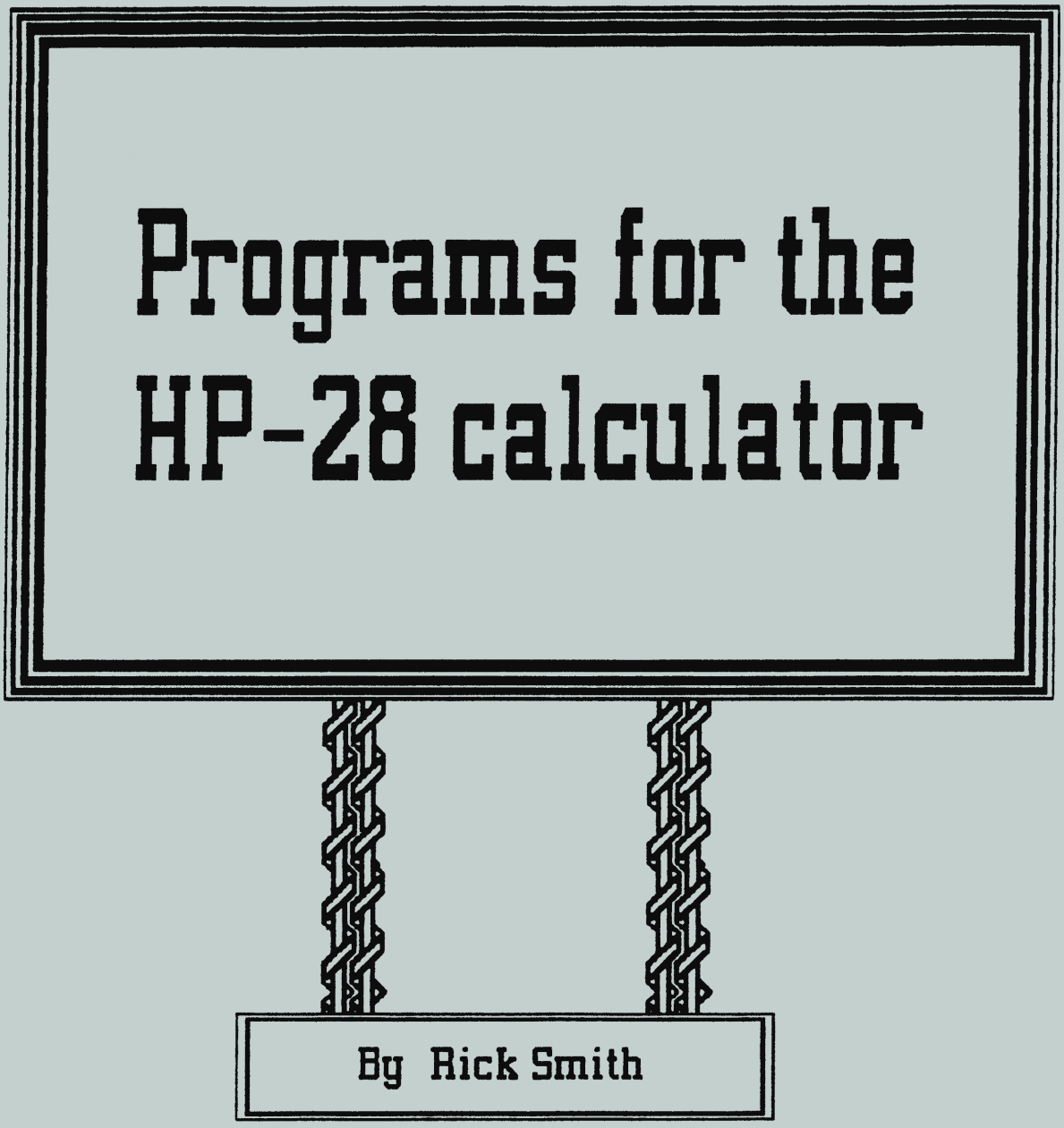
Programs for the HP-28 calculator



By Rick Smith

TABLE of CONTENTS

DISCLAIMER NOTICE:

The author makes no warranty or guarantees of any kind regarding the material contained in this book.

Determinations of accuracy, performance, proper function and useage is the sole responsibility of the user.

The author of this book shall not be liable for consequential and/or incidental damages in connection with the useage, performance or the furnishing of these programs and written material.

Contents of this book are subject to change without notice.

(*)- **Registered Trademarks:**

Hewlett-Packard, HP and HP-28 are trademarks of Hewlett-Packard Co.

A Brief Introduction into Programming the HP-28*

The programs in this book use the more fundamental commands of the HP-28 language. These are the loop commands **START-NEXT** and **FOR-NEXT** and the **IF-THEN-END** command.

Before we talk about how these commands work, let's discuss a few fundamental points.

From the point of view of the calculator, anything enclosed in the double-arrow brackets « and » is considered a program. For example **« "Hello, world" 1 DISP »**. This uses the **DISP** (display) command to print Hello, world at the top of the screen.

Suppose you want to write a program that takes any number given to it and multiplies it by 5 then subtracts 3. The first thing you need to do is to be able to input or "grab" a number. This is done with the → command (the shifted U on the left side keyboard). This command is used in conjunction with what are called "local variables" like this: **→ a** . The "a" is the local variable. Local variables differ from "global" variables (the other kind) in that they are only used for one calculation or procedure (which HP calls a defining program but "procedure" is more to the point) and then they are thrown away. By contrast, a global variable is stored under a name in a **USER** menu. For example, if you put the number 10 on the stack and did this: **'A' STO** , under the **USER** menu you would see [A] for a function key. This "global" variable doesn't disappear unless you **PURGE** it! This brings up another point; notice that the lower case letters are used for local variables and the upper case letters are used for global variables. This is done by convention so you know which are which.

