:=numCards-nc2; | Store the index of the next shuffled card. |
| temp:=cards(1,j); | Swap the shuffled card(1,i) with card(1,j). |
| cards(1,j)=cards(1,i); | |
| cards(1,i)=temp; | |
| nc2:=nc2+1; | Increment the number of shuffled cards. |
| nc:=nc-1; | Decrement the number of cards to be shuffled |
| END; | |
| stackHeight:=numCards; | Initialize the number of available cards to draw. |
| RETURN "DECK SHUFFLED"; | |
| END; | |

It is also possible to shuffle the arrays of cards by repeatedly selecting two cards at random and then swapping them. Here is the algorithm for this approach:

1. For MAX_SHUFFLE times perform the remaining tasks
2. Select i as a random number between 1 and 52.
3. Select j as a random number between 1 and 52.
4. If i and j are different swaps the values at element i and j.

The above algorithm is easy to implement but does not guarantee that every card is moved to a new random location in the deck. The value of MAX_SHUFFLE should be high enough to ensure a good card shuffling.

The second card shuffling program emulates splitting the deck of cards around the middle (between 40% to 60% of the number of cards) and then merging the two halves. The function moves the cards from the split deck onto another deck of cards, by alternating card selection from each sub-deck.

Table B7 shows the listing of function **SPLITCARDS**. This function has the parameter **numSplits** which tells the function how many times to split and merge the cards.

Drawing a Card

Table B8 shows the listing of function **GETCARD** which returns a card from the shuffled deck. The function uses the exported variable **stackHeight** to determine the next card to draw, if there are cards available to draw. The function returns a list containing the face name and the card name.

Table B7– The listing for the SPLITCARDS program

| Statement | Comment |
|---|--|
| EXPORT SPLITCARDS(numSplits) | |
| BEGIN | |
| LOCAL ii,cardsCopy,median; | |
| LOCAL i,i1,i2; | |
| cardsCopy:=MAKEMAT(0,1,numCards); | |
| FOR ii FROM 1 TO numSplits DO | Repeat the shuffling of cards numSplits times. |
| FOR i FROM 1 TO numCards DO | Make a duplicate card deck. |
| cardsCopy(1,i):=cards(1,i); | |
| END; | |
| median:=ROUND(RANDOM(0.4*numCards,0.6*numCards),0); | Select a median for splitting the card. |
| i:=1; | Initialize the indices for copying cards from the two sub-decks. |
| i1:=1; | |
| i2:=median+1; | |
| REPEAT | Start merging alternating cards from the split deck. |
| IF i1≤median THEN | Any more cards from the first sub-deck to merge? |
| cards(1,i):=cardsCopy(1,i1); | Copy the card to the merged deck. |
| i1:=i1+1; | |
| i:=i+1; | |
| END; | |
| IF i2≤numCards THEN | Any more cards from the second sub-deck to merge? |
| cards(1,i):=cardsCopy(1,i2); | Copy the card to the merged deck. |
| i2:=i2+1; | |
| i:=i+1; | |
| END; | |
| UNTIL i1>median AND i2>numCards; | Stop when all cards have been merged. |
| END; | |
| stackHeight:=numCards; | Set the number of available cards to draw. |
| RETURN "DECK SPLIT SHUFFLED"; | |
| END; | |

Table B8– The listing for the GETCARD program

| Statement | Comment |
|--|---|
| EXPORT GETCARD() | |
| BEGIN | |
| LOCAL i,j; | |
| | |
| IF stackHeight>0 THEN | Can we draw another card? |
| i:=cards(1,stackHeight); | Get the numeric code for the next card to draw. |
| j:=i; | |
| i:=INT(i/100); | Calculate card face index. |
| j:=j-100*i; | Calculate card value. |
| stackHeight:=stackHeight-1; | |
| RETURN CONCAT(setNames(i),cardNames(j)); | Return a list that identifies the card drawn. |
| ELSE | |
| RETURN "NO CARDS AVAILABLE!"; | |
| END; | |
| END; | |

Sample Sessions

Let's use the functions **InitDOC**, **SHUFCARDS**, and **GETCARD** to initialize a deck of cards, shuffle the cards, and draw cards, respectively. Figure B4 shows the output when using the following functions:

- The function **InitDOC** which initializes the deck of cards.
- The function **SHUFCARDS** that shuffles the deck by randomly arranging the cards.
- The function **GETCARD** that draws a card from the deck. Figure B4 shows two calls to function **GETCARD**.

| Function | Output |
|-----------|----------------------|
| InitDOC | "DECK INITIALIZED" |
| SHUFCARDS | "DECK SHUFFLED" |
| GETCARD | {"Diamonds","Queen"} |
| GETCARD | {"Diamonds","10"} |

Figure B4. Using functions *InitDOC*, *SHUFCARDS*, and *GETCARD*.

