About The Solver in the HP Pocket Calculators

Martin Hepperle, updated November 2018

The following description was taken from the booklet “Technical Applications” in the series “Step-By-Step Solutions” published by Hewlett Packard in 1988. It had been written for the HP 27S and HP 19B but is also applicable to the HP17B family of calculators as long as they have the LET and GET functions. Note however, that these calculators do not have trigonometric functions (which can be approximated by techniques shown in this text).

Note that the HP 17B has been reissued in form of several more or less improved versions (17B, 17BII, 17BII+ version 1 and 17bII+ version 2). In case of the later HP 17BII+ calculators we see two designs: a smaller one with a gold front plate and a re-engineered version with a larger form factor and a silver front plate. Unfortunately both of the 17BII+ designs include some software optimization that render the GET and LET functions almost unusable when assigning a value to a variable that appears in the menu of the equation. Therefore the older 17B and 17BII models a recommended if you plan to use GET and LET.

Excerpt from

“Technical Applications”

Step-by-Step Solutions for Your HP Calculator

Advanced Solver Techniques

Using LET and GET

Function Descriptions

These two functions are not covered in your owner's manual. However, you'll find them useful in a variety of applications. LET assigns the value of an algebraic expression (or number) to a specific variable. GET recalls the contents of a specific variable. The format for these two functions is:

Function:	Description:
`L ( variable name : algebraic expression)`	Evaluates the algebraic expression, stores the result in the specified variable, and also returns that result as the value of the LET function.
`G (variable name)`	Returns the contents of the specified variable.

Like Σ and IF, LET and GET are for use only in Solver equations. Thus, you will not find these functions on a calculator key or in any menu. To use LET and GET in a Solver equation, simply type the letters G or

L and include the parentheses around the arguments. If a variable appears only as the first argument of a LET function and/or only as the argument of a GET function, it will not appear in the menu of variables.

There are many ways in which LET and GET enhance the capabilities of your Solver, and we will describe each of these in the pages that follow.

First, a few examples will help introduce you to these two powerful functions.

Example 1

The Solver equation

```
A = L(D:B+C)+SIN(D)
```

stores the sum of B and C in D, and adds this result to the sine of D when calculating A. The LET function causes the value of D used in the argument of the sine function to be B+C. Notice that when this equation is "CALC"ed, A, B, C, and D all appear in the menu, but it is not necessary to explicitly enter a value for D since the LET function does so automatically.

Example 2

The Solver equation

```
A=L(D:B+C)+SIN(G(D))
```

is functionally identical to the last example when calculating a result for A, but here the GET function is used in computing the sine of D. When this equation is "CALC"ed, D does not appear in the menu. This is because D only appears as the first argument of the LET function and as the argument of the GET function in this Solver equation. In the last example, D was the argument of the SIN function (an appearance outside the first argument of LET or GET), and thus did appear in the menu.

Intermediate Variables

Normally, whenever you use a variable in a Solver equation, it will appear in the menu. When variables are used in this manner they are referred to as formal variables or simply variables when no distinction is necessary.

However, there are two cases in which a variable will not appear in the menu:

1. The only occurrence of a variable is as the counter variable of the Σ function.

2. The only occurrence of a variable is as the first argument of LET and/or as the argument of GET.

The second case was illustrated in Example 2 for the variable D. When a variable is used only as the first argument of LET and as the argument of GET it is given the special name intermediate variable. This is because intermediate variables hold intermediate results that can be used repeatedly in an equation, even though such variables do not appear in the menu. The user cannot directly assign a. value to an intermediate variable. Keeping an intermediate variable from appearing in the menu avoids confusion since the menu prompts only for relevant variables.

Example 3

The equation $a = b + c - \ln\left(b + c\right) + x^{b+c}$ can be implemented by the Solver equation

```
A=L(D:B+C)-LN(G(D))+X^G(D)
```

Here, `D` serves as an intermediate variable and will not show up in the menu. The use of `D` in this manner avoids having to type `B+C` more than once in the equation and keeps `D` from appearing in the menu, a source of possible confusion since there is no variable d in the original equation.

Reducing Keystrokes with `LET`

In some instances you may wish to use the `LET` function to reduce keystrokes (as shown in the previous examples), yet still view the value of the variable used with the `LET` function. This is accomplished by simply including the variable formally one or more times in your equation, thus causing it to appear in the menu of variables. An example will help clarify this.

Example 4

The Solver equation from Example 3 can be changed slightly to

```
A=L(D:B+C)-LN(D)+X^D
```

Notice that the `GET` function has simply been removed and now `D` appears formally two times; as the argument of ln, and as the power to which x is raised. We have still employed `LET` to reduce redundant keystrokes (typing `B+C` more than once), but now the sum b + c can be viewed by recalling `D`.

A practical application of these last two techniques is in the calculation of planetary orbits as shown in the next example.

Example 5: Orbits of Planets

The equation in polar coordinates for a planet's orbit about the sun is given by:

$$r = \frac{p \cdot e}{1 + e \cdot \cos\left(\theta\right)}$$

where

$$e = \frac{r_0 \cdot v_0^2}{G \cdot M} - 1$$

Here, r is the orbital radius, θ is the angle swept out by the planet as it orbits, p is the distance between the focus and directrix (the orbit is always one of the conic sections), r_0 is the planet's orbital radius at its closest approach to the sun, v_0 is its speed at the point of closest approach, G is the universal gravitational constant, and M is the mass of the sun.