Anyway, back to our program. We can write it three ways:

```
« → a  'a * 5 - 3'  »
     or
« → a  « a 5 * 3 - » »
     or
« → a « 'a * 5 - 3' EVAL » »
```

After typing in one of the programs you would store it under your favorite name. Then you could put any number on the stack and press the function key labeled with the name you chose (under **USER**) and, presto!, the result will appear. One thing you may have noticed is that while the first program is "readable" the second program looks like a mistake. Thats because the first program uses an *expression* which is anything enclosed in single quotes. An *expression* is written just the way you would write it on paper. Since the expression in the first program comes after the inputted variable with no other brackets, it is evaluated immediately. The second program just carries out the button pushing as if you did the function manually on the calculator and remember it's *Reverse polish notation*. The third program looks like the first program except for the extra set of brackets. If you did not have the EVAL command in the third program, the program would just put 'a * 5 - 3' on the stack unevaluated.

As a rule of thumb, when you input a number into a variable, you must use « » around whatever follows that uses the variable. Like this:

```
« → a b c
   « Whatever! »   program grabs three numbers off
»                  the stack.
```

This way you don't have to worry about using the local variable more than once. For example, the following program won't work:

```
« → k 'k + 5' 'k - 2' »
```

The first expression is evaluated, but then the local variable is gone! So the second expression is put on the stack as-is. However, the following program does work:

```
« → k « 'k + 5' EVAL 'k - 2' EVAL » »
```

The inside set of brackets defines everything inside of them as one procedure. Therefore, the local variable is good for everything inside them.

Notice that the programs in this book are laid out in a way so you can see the *structure* or organization of the program better. You don't have to type them in to *look* like this (you probably couldn't). The calculator will squash them up anyway after you **ENTER** the program. You only need to pay attention to spaces around command words. For example, →k is wrong it should be → k. Also, → ARRY is wrong, it should be →ARRY.

Program comments that are on the right side of program listings in this book are in italics. These are comments ONLY! Don't type them into the calculator.

Now for the basic loops. Lets look at the **START-NEXT** loop. It has the form of:

```
a b START
      stuff
   NEXT
```

The *stuff* is any set of commands you want repeated. *a* and *b* are the beginning and ending numbers for the loop. For example if you wanted to go through the loop 3 times *a and b* would be 1 and 3, 10 and 12, 22 and 24, etc.

The **FOR-NEXT** loop is the same except you can define a local variable which is given the value of the present loop. The value (an integer) of the present loop is called the index.

Example:

```
« 1 10 FOR x
          'x * 2' EVAL
       NEXT
»
```

The variable of your choice is defined when it is placed directly after the **FOR** statement, in this case **FOR x**. Notice that x is used in the expression 'x*2' and evaluated 10 times, putting 10 numbers on the stack.

You can change the increment of the index from 1 *(default when using* **NEXT**) to any number you choose by replacing **NEXT** with ***increment number*** **STEP**. For example, **2 STEP** causes 2 to be added to the index. You can also have the loop count from a high number down to a low number by having *a* > *b* and using a negative increment number. **STEP** can also replace **NEXT** in the **START-NEXT** loop.

The other important statement is the **IF-THEN-END** statement. It has the form:

```
IF something
THEN
   stuff
END
```

The *something* is a statement that tests something to see if it's true. For example, **IF y > 5**. If y is greater than 5 (True) then carry out the *stuff*. If y is not greater than 5 (False) then continue program right after the **END** statement.

There is also a variation called **IF-THEN-ELSE-END** in which the form is:

```
IF something
THEN
   stuff
ELSE
   different stuff
END
```

In this case the *different stuff* is carried out if the *something* is false.

Many of the programs in this book use the **IFERR-THEN-END** statement (If error). It has the form:

```
IFERR
  stuff
THEN
  different stuff
END
```

If the *stuff* commands are carried out normally, the program continues after the END statement (skips the *different stuff*). If an error occurs while carrying out *stuff*, then the *different stuff* commands are executed. The IFERR statement is used in some of the programs in this book as a way to display a help message:

```
IFERR
  program with an input.
THEN
  display help screen
END
```

This is very handy, because frequently you forget what a program does or what variables are required for input. As structured above, you could just hit the program key and the program would error-out because it was expecting a numeric input, and would then display the help screen.

You will find that the other loops and branch commands are variations and combinations of the above forms. Hopefully, the programs in this book will help you to learn how to program the HP-28 by example and give you a sense of how much time and busy work can be saved by writing useful programs.

Quadradic formula program.

The quadradic formula takes coefficients from a polynomial of the form:

$$ax^2 + bx + c = 0$$

and gives the roots (zeros) of the equation.

Example use (program name: [QUADR])

for $2x^2 + 3x + 4$.

Press **[QUADR]**.
Type **2,3,4** on command line.
Press **[CONT]**.
The program displays the answer.

program listing [QUADR] — *Comments*

```
« CLEAR  CLLCD                          Clear stack.
" Quadradic Formula " 1 DISP            Prompt user.
" Enter a,b,c, then " 2 DISP
" press continue " 3 DISP
HALT
→ a b c
  « '(-b + √ (b^2-4*a*c))/(2*a)' EVAL
    → y
   «
    IF y IM 0 > THEN
      CLLCD
      " Imaginary roots are" 1 DISP
      y RE 2 DISP
      " + or - i" 3 DISP
      y IM 4 DISP
    ELSE
      CLLCD
      " Real roots are " 1 DISP
      '(-b - √ (b^2-4*a*c))/(2*a)'
      EVAL
      → z
      « y 2 DISP
        " and " 3 DISP
        z 4 DISP
      »
    END
   »
  »
»                                       End of program.
```

Song program.

