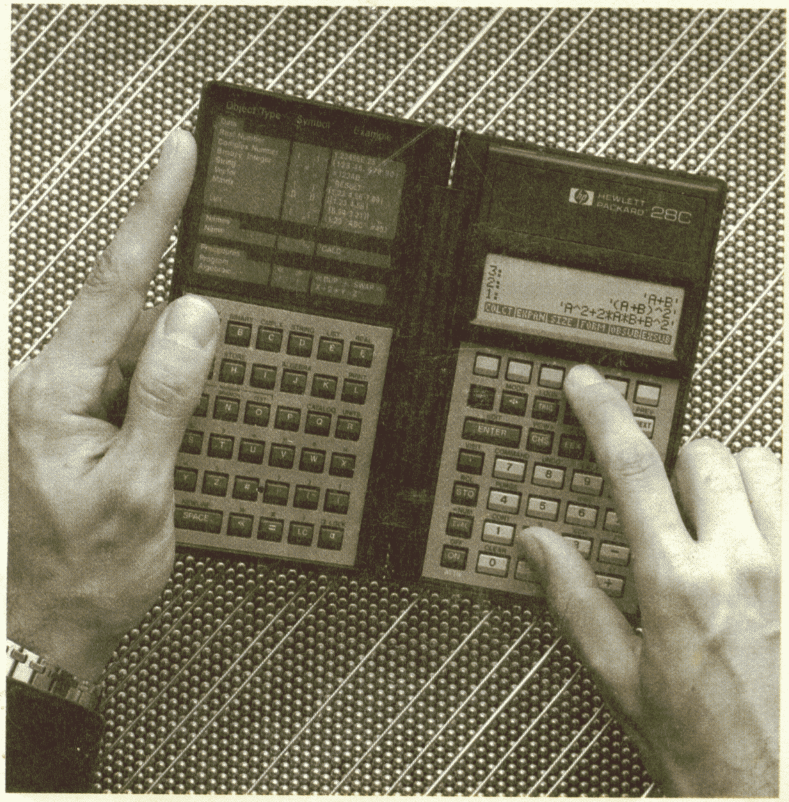
HEWLETT-PACKARD

HP-28C Programming Examples



Welcome to the HP-28C

This booklet, *HP-28C Programming Examples*, contains 19 program for your HP-28C. These programs are useful and, more importantly, they demonstrate a variety of programming techniques. You'll find a list of the techniques on page 6.

Before trying the examples in this booklet, please read "How To Use This Booklet" on page 7. It contains important information on the conventions observed in this booklet.

This booklet assumes you've read the *HP-28C Getting Started Manual*. At a minimum, you should know:

- How to enter numbers and expressions.
- How to enter programs and edit existing programs.
- How to use menus.

You can find detailed information about programming in the *HP-28C Reference Manual*, especially in the following sections.

- Programs
- PROGRAM BRANCH
- PROGRAM CONTROL
- PROGRAM TEST

HP-28C

Programming Examples

HEWLETT
PACKARD

Edition 1 June 1987
00028-90099

Portable Computer Division
1000 N.E. Circle Blvd.
Corvallis, OR 97330, U.S.A.

Printing History

Edition 1 June 1987 Mfg No. 00028-90099

Contents

List of Techniques

How To Use This Booklet

For each program you'll find the following information.

- A description of its purpose.
- A diagram showing its effect on the stack.
- A list of techniques that it demonstrates.
- A list of other programs that it requires (if applicable).
- A program listing with comments.
- An example that shows how to use it.

Each type of information is described in more detail below.

Stack Diagram. A stack diagram is a two-column table showing "Arguments" and "Results". "Arguments" shows what must be on the stack before the program is executed; "Results" shows what the program leaves on the stack.

Note that the stack diagram doesn't show everything; a program that changes user memory or displays objects might have no effect on the stack.

Techniques. This is the most interesting part. When you understand how a technique is used in this booklet, you can use it in your own programs.