To simplify the orbital equation, we have used the variable e. You are probably familiar with this method of notation for complicated expressions. In an analogous manner, repetitive keystrokes can be eliminated

in a Solver equation by using the variable E. The following Solver equation for the planet's orbit uses ANG to represent theta.

```
R=P×L(E:R0×SQ(V0)/(G×M)-1)/(1+E×COS(ANG))
```

Here we have used the LET function to assign a value to E and thus eliminate the need to type the rather long expression for E again later in the equation. Since the variable e in the original equations has special significance (the eccentricity of orbit), E is not used as an intermediate variable; instead, it appears formally as a multiplier of $\cos(\theta)$ and thus appears in the menu of variables. If it were not necessary to view E, the Solver equation

```
R=P×L(E:R0×SQ(V0)/(G×M)-1)/(1+G(E)×COS(ANG))
```

could be used. Notice that the only change made from the previous Solver equation is to employ the GET function with E. We have still used LET to reduce keystrokes by assigning a value to E, and the equation is functionally identical to the previous Solver equation when R is solved for. However, now E does not appear in the menu. In this case, E is used as an intermediate variable.

How LET and GET Change an Equation

In many cases, when an unknown variable appears only once as a formal variable, the Solver algebraically rearranges the equation to isolate a direct solution. However, if an unknown variable appears formally more than once, the direct solution method always fails and the Solver attempts to locate a solution iteratively. When a variable appears as the first argument of LET or as the argument of GET, it is not considered by the Solver in determining whether a direct solution can be found. Thus, a variable may occur many times in an equation, yet only once formally. In these instances, a direct solution may be found, but it will not normally be "correct" mathematically. The next example will clarify this.

Example 6

The Solver equation

```
A=2×A+B
```

is solved iteratively for A (since A appears twice formally) such that A=−B. On the other hand, the Solver equation

```
A=2×G(A)+B
```

is treated much differently. When A is solved for, a direct solution is found (A appears formally only once). The Solver multiplies the current contents of A by 2, adds the contents of B, and stores this result as the new value of A.

Using New and Old Values

As shown in "How LET and GET Change and Equation," the Solver will often return a solution that is not "correct" in the strict mathematical sense when LET and GET occur in an equation. Actually, this result is quite useful and allows LET and GET to be used to assign new values to variables using the values they currently contain. This technique can be used in recursive problems; i.e., problems in which the next value

of the output is dependent on the old output. Whenever you encounter an equation in which the unknown variable appears as both a formal variable and as the argument of a GET function, the GET function will use the current value of the variable (the value of the variable when the calculation is initiated).

A helpful aid in understanding this is to think of the variable as behaving in two different ways:

1. Where it appears formally, the variable is used as it would normally be in finding a solution to an equation; i.e., it can be algebraically rearranged to find a direct solution or used to find an iterative solution.

2. As the argument of a GET function, the variable is treated as though it were a constant, not an unknown. The value used is the current value as previously described.

In the LET function, the Solver does not check to see if a variable appears on both sides of the colon. Instead, it simply evaluates the algebraic expression on the right side of the colon using the current values of all variables. This result is assigned as the new value of the variable on the left side of the colon.

Some specific examples will demonstrate these features.

Example 7

Consider the Solver equation

```
A=G(A)+1
```

Since A only appears once formally, the direct Solver is used. This equation recalls the current value of A (by the GET function) then adds 1. This result is then assigned as the new value of A. Thus, this simple equation increments A by 1 each time A is calculated.

Example 8

Like the last example, the Solver equation

```
Q=L(A:A+1)
```

also increments A by 1. In this case, Q is calculated instead of A. As mentioned above, the right side of the colon in the LET argument (A+1) is evaluated using the current value of A and this result is assigned as the new value of A. Note that A appears on the right side of the colon in the LET argument and therefore appears in the menu when the equation is "CALC"ed.

The next example shows a practical use of these techniques.

Example 9: Taylor Series Expansion

The Taylor Series expansion for e is

$$e = \sum_{j=0}^{\infty} \frac{1}{j!}$$

How many terms are needed in the series to express e accurately to 4 decimal places?

The Solver equation

```
E=G(E)+1/FACT(L(J:J+1))
```

will accomplish this. The GET used with E causes E to appear only once formally, and thus the direct method is used when E is solved for. The GET technique of Example 7 is used to add a new term $1/j!$ to the current value of E each time E is calculated. This new result is then returned as the "solution" for E.

Using the LET technique of Example 8, the equation increments J by 1 each time E is calculated. When the equation is "CALC"ed, E and J appear in the menu of variables. You should set E initially by storing a zero in it, since the sum should begin by GETting a zero value for E. From the defining equation for the series expansion of e, the sum should start at $j = 0$. J in the Solver equation is set to −1 initially so that the value of J used by the factorial function is zero the first time E is calculated.

To view the full precision of the number as the Solver adds each term in the series, select the display format "ALL". You will find that E must be calculated 8 times to express e accurately to 4 decimal places. Thus, 8 terms ($j = 0$ to 7) are needed to achieve the specified accuracy.

When Not to Use LET and GET

There are times when you must iterate to find the solution to an equation. The previous examples have shown how LET and GET change the way an equation is evaluated when their arguments are unknowns. Sometimes this can result in a direct solution that is not the desired solution.

Consider the equation $x = -e^x$. This is a transcendental equation; i.e., it cannot be algebraically manipulated to isolate a solution for x. Here it is necessary to iteratively find a solution. If you write the Solver equation

```
X=-EXP(G(X))
```

with hopes that the Solver will somehow find a direct solution (since, formally, X only appears once), you are asking for a mathematical impossibility. The Solver equation above takes the base e, raises it to the X the power based on an initial value of X, and assigns the negative of this result as the final value of X.

Arranging Menu Variables

At times you may wish to arrange your equation variables in a specific order in the menu. There are two ways to do this. LET and GET Method. Recall that a variable only appears in a menu when it appears formally in an equation. When you press [CALC] after entering an equation, the Solver scans from left to right in your equation and assigns variables to the menu in the order in which they are encountered. A variable is not considered by the Solver in menu assignments under the following conditions:

1. When a variable is used as a counter variable in the Σ function.

2. When a variable is used as the first argument of LET.

3. When a variable is used as the argument of GET.

This technique is shown in the example below.