Make a list of the form { n tone duration tone duration ....} where n is the number of tone-duration pairs. The following program will play the song.

Example use (program name [PLAY])

Put list in level 1 (see example below).
Press **[PLAY]**

program listing	*comments*
« DUP	*Duplicate list.*
1 GET → s	*Grab n from list.*
« 2 1 s START	
GETI → a	*Grab tone.*
« GETI → d	*Grab duration.*
« 'a' EVAL 'd' EVAL BEEP »	*play tone.*
»	
NEXT	*Repeat until end of list.*
»	
»	*The End.*

example songs:

{ 11 560 .4 560 .3 560 .1 560 .4 660 .3 620 .1 620 .3 560 .1 560 .3 525 .1 560 .5 }

{ 15 440 .4 580 .3 580 .05 580 .4 650 .4 730 .3 730 .05 730 .6 730 .1 650 .1 730 .1 765 .4 550 .4 650 .4 580 .4 }

Linear Algebra Programs

The matrix in the following programs is stored in the global variable X.

Row operation program

This program performs the row operation
$g*ROW_h + ROW_i \text{ ---> } ROW_j$.
(g,h,i,j is sequence on command line.)

Example use (program name [RWOP])

Put a matrix on stack, level 1:

```
[[ 1 3 6 4 ]
 [ 3 -4 2 2 ]
 [ 1 2 3 4 ]
 [ 2 0 3 4 ]]
```

Type in **-2,3,4,4** on command line.

Press **[RWOP]**.

This is: -2 times row 3 plus row 4 and put result in row 4.

program listing [RWOP]

```
« → g h i j
  « 'X' STO
      X SIZE LIST→ DROP    → r c
    « 1 c FOR d
           X { h d} GET g *
         NEXT
         { 1 c } →ARRY
       1 c FOR d
           X { i d } GET
         NEXT
         { 1 c } →ARRY +
         'Y' STO
       1 c FOR d
           Y { 1 d } GET
           X { j d } ROT PUT
         'X' STO
         NEXT
         X
         'Y' PURGE
      »
    »
 »
```

Row Swap Program

Swaps two rows.

Example use (program name [RRSW])

Put matrix on stack, level 1.
Type in 2,4 on command line.
Press [RRSW].
This swaps rows 2 and 4.

program listing [RRSW]

```
«
  → h i
     «
        'X' STO
        X SIZE LIST→ DROP  → r c
         «
           1 c FOR d
                 X {i d} GET
               NEXT
                 {1 c} →ARRY
                 'Y' STO
           1 c FOR d
                 Y {1 d} GET
                 X {h d} GET
                 X {h d} 4 ROLL PUT
                 {i d} 3 ROLL PUT
                 'X' STO
               NEXT
               X
               'Y' PURGE
           »
      »
»
```

Row Multiply/Divide program

Multiplies (divides) a row by a number.

Example use (program name [RMLT] ([RDIV]))

Put matrix on stack, level 1.
Type in **-6,3** on command line.
Press **[RMLT]**. ([RDIV])
This multiplies (divides) row 3 by -6.

program listing

```
«
   → g h
      «
         'X' STO
         X SIZE LIST→ DROP  → r c
         «
           1 c FOR d
                 X {h d} GET g *     <<NOTE: change * to
               NEXT                    / for [RDIV]
               {1 c} →ARRY               program.
               'Y' STO
           1 c FOR d
                 Y {1 d} GET
                 X {h d} ROT PUT
                 'X' STO
               NEXT
               X
               'Y' PURGE
           »
       »
»
```

To make the two different programs: type in the program listing above; store as **[RMLT]**; then **RCL** and **EDIT** it, and replace the * with / ; then store this new version as **[RDIV]**.

Synthetic Division Program

A polynomial:

$$f(x) = a_n x^n + a_{n-1} x^{n-1} + ... + a_1 x + a_0$$

has possible rational roots equal to:

$$\pm \frac{\text{factors of } a_0}{\text{factors of } a_n}$$

Use synthetic division on the equation f(x) to determine if each possible root produces a remainder of 0 when divided into f(x).

The remaining roots are true factors of the "residue equation" that the previous true factor produced after division.

```
        [-2]   3   14   14   -8   -8
             +.    -6  -16    4    8
residue eq.-> 3    8   -2   -4    0 <--- shows that -2
(not including 0)                     is a root of
                                   3x⁴+14x³+14x²-8x-8.
```

Example use (program name [SYNTH])

Enter the coefficients of the polynomial as a vector in level 1.

ie [3 14 14 -8 -8]

Type in the root you what to try -2 .
Press [SYNTH].

The program puts the residue equation and remainder on the stack, and stores the original vector in X.

```
ie      3:
        2:      [ 3,8,-2,-4 ]
        1:                  0
```

If the remainder is 0 (a root is found), then [DROP] the zero, enter the next possible root, and press [SYNTH].