Required Programs. Some programs call others as subroutines. You can enter the required programs and the calling program in any order, but you must enter all of them before executing the calling program.

The HP-28C can't hold all the programs in this booklet at one time. Before purging one program to make room for another, make sure the program you're purging isn't required by another program that interests you.

Program and Comments. This booklet formats the program listing to show a program's structure and process. You don't need to follow the format of the listing when you enter a program. However, be sure to key in spaces where they appear in the listing or between objects appearing on separate lines.

You can key in a program character by character, or you can use the menus to key it in command by command. It makes no difference as long as the result matches the listing.

When you key in the program you can omit all closing parentheses and delimiters *that appear at the very end of the program*; when you press [ENTER] the closing parentheses and delimiters are added for you.

Example. The examples observe the following conventions.

The illustrations assume STD display format. To select STD display format, press STD [ENTER] or use the MODE menu.

A box represents a key on the calculator keyboard.

[USER]	[ENTER]	[STO]
[ON]	[EVAL]	

In some cases a box represents a shifted key on the HP-28C. The shift key is *not* shown explicitly.

[CLEAR]	[CONT]	[VISIT]
[CTRL]	[BINARY]	[STAT]

The "inverse" highlight represents a menu label.

CLΣ, Σ+, and NΣ in the STAT menu.

SST and KILL in the CTRL menu.

DEC in the BINARY menu.

Variable names in the USER menu also appear as menu labels.

Menus typically include more than one menu level. Press [NEXT] and [PREV] to roll through the menu levels. In the examples, [NEXT] and [PREV] are *not* shown explicitly.

Programming Examples

The most important technique demonstrated in this booklet is *structured programming*: small programs used to build other programs. The following programs are used in other programs.

- BOXS is used in BOXR.
- MULTI is used in EXCO.
- PAD and PRESERVE are used in BDISP.
- ΣGET is used in ΣX2, ΣY2, and ΣXY.
- SORT and LMED are used in MEDIAN.

RENAME (Renaming a Variable)

Recall the contents of a variable, purge the variable, and store the contents in a new variable.

Arguments	Results
2: '*name*' *(old)* 1: '*name*' *(new)*	2: 1:

Techniques:

- Basic stack manipulations.

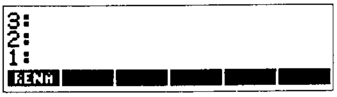
/

Program	**Comments**
«	Begin the program.
OVER	Copy the old name to level 1.
RCL	Recall the contents of the variable.
ROT	Move the old name to level 1.
PURGE	Purge the old variable.
SWAP	Put the contents and new name in the correct order.
STO	Create the new variable.
»	End the program.
[ENTER]	Put the program on the stack.
'RENAME [STO]	Store the program as RENAME.

Example. Create a variable A with contents 10, then rename A to B, then evaluate B to check that its value is 10.

Clear the stack and select the USER menu.

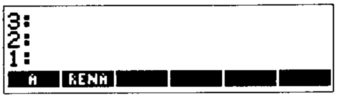
[CLEAR]
[USER]



```
3:
2:
1:
RENA
```

Create a variable A with contents 10.

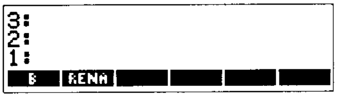
10 [ENTER]
'A [STO]



```
3:
2:
1:
 A   RENA
```

Rename variable A to B.

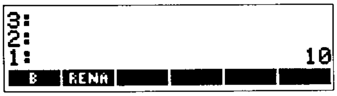
'A [ENTER]
'B [ENTER]
≣ RENAME ≣



```
3:
2:
1:
 B   RENA
```

Check the value of B.

≣ B ≣



```
3:
2:
1:                   10
 B   RENA
```

Box Functions

This section contains two programs:

- BOXS calculates the total surface area of a box.
- BOXR uses BOXS to calculate the ratio of surface to volume for a box.