Example 10

The equation $a = \ln(b \cdot d) + e^c + d^2$ can have its variables arranged alphabetically using the Solver equation

```
A=LN(B×G(D))+EXP(C)+SQ(D)
```

Since `D` first occurs as the argument of a `GET` function, it is not assigned to the menu until it is encountered as a formal variable in the term `SQ(D)`.

Note

This technique of arranging menu variables is most useful when certain variables will not be unknowns. If the Solver equation of Example 10 is used to calculate `D`, the `GET` function causes new and old values of `D` to be used rather than a true algebraic solution. If the equation always calculates variables other than `D`, it is perfectly acceptable.

Multiplication by 0 Method

Unlike the `LET` and `GET` method, multiplication by zero does not cause the Solver to use new and old values of a variable. Instead, the true mathematical integrity of the equation is preserved. The next example shows how.

Example 11

The Solver equation of Example 10 can be rewritten as

```
A=LN((B+0×C)×D)+EXP(C)+SQ(D)
```

This will arrange the menu variables alphabetically and still allow all variables to be calculated. Notice that as the Solver scans this equation, it encounters the formal variables in the order `A`, `B`, `C`, and `D`. The multiplication by zero adds nothing to the argument of the natural logarithm function and its sole purpose is to arrange the variables alphabetically in the menu.

Note

Multiplication by zero causes variables that appear only once in a defining equation to appear more than once in a Solver equation. If these variables are unknown, the Solver will iteratively locate a solution since they appear formally more than once. For example, `C` appears formally twice in the Solver equation above even though it appears only once in the defining equation of Example 10. Thus, when `C` is calculated, the Solver will locate a solution iteratively.

Solving for More Than One Variable at a Time

In the last example, multiplication by zero was used to arrange the menu of variables in a Solver equation. Multiplication by zero has another very powerful use in Solver equations ... assigning a result to more than one variable when a single unknown is calculated. This is done by employing the `LET` function and multiplication by zero. An example will illustrate this technique.

Example 12: Nautical Depth Conversions

Nautical depths are often measured in fathoms. Since most people do not "think" in terms of fathoms, conversion to more customary units is desirable. To convert from fathoms to feet, multiply by 6.000012; to convert from fathoms to meters multiply by 1.828804. A Solver equation that converts fathoms to feet and meters at the same time is

```
FT=FATH×6.000012+0×L(M:FATH×1.828804)+0×M
```

Here we assume that fathoms (FATH) is the known quantity, and feet (FT) and meters (M) are unknown. Notice that when FT is calculated, the Solver multiplies FATH by the proper conversion factor. Then, the LET function is multiplied by zero so that its value will not affect the value of FT. The LET function will assign the proper value to M. We have also included the term 0*M to cause M to appear formally and hence, in the menu of variables. When FT is calculated, the Solver returns a proper result and stores the number of meters in M. To see M, you must recall it using RCL M.

Note

This technique of solving for more than one unknown is most useful when certain variables will not be unknowns. The Solver equation of Example 12 is intended to be used only when FATH is known and FT and M are to be found. Solving for M will result in the message SOLUTION NOT FOUND since the direct Solver will attempt to isolate M and a division by zero error occurs. In general, this technique should not be employed when all variables in an equation will be calculated.

The next example combines several of the techniques you have learned so far to solve a practical problem.

Example 13: Complex Multiplication

To multiply two complex numbers $x = a + i \cdot b$ and $y = e + i \cdot d$, use the formula

$$x \cdot y = \left(a \cdot e - b \cdot d\right) + i \cdot \left(b \cdot e + a \cdot d\right) .$$

A Solver equation can be written that calculates the product, stores the real part of the product as a and the imaginary part as b, and leaves e and d unchanged. This makes the equation useful for chain calculations.

`0×L(R:A×G(C)-B×G(D))`	Stores the real part of the product $x \cdot y$ in the intermediate variable R. The intermediate variable R is employed since we do not want to store the real part of the product in A yet. Before A can be assigned a new value, the current value of A is needed to calculate the imaginary part of the product. Notice that GET is used with C and D so that the menu of variables will be in the order A, B, C, D, and XY. This can be verified by looking at the equation as a whole and noting the order in which the values appear formally when scanning from left to right.
`+0×L(B:B×C+A×D)`	Stores the imaginary part of the product $x \cdot y$ in B.
`+L(A:G(R))=XY`	Stores the real part of the product $x \cdot y$ in A since the original value of A is no longer needed. GET is used with R since it is an

intermediate variable and is not to appear in the menu. Notice that all the LET functions except this one are multiplied by zero. Thus, when XY is solved for, the Solver uses a direct solution method (XY appears formally only once) and effectively reduces the equation to XY = R. When XY is solved for, the real part of the product is displayed and the real and imaginary parts of the product are stored in A and B respectively. Notice that C and D are left unchanged.

Since the real part of the product is returned as the value of XY, this eliminates having to press RCL A after every calculation to see the real part of the product. To see the imaginary part, press RCL B. An application with several complex number functions has been developed in Chapter 2 of this book using the ideas m this equation.

Evaluation Order

As your Solver equations become increasingly more sophisticated, you may find that using LET and GET takes a bit of forethought to ensure that the Solver assigns and recalls values in the order you intended. When calculating an unknown, your calculator effectively rearranges the equation and either isolates the variable in question and solves for it directly, or uses an iterative process. During rearrangement, the simple left-to-right order of evaluation may be disturbed.

For example, when

```
G(X)+Y=0×L(X:4+X)+4
```

is solved for Y, it is not obvious if the GET or LET is performed first. Actually, the Solver performs the LET before the GET in this equation.

Most ambiguities in using LET and GET can be avoided by observing the following guidelines:

1. Place all LET and GET functions on the same side of an equal sign.

2. Try to group the variable(s) you are calculating on the other side of the equals sign.

When these guidelines are followed, you can assume that the simple left-to-right evaluation process occurs.