If the remainder is NOT zero (not a root) then [CLEAR] the stack and press [X] to bring up your original vector. Then try next possible root.

program listing [SYNTH]

```
«
   → c
        «
           'X' STO
            X SIZE LIST→ DROP  → j
            «
                    X 1 GET DUP c *
              2 j FOR i
                    X i GET +
                    DUP
                    c *
                NEXT
                DROP  → r
                 «
                    DEPTH
                    →ARRY
                    r
                 »
              »
           »
»
```

Data Bank Programs

The following programs should be put in their own sub-directory. A sub-directory can be made ,for example, by typing:

```
'DATA'              Name of sub-directory
[CRDIR]             Command "Create Directory"
                    found under MEMORY menu.
```

[DATA] will now be a key in the USER menu. Press [DATA] to get into the DATA sub-directory. The user keys go blank, indicating that there is nothing in the sub-directory DATA. Now type the following:

```
<< HOME >> [ ENTER ]     Command HOME returns you
'RTN'                    to the HOME directory.
[STO]
```

Now you have [RTN] for a user key. When you press [RTN], it returns to the HOME or Root directory. This is just a simple way to pop back and forth between your sub-directories and the HOME directory.

The Data bank programs scroll through a list of data. Create a list like the following:

```
{ "Bob 123-4567" "Joe 234-5678"  "Fred 567-3456"
  "John 444-5555"  "Carol 555-2323" }
```

Then store the list under the name 'Q' :

```
'Q'
[STO]
```

The programs also uses an index number which is stored under the name 'N'. So you need to create that also:

```
1 [ ENTER ]          Start out with an index of 1.
'N'
[STO]
```

Now the only thing left is the following 2 programs which will scroll up and down the list:

Program: [UP]

```
« Q SIZE 1 -
   → s
  « IF N s >
      THEN s 1 +
       'N' STO
       "no more" 1 DISP
      ELSE Q N 1 + DUP
       'N' STO GET 3 DISP
       Q N 1 - GET 2 DISP
       Q N 2 - GET 1 DISP
     END
   »
»
```

Program: [DOWN]

```
« IF N 4 <
    THEN
     3 'N' STO
     "no more" 3 DISP
    ELSE
     Q N 1 - DUP
     'N' STO GET 3 DISP
     Q N 1 - GET 2 DISP
     Q N 2 - GET 1 DISP
   END
»
```

Classical Mechanics

In some statics problems you are working with a 3-space cartesian coordinate system and force vectors at various positions. The clue to solving a system of forces in equilibrium is that the sum of all forces acting on point is equal to zero.

Enter vectors like 3i - 4j + 5k in HP vector brackets like this [3 -4 5]. That way, adding 2 vectors or multipling the vector by a scalar is easy.

Vector magnitude program

When you have a force vector out in space and have the coordinates for the head and the tail, one of the first things you can do is find the *Position Vector* which is the head coordinates minus the tail coordinates. So for example, you would enter the head coordinates [4 3 0] then the tail coordinates [3 5 8], then subtract. The result [1 -2 -8] is the *Position Vector* i -2j -8k. In order to use this to show the direction of a force, you need to make it a *Unit Vector* or a vector with magnitude = 1. You do this by simply dividing the *Position vector* by it's magnitude or length.

The following program finds the magnitude of a 3-space vector:

Example use: (program name: [MAG3])

Enter the 3-space vector on level 1.

```
3:
2:
1:          [1 -2 -8]
```

Press [MAG3]: *Result:*

```
3:
2:          [1 -2 -8]
1:     8.30662386292      This is the magnitude of
                            the vector.
```

Now to get the *Unit Vector* just divide.

```
3:
2:
1:    [.120385853086 -.240771706171 -.963086824686]
```

This is the *Unit vector*:

.120385853086i -.240771706171j -.963086824686k

program listing [MAG3]

```
« DUP 1 GET SQ → a
  « DUP 2 GET SQ → b
    « DUP 3 GET SQ
       a + b + √          <Remember, this is RPN!
    »
  »
»
```

Coordinate direction angle program

When you have a unit vector representing the direction of a force vector whose tail is at the origin, you can find the *Coordinate Direction Angles* or the angles the vector makes with the x, y, and z axes. These angles are defined as:

alpha = Cos^{-1}(i of unit vector)
beta = Cos^{-1}(j of unit vector)
gamma = Cos^{-1}(k of unit vector)

Example use (program name: [CDA])

Enter the unit vector on level 1.

```
3:
2:
1:   [.120385853086 -.240771706171 -.963086824686]
```

Press [CDA].
The result is:

```
3:
2:  [.120385853086 -.240771706171 -.963086824686]
1:  {  83.0856282278
       103.932091549
       164.383870595  }
```

On level 2 is your unit vector and on level 1 is the coordinate direction angles. This says that the vector makes a 83 degree angle with the X axis, a 103.9 degree angle with the Y axis, and a 164.38 degree angle with the Z axis.

program listing [CDA]

```
« DUP 1 GET ACOS → a          ‹Duplicate vector, get 1st
  « DUP 2 GET ACOS → b      entry and take arccos
    « DUP 3 GET ACOS → c    of it, and put it in "a".
      « a b c 3 →LIST »
» » »
```

Interpolation program

When using tables (ie. steam tables or trig tables), many times the value your looking for is in-between two other values. For example:

Steam table for saturated H_2O

Temp. degC	*Press. kPa*	*etc.....*
165	*0.7005*	
170	*0.7917*	

If you need the pressure at 167.5 degrees, you need to interpolate between the values. This would be:

P = 0.7005 + (167.5-165)/(170-165) * (0.7917-0.7005)