BOXS (Surface of a Box)

Given the height, width, and length of a box, calculate the total area of its six sides.

Arguments	Results
`3:` *height*	`3:`
`2:` *width*	`2:`
`1:` *length*	`1:` *area*

Techniques:

- Local-variable structure. Local variables allow you to assign names to arguments without conflicting with global variables. Like global variables, local variables are convenient because you can use arguments any number of times without tracking their positions on the stack; unlike global variables, local variables disappear when the program structure that creates them is done.

  A local-variable structure has three parts.

  1. A command named "→". When you key in this command, remember to put spaces before and after it. (Like any command, → is spelled using normal characters and is recognized only when it's set off by spaces. Don't confuse this one-character command with delimiters like # or «.)
  2. One or more names.
  3. A procedure (expression, equation, or program) that includes the names. This procedure is called the *defining* procedure.

When a local-variable structure is evaluated, a local variable is created for each name. The values for the local variables are taken from the stack. The defining procedure is then evaluated, substituting the values of the local variables.

To appreciate the power of local variables, compare the version of BOXS given below with the version that appears on page 13.

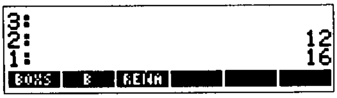
- User function. This type of program works in either RPN or algebraic syntax. A user function is a program with two characteristics: (1) It consists solely of a local-variable structure. (2) The defining procedure is an expression.

Program	Comments
`«`	Begin the program.
`→ h w l`	Create local variables for height, width, and length. By convention, lower-case letters are used. The values are taken from the stack (in RPN) or from the arguments to the user function (in algebraic syntax).
`'2*(h*w+h*l+w*l)'`	The defining expression for the surface area. Evaluating the user function causes evaluation of this expression, returning the area to the stack.
`»`	End the program.
[ENTER]	Put the program on the stack.
'BOXS [STO]	Store the program as BOXS.

Example. One of the advantages of user functions is that they work in either RPN or algebraic syntax. Calculate the surface of a box 12 inches high, 16 inches wide, and 24 inches long; make the calculation first in RPN and then in algebraic syntax.

For the RPN version, first enter the height and width.

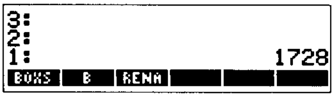
```
[USER]
12 [ENTER]
16 [ENTER]
```



```
3:
2:                    12
1:                    16
```

Then key in the length and execute BOXS.

`24` BOXS

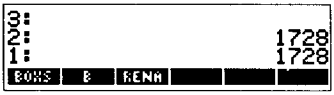


```
3:
2:
1:                1728
```

The surface area is 1728 square inches.

Now try the algebraic version.

`'BOXS(12,16,24` EVAL



```
3:
2:                1728
1:                1728
```

Again, the surface area is 1728.

BOXS Without Local Variables

The following program uses only stack operations to calculate the surface of a box. Compare this program with BOXS.

Arguments	Results
3: *height*	3:
2: *width*	2:
1: *length*	1: *area*

Program	Comments
«	Begin the program.
DUP2 *	Calculate wl.
ROT	Move w to level 1.
4 PICK	Copy h to level 1.
*	Calculate wh.
+	Calculate $wl+wh$.
ROT ROT	Move h and l to levels 2 and 1.
*	Calculate hl.
+	Calculate $wl+wh+hl$.
2 *	Calculate $2(wl+wh+hl)$.
»	End the program.

Because this version of BOXS isn't a user function, it can't be used in algebraic syntax.

BOXR (Ratio of Surface to Volume of a Box)

Given the height, width, and length of a box, calculate the ratio of its surface to its volume.

Arguments	Results
3: *height*	3:
2: *width*	2:
1: *length*	1: *area/volume*

Techniques:

- Nested user functions. BOXR is a user function whose defining expression uses BOXS in its calculation. In turn, BOXR could be used to define other user functions.