Forcing Iteration

Although the idea of forcing iteration was introduced in your owner's manual, it bears repeating here. An equation such as $1 = \sin(x)$ can be solved directly for x, but there are an infinite number of solutions to this equation given by $x = \pm(2 \cdot n + 1) \cdot \pi/2$ for $n = 0, 1, 2, ...$

The Solver will find the root corresponding to the principle value of the sine function. In general, the trigonometric functions operate using the principle value. If you are interested in a root that is not a principle value, you can re-write the equation in a mathematically equivalent form that forces an iterative solution. This allows you to enter initial guesses instructing the Solver to look for a root between the two bounds. The equation above can be entered as the Solver equation

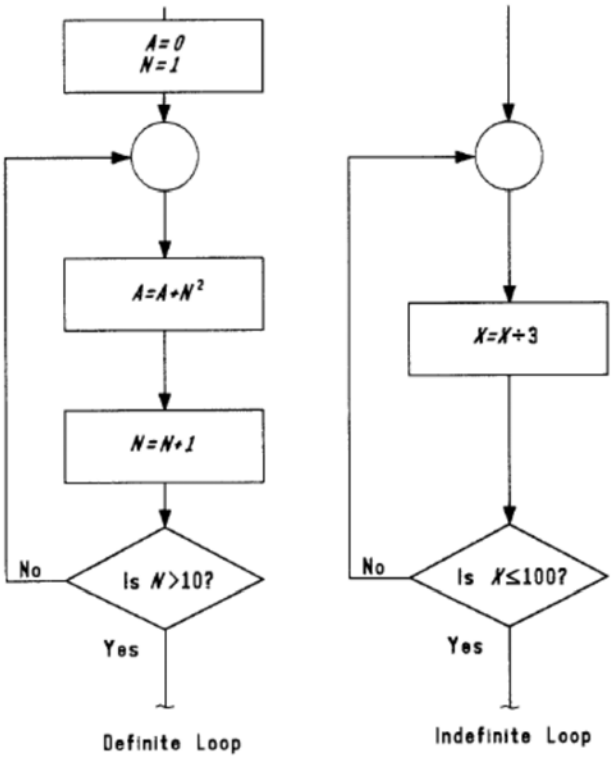
```
1=0×X+SIN (X)
```

This forces an iterative search since X appears formally twice.

More on Using the Σ Function

Definite and Indefinite Loops

A loop is a technique used by computer programmers to repeat a certain section of instructions a number of times before continuing to other instructions. Often, the loop is executed a fixed number of times (a definite loop), while other times the loop repeats indefinitely until a certain condition is met. The second type of loop is referred to as an indefinite loop.



The definite loop illustrated in the figure will generate the sum of the squares of the integers 1 through 10. Notice that A is 0 initially, the loop counter N is 1 initially, and the loop counter is incremented by 1 with each pass through the loop. The definite loop is performed a fixed number of times (10 in this case). On the other hand, the indefinite loop repeats indefinitely until the desired condition (X<=100) is met. By using the Σ function you can effectively include definite loops in your equations. In fact, the Σ function was designed to operate as a definite loop. Indefinite loops may also arise. While you are not able to construct a true indefinite loop for reasons that are explained below, you can effectively simulate one.

Simulating an Indefinite Loop

The Σ function is defined as follows:

```
Σ(cv:c1:c2:s:alg)
```

where the algebraic expression (alg) is evaluated and summed for values of the counter variable (cv). The counter variable starts with value c1 and is incremented in steps of s to a final value of c2. When the Σ function is first encountered in an equation, the Solver stores the step size s and the counter variable's initial and final values c1 and c2 in a special location in memory not accessible to the user. Any attempt to alter the values of s, cv, c1, or c2 using the LET function causes the Solver to create separate variables of the same name. Since the value of these variables cannot be changed, a loop cannot be prematurely exited. This is precisely what makes construction of a true indefinite loop impossible. However, an indefinite loop can be simulated as shown in the next example.

Example 14: An Indefinite Loop

To simulate the indefinite loop shown in the previous figure, the Σ function can be performed until the desired condition (X<=100) is met. Then the loop can simply add zeroes to this result on subsequent passes until the final value of the counter variable is reached. To avoid having too few or far too many loop repetitions, a way is needed to determine in advance the maximum number of loops necessary to meet the desired condition and to assign this value to c2. For the example at hand, we must find the number of times n that X must be divided by 3. This is given by the equation

$$\frac{X_{initial}}{3^n} < 100 \ .$$

If this is solved for n, we obtain the maximum number of loops needed to obtain the desired result (X<=100). Rearranging and taking the logarithms of both sides we have

$$\frac{X_{initial}}{3^n} < 100 \quad \rightarrow \quad \ln\left(\frac{X_{initial}}{100}\right) = n \cdot \ln\left(3\right) \ .$$

Solving for n, we obtain the final result

$$\mathrm{n} = \frac{\ln\left(\frac{X_{initial}}{100}\right)}{\ln\left(3\right)} \ .$$

This value of n is the value for c2 that guarantees sufficient passes through the loop. The Solver equation is shown below.

Equation	Comments
A=	The variable we will calculate.
Σ(N:1:	The counter variable is N and is set initially to 1.
IF(X<=300:1: LN(X/100)/LN(3)+1):	This is the final value of the counter variable. The conditional check made by the IF insures that at least 1 loop will be performed (if X is less than 300, n is less than 1). In the event that X>300, the result for n derived above is used with a small

change: here we have added a `1` to the result. If this had not been done, the loop would only be performed `n-1` times instead of n times.

```
1:
```

This is the step value; i.e., `N` is incremented by `1` each time the loop is repeated.

```
IF(X<=100 AND N<>1:
0:0×L(X:X/3)))
```

The body of the loop. If `X` is less than or equal to `100` and it is not the first pass through the loop (`N<>1`), a zero is added to the loop and `X` remains unchanged. If `X` is greater than `100` or `N=1` (first pass), the current value of `X` is divided by `3` and this result is assigned as the new value of `X`. Notice that the `LET` function is multiplied by zero causing the `Σ` function to have a value of zero. The 3 closing parentheses are needed to complete the `LET` function, the `IF` function, and the `Σ` function respectively.

```
+X
```

This term simply adds the final value of `X` to the value of the `Σ` function (which is zero as noted above) leaving the effective result `A=X`. This final value of `X` is returned as the solution for `A`.