= 0.7461 kPa

Example use (program name: [INTPO])

```
Press [INTPO].
Type 165,.7005      Low lookup, related value
     170,.7917      High lookup, related value
     167.5          Number you're trying to lookup
Press [CONT]
```

The program displays interpolated value for 167.5.

program listing [INTPO]

```
« CLLCD                                  ‹Clears LCD.
  "Enter LowLup,LLval"  1 DISP           ‹Promts user.
  "    HighLup,HLval"   2 DISP
  "    YourLup      "   3 DISP
  "Then hit [CONT]"     4 DISP
  HALT                                ‹Halts the program
  → a a1 b b1 x                        to allow user
    'a1 + (x-a)/(b-a) * (b1-a1)'       to enter data.
»
```

Timer program

Example use (program name [TIMER])

Enter starting time in seconds, press **[TIMER]**. Timer counts down and sounds alarm. Then press any key to stop alarm. Note: may not be extremely accurate but you can play around with a constant in the program to make it more accurate if you like.

program listing

```
« IFERR                          ‹ Set up error detection.
     → s
        «
          s 0 FOR t
               CLLCD "Timer running" 1 DISP
               t →STR    3 DISP
               "seconds" 4 DISP
               .890 WAIT              ‹Play with this
          -1 STEP                     number to adjust
          51 CF                       accuracy.
          DO
               2000 .1 BEEP
               .1 WAIT
               2000 .1 BEEP
               .5 WAIT
          UNTIL KEY END
          51 SF
        »
   THEN
     CLLCD
     "Enter seconds" 1 DISP
     "Press [TIMER]" 2 DISP
   END
»
```

Lotto program

The lotto program picks 6 random numbers from 44. You can adapt this program for different lottos.

Example use (program name [LOTTO])

Press [LOTTO].

The program displays 6 numbers.

program lising

```
« 1 6 FOR X
     RAND 44 *
     0 FIX RND STD
    NEXT
    6 →LIST →STR
    DUP SIZE 2 -
    2 SWAP SUB
    2 DISP
    "Lotto numbers" 1 DISP
    "    " 3 DISP
»
```

Interest program

Given the principal, interest rate, number of times a year compounded, and number of years, this program calculates the total amount.

Example use (program name [INTEREST])

Enter principal amount. (ex. 10,000.00)
Enter interest rate. (ex. .085)
Enter number of times a year compounded.(ex. 12 times a year.)
Enter number of years left to accumulate.(ex. 5 years)

Press [INTEREST]

result:

15273.00 dollars.

program listing

```
« IFERR
    → p r n t
     « 'p*(1+r/n)^(n*t)' EVAL »
   THEN
    "Enter P R N T" 1 DISP
   END
»
```

Plot programs

The following programs make plotting a little easier.

Program [Draw]

This program takes an equation in the form of an expression off level 1, draws it, and stores the screen in global variable 'PT'. This same plot can then be instantly recalled later with the program [REDW].

program listing [Draw]

```
« IFERR
     'EQ' STO CLLCD
     DRAW LCD→ 'PT' STO
     PT →LCD DGTIZ
   THEN
     DROP
     "Need equation on stack" 1 DISP
   END
»
```

program listing [REDW]

Instantly redraws plot stored in 'PT'.

```
« PT →LCD DGTIZ »
```

program listing [FAST]

For fast, low resolution plots.

```
« 2 RES »
```

program listing [SLOW]

For slow, high resolution plots.

```
« 1 RES »
```

More Data Bank programs

These programs are simular to the previous data bank programs except these programs "page-flip" instead of scroll. Also when the program reaches the end of the list, it returns to the top (or bottom) and continues. The only catch is that when you make up your list of items, you must have a total number of items that are divisable by 3 (since it displays 3 at a time). This is no problem though, since you can just put in empty strings to take up the slack (i.e. " ").

Program name [Next] (Note this name must have some lower case letters to distinguish from the NEXT command word.)

listing

```
«   IFERR
          N 3 +
          'N' STO
          1 3 FOR r
               Q N r +
               GET r DISP
          NEXT
     THEN
          0 'N' STO
          CLEAR
          1 3 FOR r
               Q N r +
               GET r DISP
          NEXT
     END
»
```

Program name [PREV]

listing

```
« IFERR
     N 3 -
     'N' STO
     1 3 FOR r
          Q N r +
          GET r DISP
     NEXT
   THEN
     Q SIZE 3 - 'N' STO
     CLEAR
     1 3 FOR r
          Q N r +
          GET r DISP
     NEXT
   END
»
```

Example list of items for previous programs:

```
{ "Joe Jones"
  "123 Main St."
  "ph. 123 4567"
  "Norman Bartfaster"
  "2345 E. West St."
  "ph. 345 5678"
  "This space for sale"
  "  "
  "ph. 345 2345"
  "The End"
  "  "
  " "
}
```

Note the use of empty strings to fill out the "pages" and in particular to finish the last page. Remember you must have three items for each page.

Probability

Flip a coin until you get a head, and count the number of times, R, that you flipped. R could be 1, 2, 3, 4, and so on. Obviously, on the first flip, the probability of getting a head is .5. The chance of having to flip twice (R=2) to get a head is .5 x .5 or $(.5)^2$. You might guess that for an experiment like this the probability of getting a head after R flips is $(.5)^R$. Where R is the number of flips it took.