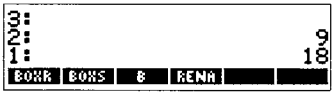
  Recall that BOXS was defined using h, w, and l as local variables, and note below that BOXS takes x, y, and z as arguments in the definition for BOXR. It makes no difference if the local variables in the two definitions match, or if they don't match, because each set of local variables is independent of the other. However, it's essential that local variables be consistent within a single definition.

Program	Comments
`«`	Begin the program.
`→ x y z`	Create local variables for height, width, and length. This program uses x, y, and z, rather than h, w, and l.
`'BOXS(x,y,z)`	Begin the defining expression with the user function BOXS.
`/(x*y*z)'`	Divide by the volume of the box.
`»`	End the program.
ENTER	Put the program on the stack.
`'BOXR` STO	Store the program as BOXR.

Example. Calculate the ratio of surface to volume for a box 9 inches high, 18 inches wide, and 21 inches long; make the calculation first in RPN and then in algebraic syntax.

For the RPN version, first enter the height and width.

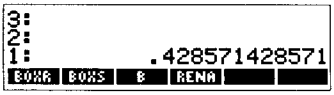
USER
9 ENTER
18 ENTER



```
3:
2:                9
1:               18
```

Then key in the length and execute BOXR.

21 BOXR

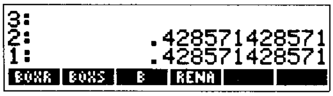


```
3:
2:
1:    .428571428571
```

The ratio is .428571428571.

Now try the algebraic version.

'BOXR(9,18,21 EVAL



```
3:
2:    .428571428571
1:    .428571428571
```

Again, the ratio is .428571428571.

Fibonacci Numbers

Given an integer n, calculate the nth Fibonacci number F_n, where

$$F_0 = 0, \quad F_1 = 1, \quad F_n = F_{n-1} + F_{n-2}$$

This section includes two programs, each demonstrating an approach to this problem.

- FIB1 is a user function that is defined *recursively* – its defining expression contains its own name. FIB1 is short, easy to understand, and usable in algebraic objects.
- FIB2 is a program with a definite loop. It's not usable in algebraic objects, it's longer and more complicated than FIB1, but it's faster.

FIB1 (Fibonacci Numbers, Recursive Version)

Arguments	Results
`1:` n	`1:` F_n

Techniques:

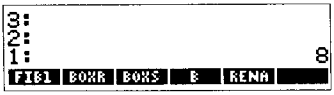
- User function. See the description on page 12.
- IFTE (If-Then-Else function). The defining expression for FIB1 contains the conditional function IFTE, which can be used in either RPN or algebraic syntax. (FIB2 uses the program structure IF . . . THEN . . . ELSE . . . END.)
- Recursion. The defining expression for FIB1 is written in terms of FIB1, just as F_n is defined in terms of F_{n-1} and F_{n-2}.

Program	**Comments**
`«`	Begin the program.
`  → n`	Define a local variable.
`  '`	Begin the defining expression.
`   IFTE(n≤1,`	If $n \leq 1$,
`   n,`	Then $F_n = n$;
`   FIB1(n-1)+FIB1(n-2))`	Else $F_n = F_{n-1} + F_{n-2}$.
`  '`	End the defining expression.
`»`	End the program.
ENTER	Put the program on the stack.
`'FIB1` STO	Store the program as FIB1.

Example. Calculate F_6 using RPN syntax and F_{10} using algebraic syntax.

First calculate F_6 using RPN.

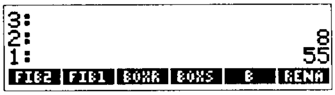
```
USER
6 FIB1
```



```
3:
2:
1:                    8
```

Next calculate F_{10} using algebraic syntax.

```
' FIB1 ( 10 EVAL
```



```
3:
2:                    8
1:                   55
```

FIB2 (Fibonacci Numbers, Loop Version)

Arguments	Results
1: n	1: F_n

Techniques:

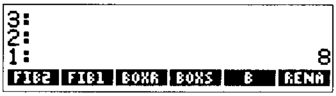
- Local-variable structure. See the description on page 11.
- IF . . . THEN . . . ELSE . . . END. FIB2 uses the program-structure form of the conditional. (FIB1 uses IFTE.)
- START . . . NEXT (definite loop). To calculate F_n, FIB2 starts with F_0 and F_1 and repeats a loop to calculate successive F_i's.