A practical use of this Solver technique can be found in the application "Greatest Common Divisor and Least Common Multiple" in Chapter 2 of this book.

Using Trigonometric Functions

Equations involving trigonometric functions often demand that the variables be in radians rather than degrees. For example, in a branch of mathematics known as Fourier Transforms, the sinc function arises and is defined as

$$\operatorname{sinc}\left(x\right) = \frac{\sin\left(x\right)}{x} \ .$$

Here, x must be in radians; however, it is often desirable to enter x in either radians or degrees. A convenient way to accomplish this is with a conditional check, illustrated in the following Solver equation:

```
SINC=IF(SIN(90)=1:SIN(X)/RAD(X):SIN(X)/X)
```

Notice that the conditional check is true only when the calculator is in degrees mode. Although you must be aware of what mode the calculator is in when entering numbers in this Solver equation, this technique eliminates the need to always set radians mode and the need for two separate Solver equations (one for degrees and one for radians).

Note

The sinc function has the indeterminate value $0/0$ at $x = 0$. By a technique of calculus known as L'Hopital's Rule, the sinc function can be shown to approach 1 as x approaches 0. Thus, the sinc function is defined as 1 at $x = 0$. To give a correct result for $x = 0$, the above Solver equation can be modified slightly to:

```
SINC=IF(X=0:1:IF(SIN(90)=1:SIN(X)/RAD(X):SIN(X)/X))
```

In Conclusion

Although it is unlikely that you will want to use every application in this book, they represent operations that arise frequently in science and engineering. For this reason, you will probably want to keep several applications in your calculator's memory. To give yourself plenty of room to store and “CALC” the application Solver equations, we recommend that you delete the example equations created in this chapter after you have worked through them. Recall from your owner's manual that Solver variables are “remembered” by the calculator for use in moving from one Solver equation to another. These variables consume a significant amount of calculator memory and should be periodically reviewed and deleted as described in your owner's manual.

Summary of Functions for the Solver in the HP 17/18/19 Calculator Family

Martin Hepperle, May 2015

17B 17BII 17BII+	18C	19BII	27S	Function Group	Ayntax	Purpose
x	x	x	x	math	ABS (A)	Return the absolute value of A
		x	x	math	ACOS (A)	Return the arcus cosine of A. Also known as inverse cosine of A.
		x	x	math	ACOSH (A)	Return the arcus hyperbolic cosine of A. Also known as the inverse hyperbolic cosine of A.
x	x	x	x	math	ALOG (A)	Return the common antilogarithm (base 10) of A
		x	x	math	ANGLE (A : B)	Return the polar coordinate angle of the point (A : B) given in rectangular coordinates.
		x	x	math	ASIN (A)	Return the arcus sine of A. Also known as the inverse sine of A.
		x	x	math	ASINH (A)	Return the arcus hyperbolic sine of A (inverse hyperbolic sine of A).
		x	x	math	ATAN (A)	Return the arcus tangent of A (inverse tangent of A).
		x	x	math	ATANH (A)	Return the arcus hyperbolic tangent of A (inverse hyperbolic tangent).
		x	x	math	COMB (A : B)	Return the number of combinations of A items taken B at a time.
		x	x	math	COS (A)	Return the cosine of A.
		x	x	math	COSH (A)	Return the hyperbolic cosine of A.
		x	x	math	DEG (A)	Return A converted from radian to decimal degrees.
x	x	x	x	math	EXP (A)	Return the natural antilogarithm (base e) of A
x	x	x	x	math	EXPM1 (A)	Return e^A - 1 : one less than the natural antilogarithm (base e) of A
x	x	x	x	math	FACT (A)	Return the factorial of A. Mathematical notation: A!.
x	x	x	x	math	FP (A)	Return the fractional part of A
x	x	x	x	math	IDIV (A : B)	Return the integer part of the quotient of A/B.
x	x	x	x	math	INT (A)	Return the greatest integer less than or equal to x.
x		x	x	math	INV (A)	Return the inverse of x; 1/A.
x	x	x	x	math	IP (A)	Return the integer part of A.
x	x	x	x	math	LN (A)	Return the natural (base e) log of A.
x	x	x	x	math	LNP1 (A)	Return the natural (base e) log of (A+1).
x	x	x	x	math	LOG (A)	Return the common (base 10) log of A.
x	x	x	x	math	MAX (A : B)	Return the larger of A and B.
x	x	x	x	math	MIN (A : B)	Return the lower of A and B.
x	x	x	x	math	MOD (A : B)	Return the remainder of the division A/B. MOD(A:B) = A - B * INT(A/B)
		x	x	math	PERM (A : B)	Return the permutations of A items taken B at a time.