Probability Density Function of a random variable of the discrete type is denoted by f(x). It is the *probability that an event occurs.*
It is also denoted by P(X=x), where X is the random variable and x is the specific number R, ex. P(X=2). Therefore in the experiment above:

$$p.d.f. \quad f(x) = P(X=x) = (.5)^x .$$

Now, what if you needed to know the probabily of getting a head in 3 flips or less, ie. $P(X \leq 3)$. This is the cummulative sum of all the probabilities less than or equal to three, and is formally called the *Cummulative Distribution Function (c.d.f.).* The c.d.f. is denoted by F(x) and is the following:

$$F(x) = P(X \leq x) = \sum_{t=1}^{x} f(t)$$

Note that x can only be an integer, *(discrete variable type).*

Summation Program

Performs the following:

$$\sum_{x=a}^{b} f(x)$$

Example use (program name [ΣFX])

Enter f(x) on stack as an expression.

```
3:
2:
1:      '.5^X'        note: capital letter X
```

Enter the bounds, a and b, you want the expression summed over.

```
3:      '.5^X'
2:           3              This is a.
1:           5              This is b.
```

Press [ΣFX]

result:

```
3:
2:       '.5^X'      <- left on stack
1:       .21875      <- This is the answer
```

program listing: ([ΣFX])

```
«  IFERR                          <Set up error detection
     → a b                        to display a help screen.
      «  0 'ANS' STO
        a b FOR x
            x 'X' STO
            DUP →NUM
            'ANS' STO+
          NEXT
      »
      ANS 'X' PURGE
      'ANS' PURGE
    THEN
      CLLCD
      "Enter 'expression' in" 1 DISP
      "terms of X, then enter" 2 DISP
      "summation bounds a,b" 3 DISP
    END
»
```

Expected Value of X Program

(Mean of X)

The expected value of X or E(X) is defined as:

$$\sum_{x=a}^{b} x*f(x)$$

program listing: ([ΣXFX])

```
«  IFERR
     → a b
      «  0 'ANS' STO
        a b FOR x
            x 'X' STO
            DUP →NUM X * ‹Difference from summing
            'ANS' STO+          program.
          NEXT
      »
      ANS 'X' PURGE
      'ANS' PURGE
    THEN
      CLLCD
      "Enter 'expression' in" 1 DISP
      "terms of X, then enter" 2 DISP
      "summation bounds a,b" 3 DISP
    END
»
```

Binomial Distribution

The binomial distribution is a function given by:

$$P(a \le x \le b) = \sum_{k=a}^{b} [k!/(k!(n-k)!)]p^k(1-p)^{n-k}$$

With the mean = np, and variance = np(1-p), where n is the number of Bernoulli trials, and p is the probability of success.

Example use

The probability that a part fails during the first year of operation is 0.05. What is the probability that out of 10 parts, between 3 and 5 are defective?

In this case n = 10, p = 0.05, a = 3, and b = 5.

Type **10,.05,3,5** on command line.

Press **[BIN]**

result:

```
3:
2:
1:      1.15008027966E-2
```

program listing [BIN]

```
« IFERR
     → n p a b
        « 0 'ANS' STO
           a b FOR y
                n y COMB
               'p^y*(1-p)^(n-y)' EVAL *
               'ANS' STO+
           NEXT
        »
     ANS 'ANS' PURGE
  THEN
   "→ n,p,a,b" 1 DISP          <Display help screen if
  END                           error.
»
```

Poisson Distribution

The Poisson distribution is a function given by:

$$P(a \leq x \leq b) = \sum_{k=a}^{b} (L^k e^{-L})/k!$$

where L = np = mean. The approximation works well when the probability of success, p, is small and the number of trials, n, is large.
(The greek letter lambda is usually used in place of L.)

Example use

The probability that a part is defective is 0.02. what is the probability that out of 200 parts, between 10 and 15 parts are defective?

In this case lambda = np = 200(0.02) = 4, a = 10, and b = 15.

Type **4,10,15** on command line.

Press **[POISS]**

The answer is about .008127.

program listing [POISS]

```
« IFERR
     → la a b
        « 0 'ANS' STO
           a b FOR k
              'la^k*e^(-la)' →NUM
              k FACT /
              'ANS' STO+
           NEXT
        »
     ANS 'ANS' PURGE
  THEN
   "→ la,a,b" 1 DISP        <Help screen.
  END
»
```

Normal Distribution

Unlike the previous distributions which were discrete types, the normal distribution is a continuous type. The normal distribution with mean = 0 and variance = 1 is given by:

$$P(Z < z) = I(z) = \int_{-infinity}^{z} [e^{-z^2/2}]/\sqrt{(2\pi)}$$

and is denoted N(0,1).

Example use

Whats the probability of Z<.7?

Type .7 on command line.

Press **[NRM]**

answer: .758

program listing [NRM]

```
« IFERR
     → b
       « '.39894228*EXP(-X^2/2)'
          { X 0 b }
          .0001 ∫ DROP .5 +
          4 FIX RND STD            <rounds answer to 4
        »                        decimal places.
    THEN
     " → b " 1 DISP
    END
»
```

Another version of this program can take upper and lower bounds and compute the probability.
(i.e. P(a ≤ X ≤ b))

program listing [NRM2]

```
« IFERR
     → a b
       « '.39894228*EXP(-X^2/2)'
          { X a b }
          .0001 ∫ DROP
          4 FIX RND STD
        »
    THEN
     " → a,b " 1 DISP
    END
»
```

Pooled Variance

Pooled variance is used in many Statistical methods such as confidence intervals and hypothesis tests. The following program computes this.

program listing

```
« IFERR
     → s1 s2 n1 n2
        «  '((n1-1)*s1+(n2-1)*s2)/(n1+n2-2)'
           →NUM DUP 3 DISP
           "Pooled variance =" 2 DISP
        »
   THEN
       "Pooled variance" 1 DISP
       "Enter var1,var2,n1,n2" 2 DISP
   END
»
```

Parallel resistance program

Example use (program name: [P.RES])

Enter the values of resisters.