Figure B5 shows additional calls to function **GETCARD**.

| Function | Output |
|----------|-------------------|
| GETCARD | {"Diamonds","10"} |
| GETCARD | {"Diamonds","2"} |
| GETCARD | {"Clubs","10"} |
| GETCARD | {"Hearts","3"} |

Figure B5. Additional calls to function *GETCARD*.

Now, let's use the functions **InitDOC**, **SPLITCARDS**, and **GETCARD** to initialize a deck of cards, shuffle the cards, and draw cards, respectively. Figure B6 shows the output when using the following functions:

- The function **InitDOC** which initializes the deck of cards.
- The function **SPLITCARDS(7)** that shuffles the deck by splitting and merging the cards seven times.
- The function **GETCARD** that draws a card from the deck.

| RAD | Function | |
|---------------|-----------------------|--|
| InitDOC | "DECK INITIALIZED" | |
| SPLITCARDS(7) | "DECK SPLIT SHUFFLED" | |
| GETCARD | {"Diamonds","4"} | |
| STO ▶ | | |

Figure B6. Using functions *InitDOC*, *SPLITCARDS*, and *GETCARD*.

Figure B7 shows additional calls to function **GETCARD**.

| RAD | Function | |
|---------|------------------|--|
| GETCARD | {"Diamonds","4"} | |
| GETCARD | {"Clubs","Jack"} | |
| GETCARD | {"Spades","2"} | |
| GETCARD | {"Diamonds","9"} | |
| STO ▶ | | |

Figure B7. Additional calls to function *GETCARD*.

Bonus Program

This section presents a bonus program that allows you to check that a shuffled deck contains no duplicates or missing cards. Table B9 contains the listing for the **COMPARE** function. This function has one parameter, **myCards**, which is the single-row matrix that stores the numerical codes for the various cards. The function compares the values in the argument for **myCards** with those in an internally created standard card deck. The function returns the number of missing or duplicate cards. A zero result indicates that the argument for **myCards** stores a valid deck of cards.

Figure B8 shows a sample session for using function **COMPARE**. The output of this function shows that the shuffled cards, stored in variable **cards**, match the cards of a standard deck.

| RAD | Function | |
|----------------|-----------------------|--|
| InitDOC | "DECK INITIALIZED" | |
| SPLITCARDS(7) | "DECK SPLIT SHUFFLED" | |
| COMPARE(cards) | 0 | |
| STO ▶ | | |

Figure B8 - Additional calls to function *GETCARD*

Let's test **COMPARE**'s ability to catch errors in a corrupted deck of cards. Perform the following tasks:

- Execute the **InitDOC** function to initialize the deck of cards.
- Set cards(1,2) to 101 by typing **cards(1,2)=101**.
- Set cards(1,3) to 101 by typing **cards(1,3)=101**. Now we have two duplicates 101 and no values for 102 and 103.
- Execute **COMPARE(cards)**. The function returns 2, the number of duplicate/missing cards.

Figure B9 and B10 show the above tasks.

Table B9 – The listing for the COMPARE Program

| Statement | Comment |
|-------------------------------------|---|
| EXPORT COMPARE(myCards) | |
| BEGIN | |
| LOCAL stdCards; | |
| LOCAL i,j,k,n; | |
| | |
| stdCards:=MAKEMAT(0,1,numCards); | Create matrix that will store the data for a standard deck of cards. |
| k:=0; | |
| FOR i FROM 1 TO 4 DO | Start nested loop to store numerical codes for cards in matrix stdCards. |
| FOR j FROM 1 TO 13 DO | |
| k:=k+1; | |
| stdCards(1,k):=100*i+j; | |
| END; | |
| END; | |
| | |
| n:=numCards; | Initialize number of different cards. |
| FOR i FROM 1 TO numCards DO | Start looping for each member of matrix stdCards. |
| FOR j FROM 1 TO numCards DO | Start looping for each member of matrix myCards. |
| IF stdCards(1,i)==myCards(1,j) THEN | If the values of stdCards(1,i) and myCards(1,j) have the same values, decrement variable n. |
| n:=n-1; | |
| j:=numCards; | Set j to last loop value for an early loop exit. |
| END; | |
| END; | |
| END; | |
| RETURN n; | Return the number of different cards. |
| END; | |