x	x	x	x	math	PI	Return the number pi: 3.14159265359
		x	x	math	RAD (A)	Return A converted from decimal degrees to radians.
		x	x	math	RADIUS (A : B)	Return the polar coordinate radius of the point (A : B) given in rectangular coordinates.
		x	x	math	RAN#	Return the a pseudo-random number (within 0…1).
x	x	x	x	math	RND (A : B)	Return A rounded to B decimal places if 0 <= B <= 11. Return A rounded B significant digits if - 12 <= B <= -1. B must be an integer.
x	x	x	x	math	SGN (A)	Return the sign of A: +1 if A > 0, 0 if A = 0, -1 if A < 0.
		x	x	math	SIN (A)	Return the sine of A.
		x	x	math	SINH (A)	Return the hyperbolic sine of A.
x		x	x	math	SQ (A)	Return the square of A; A^2.
x	x	x	x	math	SQRT (A)	Return the square root of A.
		x	x	math	TAN (A)	Return the tangent of A.
		x	x	math	TANH (A)	Return the hyperbolic tangent of A.
x	x	x	x	math	TRN (A : B)	Return A truncated to B decimal places if 0 <= B <= -11. Return A to B significant digits if -12 <= B <= -1. B must be an integer.
		x	x	math	XCOORD (A : B)	Return the x-coordinate of point given in polar coordinates with radius A and angle B. Uses the current angle unit.
		x	x	math	YCOORD (A : B)	Return the y-coordinate of a point given in polar coordinates with radius A and angle B. Uses the current angle unit.
x	x	x	x	conditional operators	A > B	Return TRUE if A higher than B
x	x	x	x	conditional operators	A < B	Return TRUE if A lower than B
x	x	x	x	conditional operators	A = B	Return TRUE if A equal to B
x	x	x	x	conditional operators	A >= B	Return TRUE if A greater or equal to B
x	x	x	x	conditional operators	A <= B	Return TRUE if A lower or equal to B
x	x	x	x	conditional operators	A <> B	Return TRUE if A not equal to B
x	x	x	x	logical operators	A AND B	Return TRUE if both (A and B) are TRUE
x	x	x	x	logical operators	A OR B	Return TRUE if one or both of (A or B) are TRUE
x	x	x	x	logical operators	A XOR B	Return TRUE if only one of (A or B) but not both are TRUE (in 18C manual incorrectly named EXOR)
x	x	x	x	logical operators	NOT A	Return TRUE if A is FALSE
x	x	x	x	logical operators	IF (A : B : C)	Return B if A is TRUE, else C. Note: some expressions may need + in front of A to be valid.
x		x	x	logical operators	S (A)	Return TRUE if solving for the variable named A. Can be used inside the IF() function to combine related equations into one Solver menu.
x		x	x	date	CDATE	Return the current date
x		x	x	date	CTIME	Return the current time. For the format see MAIN->TIME->SET->HELP.
x	x	x	x	date	DATE(A:B)	Return the date B days after (when B is positive) or before (when B is negative) date A. The format for A is set in the TIME/SET menu.
x	x	x	x	date	DDAYS (A : B : C)	Return the number of days between dates A and B. Formats for A and B are set in the TIME menu; C designates the calendar: C = 1 for the actual calendar, which recognizes leap years, C = 2 tor the 365-day calendar, which ignores leap years, C = 3 for the 360-day calendar, which uses 12, 30-day months.
x		x	x	date	HMS (A)	Return A in decimal hours converted to HH.MMSS format.
x		x	x	date	HRS (A)	Return A in HH.MMSS format converted to decimal hours.
x		x	x	statistics (SUM lists)	ITEM (A : B)	Return the value of item number B from the sum-list A.
x		x	x	statistics (SUM lists)	∑ (A : B : C : D : E)	Return the summation of the algebraic expression E for values of the counter A, stepping from B to C at increments of D.

x		x	x	statistics (SUM lists)	SIZES (A)	Return the number of items in the specified SUM list A.
		x		TVM functions	N (A : B : C : D : E : F)	Return the function N for TVM calculations.
		x		TVM functions	I%YR (A : B : C : D : E : F)	Return the function I%YR for TVM calculations.
		x		TVM functions	PV (A : B : C : D : E : F)	Return the function PV for TVM calculations..
		x		TVM functions	PMT (A : B : C : D : E : F)	Return the function PMT for TVM calculations.
		x		TVM functions	FV (A : B : C : D : E : F)	Return the function FV for TVM calculations.
x		x		financial (CFLO lists)	FLOW (A : B)	Return the value of the specified cash flow.
x		x		financial (CFLO lists)	SIZEC (A)	Return the number of the last flow in CFLO list A.
x	x	x	x	financial (CFLO lists)	SPPV (A : B)	Return the present value of a single $1.00 payment; equivalent to 1 / SPFV(A : B). B is the number of compounding periods. A is the interest rate per compounding period, expressed as a percentage.
x	x	x	x	financial (CFLO lists)	SPFV (A : B)	Return the future value of a single $1.00 payment; equivalent to (1 + A / 100%)^B. B is the number of compounding periods. A is the interest rate per compounding period, expressed as a percentage.
x		x		financial (CFLO lists)	#T (A : B)	Return the number of times that specified cash flow number B occurs in list A.
x	x	x	x	financial (CFLO lists)	USFV (A : B)	Return the future value of a uniform series of $1.00 payments; equivalent to (SPFV(A:B)-1)/(B/100). A is number of payments. B is the periodic interest rate, expressed as a percentage.
x	x	x	x	financial (CFLO lists)	USPV (A : B)	Return the present value of a uniform series of $1.00 payments; equivalent to USFV(A:B)/SPFV(A:B). A is number of payments. B is the periodic interest rate, expressed as a percentage.
(x)		x	x	assignments	L (A : B)	Return the value of B and perform the assignment A = B. Defines the value of variable A to be B. B can be a number or a variable. Returns the value set, i.e. B. (Note: not implemented in the first generation golden 17bII+, but in the later larger silver 17bII+)
(x)		x	x	assignments	G (A)	Return the current value of A. (Note: not implemented in the first generation golden 17bII+, but in the later larger silver 17bII+)

Some examples of creative usage of the sum function:

Replacement for missing trigonometric functions:

Determine the cosine of an angle θ (in radian) by a series using the `SUM` (Σ) function

```
COS=1+Σ(I:2:12:2:((MOD(I/2:2)×-2)+1)×(θ^I)/FACT(I))
```

Determine the sine of an angle θ (in radian) by a series using the Σ function

```
SIN=Σ(I:1:13:2:((MOD((I-1)/2:2)×-2)+1)×(θ^I)/FACT(I))
```

Determine the tangent of an angle θ (in radian) by a series using the Σ function

```
TAN=Σ(I:1:13:2:((MOD((I-1)/2:2)×-2)+1)×(θ^I)/FACT(I)) /
(1+Σ(I:2:12:2:((MOD(I/2:2)×-2)+1)×(θ^I)/FACT(I)))
```

Other Snippets

A compact way to set three variables `A`, `B`, `C` to zero

```
L(A:L(B:L(C:0)))
```

To assign a local variable and not use the result in the equation multiply by zero

```
L(A:12)×0
```

A trivial example how to use local variables `X` and `Y` to solve the equation `A+B=0. A×0` and `B×0` are required to allow for solver to find solutions for `A` and `B`, otherwise he finds no solution.

```
L(X:A)×L(Y:B)×0+G(X)+G(Y)+A×0+B×0
```

Integration

Determine the area `S` under the curve $y = x^k$ in the interval from `X0=0` to `X1=1`.