470,1000,100,390,560 [ENTER]

Enter the number of resisters you entered.

5 [ENTER]

Press **[P.RES]**

The equivalent resistance of (in this case) the 5 resisters is displayed.

program listing [P.RES]

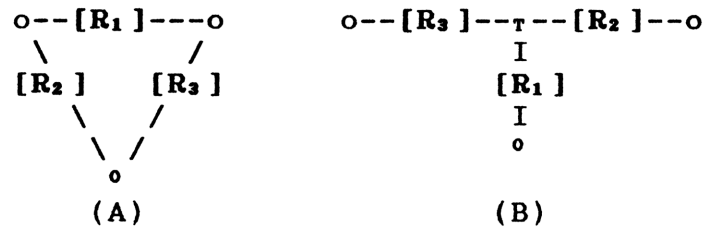
```
«  → d                    Grabs the "number of
   « 1 d START       resisters" number.
       INV
       d ROLL
     NEXT
     1 d 1 - START            This loop adds up all the
      +                       inverted numbers.
     NEXT
     INV                 Inverts the final value to give
   »                   answer.
»
```

Delta to Y network transformation

The resister network (A) is called a delta network. It can be transformed into a Y network ,(B), with the following formulas:

$R_{1Y} = R_2 * R_3 / (R_1 + R_2 + R_3)$
$R_{2Y} = R_1 * R_3 / (R_1 + R_2 + R_3)$
$R_{3Y} = R_1 * R_2 / (R_1 + R_2 + R_3)$

The R's on the right side of equation are from the delta network.



Notice the pattern of the transformed resisters.

Example use (program name: [D.Y])

Enter R_1 , R_2 , and R_3 from delta network on stack.

```
3:      470          R1
2:      100          R2
1:      180          R3
```

Press [D.Y]

The program returns the R_1 , R_2 , and R_3 for the Y network.

```
3:        24         R1
2:     112.8         R2
3:  62.66667         R3
```

program listing [D.Y]

```
« → a b c
   « 'b*c/(a+b+c)' EVAL
      'a*c/(a+b+c)' EVAL
      'a*b/(a+b+c)' EVAL
   »
»
```

Y to Delta network transformation

This program does the reverse of the Delta to Y program. See Delta to Y program for more details.

program listing name: [Y.D]

```
«  → a b c
   «  '(a*b+b*c+c*a)/a'  EVAL
      '(a*b+b*c+c*a)/b'  EVAL
      '(a*b+b*c+c*a)/c'  EVAL
   »
»
```

Angular frequency and frequency conversion programs

The following program converts angular frequency (rad/s) to frequency (Hz).

program listing: name: [W→F]

```
«  → w
         « 'w/(2 * π)' →NUM »
»
```

The following program converts frequency (Hz) to angular frequency (rad/s).

program listing: name: [F→W]

```
«  → f
         « '2 * π * f' →NUM »
»
```

Complex Impedence converter programs

This program converts a resistance into it's complex impedence Z_R. Input: resistance (ohms).

Example: enter 1000 for a 1k resistor.
Example output: (1000,0)

This is the complex number 1000 + 0j.

program listing: name: [RESI]

```
« IFERR
   → r
      « r 0  R→C  »
  THEN
     " Need resistance " 1 DISP
  END
»
```

The following program converts a capacitance into it's complex impedence Z_c. Input: angular frequency (rad/s), capacitance (F).

program listing: name: [CAP]

```
« IFERR
   → w c
      « w c * INV NEG 0 SWAP  R→C  »
  THEN
     " Need angular freq.," 1 DISP
     "       capacitance " 2 DISP
  END
»
```

The following program converts an inductance into it's complex impedence Z_L. Input: angular frequency (rad/s), inductance (H).

program listing: name: [IDUCT]

```
« IFERR
      → w L
         « w L * 0 SWAP  R→C  »
  THEN
     " Need angular freq.," 1 DISP
     "      inductance " 2 DISP
  END
»
```

Evaluating Boolean Expressions

The HP-28 can evaluate boolean expressions. The following example describes the process.

Suppose you have a 3 input function with the following output:

A	B	C	F
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	0

The boolean expression from the truth table is:

A'B'C + A'BC' + ABC'

To verify this expression, the values of A, B, and C must first be stored in global variables.

Looking at A, B, and C on the truth table from bottom to top we have:

A = 11110000
B = 11001100
C = 10101010

Go into the binary menu and select [BIN] mode.

Enter the numbers (must use the pound symbol) and store each one under A, B, and C respectfully.

Then you can use the AND OR and NOT operators (also under binary menu) to evaluate the expression. For the example above, you would key in:

A [NOT] B [NOT] C [ENTER] [AND] [AND]
A [NOT] B [ENTER] C [NOT] [AND] [AND] [OR]
A [ENTER] B [ENTER] C [NOT] [AND] [AND] [OR]

The result should be the output of the truth table 01000110.

RLC circuit programs

Damping factor and undamped angular frequency^2 for RLC circuits.