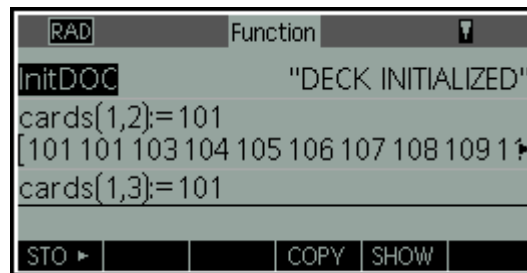


Figure B9. Testing a corrupted deck of cards, part 1.

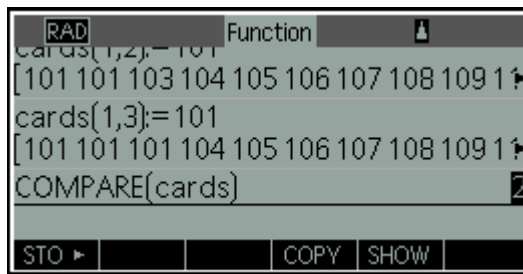


Figure B10. Testing a corrupted deck of cards, part 2.

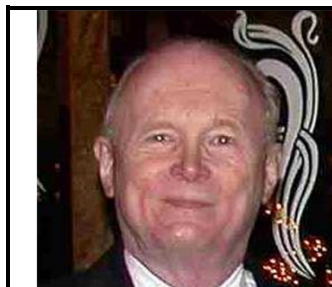
References

1. Hishashi Tanizaki, *Computation Methods in Statistics and Econometrics*. Available from <http://stat.econ.osaka-u.ac.jp/~tanizaki/cv/books/cmse/cmse.pdf>.
2. Luc Devroye, *Non-Uniform Random Variate Generation*. Available from <http://www.eirene.de/Devroye.pdf>.
3. HP-65 STAT PAC 1 manual, page 28.
4. HP 39gII User's Guide, Edition 1, November 2011.

Notes for The Randomness of HP Appendix B

- (B1). For a discussion on how HP's RPN has evolved see the article *HP RPN Evolves* (pdf file) at: <http://h20331.www2.hp.com/hpsub/downloads/S07%20HP%20RPN%20Evolves%20V5b.pdf>
- (B2) These programs are taken from an Educalc class handout from instructors Joe Horn & Richard Nelson titled *BINGO – Part I* (identifying the balls, 2 pp.) and part II (creating, mixing, and selecting the balls, 4 pp.) dated March 27, 1998.
- (B3) The card identification program was taken from a 7 pp. HHC 1994 HP 49 Programming Problem CHIP Meeting – June 24, 1994 titled *Describe Playing Card from its number* by Richard J. Nelson. This is written from a Programming Exercise perspective which includes multiple programs for a problem with each version making improvements towards an optimum shortest/fastest program as a student's thought process to learn programming.

About the Author



Richard J. Nelson has written hundreds of articles on the subject of HP's calculators. His first article was in the first issue of *HP 65 Notes* in June 1974. He became an RPN enthusiast with his first HP Calculator, the HP-35A he received in the mail from HP on July 31, 1972. He remembered the HP-35A in a recent article that included previously unpublished information on this calculator. See <http://hhuc.us/2007/Remembering%20The%20HP35A.pdf> He has also had an article published on HP's website on HP Calculator Firsts. See <http://h20331.www2.hp.com/Hpsub/cache/392617-0-0-225-121.html>. Email Richard at: rjnelsoncf@cox.net.

About the Author



Namir Shammas is a native of Baghdad, Iraq. He resides in Richmond, Virginia, USA. Namir graduated with a degree in Chemical Engineering. He received a master degree in Chemical engineering from the University of Michigan, Ann Arbor. He worked for a few years in the field of water treatment before focusing for 17 years on writing programming books and articles. Later he worked in corporate technical documentation. He is a big fan of HP calculators and collects many vintage models. His hobbies also include traveling, music, movies (especially French movies), chemistry, cosmology, Jungian psychology, mythology, statistics, and math. As a former PPC and CHHU member, Namir enjoys attending the HHC conferences. **Email Namir at: nshammas@aol.com**