Program	**Comments**
`«`	Begin the program.
` → n`	Create a local variable.
` «`	Begin the defining program.
`  IF n 1 ≤`	If $n \leq 1$,
`  THEN n`	Then $F_n = n$;
`  ELSE`	Begin ELSE clause.
`   0 1`	Put F_0 and F_1 on the stack.
`   2 n`	From 2 to n ,
`   START`	Do the following loop:
`    DUP`	Make a copy of the latest F (initially F_1).
`    ROT`	Move the previous F (initially F_0) to level 1.
`    +`	Calculate the next F (initially F_2).
`   NEXT`	Repeat the loop.
`   SWAP DROP`	Drop F_{n-1}.
`  END`	End ELSE clause.
` »`	End the defining program.
`»`	End the program.
ENTER	Put the program on the stack.
`'FIB2` STO	Store the program as FIB2.

Example. Calculate F_6 and F_{10}. Note that FIB2 is faster than FIB1.

Calculate F_6.

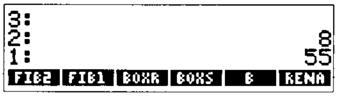
USER
6 FIB2



```
3:
2:
1:                              8
```

Calculate F_{10}.

10 FIB2



```
3:
2:                              8
1:                             55
```

Comparison of FIB1 and FIB2

FIB1 calculates intermediate values F_i more than once, while FIB2 calculates each intermediate F_i only once. Consequently, FIB2 is faster.

The difference in speed increases with the size of n because the time required for FIB1 grows exponentially with n, while the time required for FIB2 grows only linearly with n.

The diagram below shows the beginning steps of FIB1 calculating F_{10}. Note the number of intermediate calculations: 1 in the first row, 2 in the second row, 4 in the third row, and 8 in the fourth row.

$$\begin{tikzcd}
& & & & & & & F_{10} \arrow[dllll] \arrow[drrrr] & & & & & & & \\
& & & F_9 \arrow[dll] \arrow[drr] & & & & & & & & F_8 \arrow[dll] \arrow[drr] & & & \\
& F_8 \arrow[dl] \arrow[dr] & & & & F_7 \arrow[dl] \arrow[dr] & & & & F_7 \arrow[dl] \arrow[dr] & & & & F_6 \arrow[dl] \arrow[dr] & \\
F_7 & & F_6 & & F_6 & & F_5 & & F_6 & & F_5 & & F_5 & & F_4
\end{tikzcd}$$

Single-Step Execution

It's easier to understand how a program works if you execute it step by step, seeing the effect on the stack of each step. Doing this can help you "debug" your own programs or help you understand programs written by others.

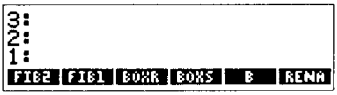
This section shows you how to execute FIB2 step by step, but you can apply these rules to any program. The general rules are:

1. Use VISIT to insert the command HALT in the program. Place HALT where you want to begin single-step execution. (You'll see how the position of HALT within FIB2 affects execution.)
2. Execute the program. When the HALT command is executed, the program stops (indicated by the "stopsign" annunciator).
3. Select the PROGRAM CONTROL menu.
4. Press SST once to see the next program step displayed and then executed.
5. You can now:
   - Keep pressing SST to display and execute sequential steps.
   - Press CONT to continue normal execution.
   - Press KILL to abandon further program execution.
6. When you want the program to run normally again, use VISIT to remove HALT from the program.

For the first example, insert HALT as the first command in FIB2.

Clear the stack and select the USER menu.