The analytical solution by integration of the function yields the accurate result `S = 1/(k+1)×(X1-X0)^(k+1)`.

The area can be approximated by the sum of `N` strips each having the width `dx = (X1-X0)/N`.

Each strip is centered at `X0+dx/2, X0+3/2×dx`, up to the last one at `X1-dx/2`

```
S = Σ ( I : 0 : N-1 : 1 : (X0+(I+0.5)×(X1-X0)/N)^K ) × (X1-X0)/N
```

Using the `L()` and `G()` functions available on some calculators one could pre-calculate the strip width `(X1-X0)/N` and store it in a local variable `DX`.

Conversion of Numbers from different Bases

Numbers in decimal or lower base (not hexadecimal) can be converted with the following solver equation

```
ANS = N + (FROM − TO) × Σ ( I : 0 : LOG(N)/LOG(TO) : 1 :
                            L(N:IDIV(N:TO)) × FROM^I )
```

ANS returns the conversion of `N` from the base `FROM` to the target base `TO`. The two bases must be lower or equal to 10 so the conversion to binary or octal forms are the most useful application.

Example:

`10 > FROM`	(from decimal)
`8 > TO`	(to octal)
`16 > N`	(number to convert, will be reduced to zero during evaluation)
`ANS = 20`	(result: 16 in octal)

Note that this equation works in the older 17B and 17BII but not in the later HP 17bII+. This is because the input variable `N` is modified by the `LET` function inside the sum and the optimized solver of the HP17B+ executes the equation at least twice (reducing `N` to zero during the first evaluation) and finally finds … `SOLUTION NOT FOUND`.

However, if a new variable `N0` is created and initialized to the preset value of `N` first (using the `LET` function and nullifying the result) and then used in place of `N` inside the equation the algorithm also works in the HP 17BII+.

```
ANS = LET(N0:N)×0 + N +
      (FROM − TO) × Σ ( I : 0 : LOG(G(N0))/LOG(TO) : 1 :
                                L(N0:IDIV(G(N0):TO)) × FROM^I )
```

The GET function is used to suppress the appearance of the variable N0 in the solver menu.

Even if the equation is executed twice by the solver of the HP 17BII+, the variable N is not changed and N0 is always set properly to N before the sum is evaluated.

Source: Don Shepherd, Thomas Klemm HP-Forum [http://www.hpmuseum.org/forum/thread-1660.html?highlight=base+conversion].

Other Stuff

MoHPC *The Museum of HP Calculators*

HP Articles Forum

[Return to the Index]
[Previous | Next]

---

programming using the HP 17bii+ solver

*Posted by **Don Shepherd** on **14 Apr 2007, 2:48 p.m.***

The HP 17bii+ calculator has several characteristics that might make it interesting to members of this community:

- it fits in your shirt pocket nicely
- it has a 2-line display
- it has 6 soft function keys
- it has user-definable menus
- it has a very uncluttered keyboard
- it has useful finance and business functions
- it can print using a small infrared printer
- you can use either RPN or algebraic mode
- it has 12-digit accuracy
- it has 30,740 bytes of user memory
- it is programmable

- it is still being manufactured!

I became interested in this model after reading somewhere that it was programmable. I determined that it is not "keystroke" programmable like its financial brother, the venerable 12c and its successors. But it is programmable via the built-in equation solver application.

I imagine that the original inventors of the equation solver never intended that it would be used as a general purpose "programming" language, just as the inventors of the 12c cash flow functions probably never imagined that anyone would use those financial functions for indirect addressing. But the great thing about HP calculators is that both those things are possible. And members of this community have always been willing to share insights and experiences.

So I bought the 17bii+ just to see what it could do, from a programming perspective. I knew that every programming language must support three structures: sequence, decision, and iteration (loops). So I began to look at the solver. Yep, it can do sequences (especially when you put "`0×`" next to things you don't want to be included in the solution).

Yep, it can do decisions (`IF (condition : do if true : do if false)`) And yep, it can do loops (sigma function). And it can assign values to named variables (load function) and use those variables (get function) in calculations. The solver cannot use the 10 registers, but since you can create your own variable names, that is not a shortcoming. The solver cannot execute subroutines, GOTO a label or line number, or exit a loop early, and this makes it unacceptable as a general purpose programming language. But I found that you can still do some useful things with it.

For instance, you can sum the digits of a number:

```
sod = sigma(i:0:log(n):1:mod(ip(n):10)+0xL(n:n/10))
```

You can determine if a number is prime or not (a result of 0 means the number is prime, any other number is the first factor of the input number):

```
fact = 0xL(j:0)+
if(mod(n:2)=0:2:sigma(i:3:sqrt(n):2:
if(mod(n:i)=0:if(g(j)=0:0xL(j:1)+i:0):0)))
```

You can calculate a standard mod-10 checkdigit (maximum 12 input digits):

```
ckdig = 0xL(m:2) ×L(c:-1)
+mod(10-
mod(sigma(i:0:log(n):1:
0xL(a:mod(ip(n):10) ×g(m))
+if(g(a)<10:g(a):g(a)-9)
+0×L(m:g(m)+g(c))
×L(c:-g(c))
×L(n:n/10)):10):10)
```

You can convert from decimal to binary:

```
dec2bin = sigma(i:0:11:1:mod(n:2)×(10^i)+0×L(n:ip(n/2)))
```

Or from binary to decimal:

```
bin2dec = sigma(i:0:log(n):1:mod(ip(n):10)×2^i+0×L(n:n/10))
```

Or from decimal to octal:

```
dec2oct = sigma(i:0:11:1:mod(n:8)×(10^i)+0×L(n:ip(n/8))
```

I tried to carefully type the equations above, but I'm not perfect. If you try one of these on your 17bii+ and can't get it to work, post a message on the forum and I'll help.