This program calculates two important variables used in determining the natural and step responses of basic parallel and series RLC circuits. They are the damping factor α (or neper frequency) and the undamped angular frequency ω_o (or resonant radian frequency). This program calculates ω_o^2 not ω_o.

example use (program name [RLC])

The program takes four values R, L, C and x. The x is a 0 or 1 which indicates to the program whether the circuit is a series RLC or parallel RLC. x is 0 for parallel, and x is 1 for series.

If, for example R=400 ohms, L=.04H, C=1uf are the componet values for a parallel RLC circuit, enter the values of R, L and C on stack in this order.

400, .04, 1E-6 [ENTER]

Now enter a zero on the stack to indicate a parallel circuit.

0 [ENTER]
Press **[RLC]**

The program displays the value of α and ω_o^2.

program listing: name: [RLC]

```
« IFERR
     → r L c x                        <- Note: L is local
     « 'r*INV(2*L)'  EVAL → a
       «  'INV(2*r*c)'  EVAL → b
          «  'INV(L*c)'  EVAL → c
             « IF x 1 ==
               THEN
                 "Series RLC" 1 DISP
                 "alpha=" a →STR + 2 DISP
                 a c
               ELSE
                 "Parallel RLC" 1 DISP
                 "alpha=" b →STR + 2 DISP
                 b c
               END
               "wo^2=" c →STR + 3 DISP
   » » » »
  THEN
     "Need R,L,C,x" 1 DISP
     "where x=0 for parallel," 2 DISP
     "x=1 for series RCL." 3 DISP
  END
»
```

Determining expressions for basic RLC circuits.

Natural and step responses for parallel and series RLC circuits all have the same basic functions for 3 cases. The cases are overdamped, underdamped, and critically damped.

For the overdamped case:

$$f(t)= A_1 e^{s_1 t} + A_2 e^{s_2 t}$$

$$f(0)= A_1 + A_2$$

$$df(0)/dt= S_1 A_1 + S_2 A_2$$

For the underdamped case:

$$f(t)= e^{-at}[B_1 \cos(\omega_d t)+B_2 \sin(\omega_d t)]$$

$$f(0)= B_1$$

$$df(0)/dt= \omega_d B_2 - aB_1$$

For the critically damped case:

$$f(t)= e^{-at}[D_1 t + D_2]$$

$$f(0)= D_2$$

$$df(0)/dt= D_1 - aD_2$$

Note: $a = \propto$ (damping factor)

f(t) can be v(t) or i(t) for the natural response and for the step response the final value, v_f or i_f, is added to the expression

This program takes the values of $\propto$ and ω_o^2 and determines if the response is overdamped, underdamped, or critically damped and calculates the parameters S_1 and S_2 if overdamped, $\propto$ (a) and ω_d if underdamped, and $\propto$ (a) if critically damped. These values can then be used in the f(t) expressions above and the coefficients (A_1, A_2, B_1, etc.) can then be determined from initial conditions.

example use (program name: [SSAW])

You have a series RLC circuit with R=560 ohms, L=.1H and C=.1uf. The initial conditions of V_{cap}=100v and i=0 at t=0 is given. Find the natural response i(t) for t>0?

First, α and ω_o^2 is determined by using the [RLC] program.

Type **560, .1, .1E-6, 1** on command line.

Press **[RLC]**

The display shows:

Series RLC
alpha=2800
wo^2=100000000

Now press **[SSAW]**

The display shows:

Underdamped. alpha=
2800
and wd=
9600

This tells you that the response is underdamped so you'll have to use the underdamped function:

$$i(t) = e^{-\alpha t}[B_1 \cos(\omega_d t)+B_2 \sin(\omega_d t)]$$

Substituting α and ω_d:

$$i(t) = e^{-2800t}[B_1 \cos(9600t)+B_2 \sin(9600t)]$$

B_1 and B_2 can then be determined from initial conditions.

program listing name: [SSAW]

```
« IFERR → a w
     « '-a+√(a^2-w)' EVAL → y
          « IF y IM 0 >
            THEN
               CLLCD
               "Underdamped. alpha=" 1 DISP
               y RE NEG 2 DISP
               "and wd=" 3 DISP
               y IM 4 DISP
            ELSE
               '-a-√(a^2-w)' EVAL → z
               « IF y z ==
                 THEN
                    "Critically damped." 1 DISP
                    "alpha=" 2 DISP
                    z NEG 3 DISP
                 ELSE
                    "Overdamped. S1=" 1 DISP
                    y 2 DISP
                    "and S2=" 3 DISP
                    z 4 DISP
                 END
               »
            END
          »
     »
  THEN
    "Need alpha, wo^2" 1 DISP
  END
»
```

Quickies

Program name [ON] *Turns on sound. Input: none.*

« 51 CF »

Program name [OFF] *Turns off sound. Input: none.*

« 51 SF »

Program name [DEGK] *Converts degrees Celsius to degrees Kelvin. Input: number representing degrees in Celsius.*

« 273.15 + »

Program name [MODE] *Toggles between standard mode and fixed, 2 decimal place mode. Input: none.*

« **IF** 49 FC? **THEN** 2 FIX **ELSE** STD **END** »

Program name [→VECT] *Puts 3 numbers into a 3-space vector.*
Input: 3 numbers.

« 3 →ARRY »

Program name [ROT3] *Rotates 3 numbers on stack. Input: 3 number's on stack.*

« 3 ROLL 3 ROLL »

Program name [RTN] *Returns you to the home directory.*
Input: none.

« HOME »

Program name [KILO] *Multiplies a number by 1000.*
Input: a number on the stack.

« 1000 * »

Program name [MICR] *Divides a number by 10^6.*
Input: a number on the stack.

« .000001 * »