Now, I admit that I wouldn't attempt to do a payroll program on the 17bii+, and the lack of scientific functions would make this calculator inappropriate for many members of this community. Nevertheless, I think it is great that an application designed to solve equations can, in this case, be used for some general purpose programming tasks.

HP calculators are great!

Source Don Shepherd

http://www.hpmuseum.org/cgi-sys/cgiwrap/hpmuseum/articles.cgi?read=712

Another tip from Don Shepherd on using the S (solve for) function

The S Solver function lets you take a separate path depending upon which variable you are solving for. Take the quadratic formula, for example. There are normally two solutions to a quadratic equation, lets call them `X1` and `X2`. This Solver equation lets you input values for variables `A`, `B`, and `C`, then solve for either `X1` or `X2`.

```
QUAD: 0×L(D:SQRT(SQR(B)-4×A×C))+
      IF(S(X1):(-B+G(D))/(2×A)-X1:
               (-B-G(D))/(2×A)-X2)
```

$$QUAD:0\cdot L\left(D:SQRT\left(SQR\left(B\right)-4\cdot A\cdot C\right)\right)+IF\left(\begin{matrix}S\left(X1\right):\left(-B+G\left(D\right)\right)/\left(2\cdot A\right)-X1\\ :\left(-B-G\left(D\right)\right)/\left(2\cdot A\right)-X2\end{matrix}\right) \quad (1)$$

$$D = \sqrt{B} - 4 \cdot A \cdot C$$
if solving for X1:
$$\left(-B + G\left(D\right)\right) / \left(2 \cdot A\right) - X1$$
else
$$\left(-B - G\left(D\right)\right) / \left(2 \cdot A\right) - X2$$
end if

(2)

This equation also uses the trick to pre-calculate the common expression "`D`" and then to multiply the result of the `L()` by zero so that it gets executed but does not affect the result. The `IF()` block is straightforward and contains two branches depending on whether we solve for `X1` or for `X2`. Both use the value of "`D`"

I'll discuss two things: ending a sigma loop early, and using the S (solving for) function.

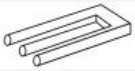
Don Shepherd on Ending a Loop Early

According to the Technical Applications guide, you can't. But user Mike Ingle made me aware of a method to do this many years ago, and it works very well in limited situations where you don't need to do further processing after you exit the loop. You achieve this by using the following logic. Within the loop, test for a condition that would normally cause you to leave the loop early. If the test is true, store (via the Lfunction) the value of interest (normally the loop index but it might be something else) in a menu variable and then do a divide by 0 or Log(0) to force the equation to error terminate. When this happens, the calculator beeps and SOLUTION NOT FOUND is displayed. But you can then RCL the variable that you stored the value of interest in. Sometimes you might want to store two values in two variables, like I do below in the Solver equation for the Happy number, the subject of the RPN programming contest at HHC 2017.

See this link for the Happy equation (post 29): http://www.hpmuseum.org/forum/thread-908...ng+contest

09-19-2017, 12:23 AM (This post was last modified: 09-19-2017 07:34 PM by Don Shepherd.)

Post: #29



Don Shepherd
Senior Member

Posts: 483
Joined: Dec 2013

RE: HHC 2017 RPN Programming contest information and results thread

Never having used a 41, I wanted to implement the problem using the 17bii solver. Here is the equation:

Code:

```
HAPPY:HAP+CYC+
ANS=SIGMA(I:1:100:1:
L(N:SIGMA(J:0:LOG(NUM):1:
SQ(MOD(IDIV(NUM:10^J):10))))
+L(NUM:G(N))+
IF(NUM=1 OR NUM=4:L(HAP:NUM)+L(CYC:I)/0:0))
```

This is not a normal solver equation, it takes advantage of the fact you can exit a loop by inducing an error in the calculation (divide by 0 works well) after saving whatever values you want in variables that you can then RCL. So, to run this, enter your desired number to test and press NUM. Then press ANS (solve for ANS). After a few seconds it will beep and display SOLUTION NOT FOUND. Just RCL HAP (4=unhappy, 1=happy) and RCL CYC for the number of cycles.

Another example of leaving a loop early is this 17b Solver equation that finds the prime factors of a number. This equation is much faster than the prime factorization equation in the Technical Applications guide. Enter the number you want to factor as N. Then solve for FACT. If n is prime, it will just say FACT=number, otherwise it will find the first factor and display SOLUTION NOT FOUND. RCL FACT to see the factor that was found. Then press FACT to find the next factor, and RCL as before when it beeps. On the last prime factor found, it won't beep, that is the signal that it is done.

PRIMEF:IF(MOD(N:2)=0:L(FACT:2)+L(N:N/2)/0:sigma(I:3:SQRT(N):2:IF(MOD(N:I)=0:L(FACT:I)+L(N:N/I)/0:0))+N)-FACT

$$PRIMEF:IF\left(\begin{matrix}MOD\left(N:2\right)=0:L\left(FACT:2\right)+L\left(N:N/2\right)/0\\:\sum\left(\begin{matrix}I:3:SQRT\left(N\right):2\\:IF\left(MOD\left(N:I\right)=0:L\left(FACT:I\right)+L\left(N:N/I\right)/0\right)\\:0\end{matrix}\right)+N\end{matrix}\right)-FACT$$

(3)