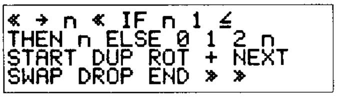
CLEAR
USER



```
3:
2:
1:
```

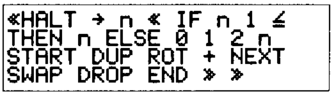
Use VISIT to return FIB2 to the command line.

' FIB2 VISIT



```
« → n « IF n 1 ≤
THEN n ELSE 0 1 2 n
START DUP ROT + NEXT
SWAP DROP END » »
```

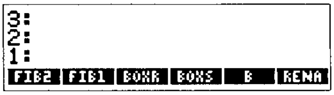
Use the cursor menu keys to insert HALT as shown.



```
«HALT → n « IF n 1 ≤
THEN n ELSE 0 1 2 n
START DUP ROT + NEXT
SWAP DROP END » »
```

Store the edited version of FIB2.

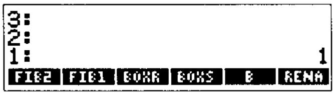
ENTER



```
3:
2:
1:
```

Calculate F_1. At first, nothing happens except that the "stopsign" annunciator appears.

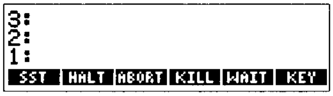
1 FIB2



```
3:
2:
1:                   1
```

Select the PROGRAM CONTROL menu and execute SST (*single-step*). (Watch the top line of the display to see the first step displayed before it's executed.)

CTRL
SST



```
3:
2:
1:
```

Note that → n constitutes one step; "step" is a logical unit rather than simply the next object in the program.

Look at the general rules at the beginning of this section. You've performed the first four steps, and now you can choose one of the three alternatives for step 5. For this example, press SST repeatedly until the "stopsign" annunciator disappears, indicating that FIB2 is completed. (These single-steps not shown here.)

The calculation for F_1 executes only the THEN clause in FIB2. For the second example, execute `3 FIB2` and single-step through the calculation for F_3. This executes the ELSE clause, including the START . . . NEXT loop. You'll see that, for $n = 3$, the START . . . NEXT loop is executed twice.

For the third example, suppose you want to single-step the START . . . NEXT loop as a whole – seeing the stack before each iteration of the loop, but not single-stepping all the steps in FIB2 or in the loop itself. To do so, move the HALT command inside the loop. Then FIB2 won't halt until it reaches the loop, and you can use [CONT] (*continue*) to execute the loop one iteration at a time.

Use VISIT to return FIB2 to the command line.

[USER]
' FIB2 [VISIT]

```
« HALT → n « IF n 1
≤ THEN n ELSE 0 1 2
n START DUP ROT +
NEXT SWAP DROP END »
```

Use the cursor menu keys to delete HALT. Then insert HALT as shown (on the third line, after START).

```
«  → n « IF n 1
≤ THEN n ELSE 0 1 2
n START HALT DUP ROT +
NEXT SWAP DROP END »
```

Store the edited version of FIB2.

[ENTER]

```
3:
2:
1:
FIB2 FIB1 BOXR BOXS  B  RENA
```

Start the calculation for F_3. FIB2 will halt before performing the loop.

3 FIB2

```
3:
2:                         0
1:                         1
FIB2 FIB1 BOXR BOXS  B  RENA
```

Continue execution of the loop. FIB2 will halt before performing the loop a second time.

[CONT]

```
3:
2:                         1
1:                         1
FIB2 FIB1 BOXR BOXS  B  RENA
```

Continue execution of the loop. Because this is the last iteration of the loop, FIB2 will execute to completion.

[CONT]

```
3:
2:
1:                         2
FIB2 FIB1 BOXR BOXS  B  RENA
```

When you're done experimenting with FIB2, don't forget to use VISIT to remove the HALT command.

Expand and Collect Completely

This section contains two programs:

- MULTI repeats a program until the program has no effect.
- EXCO uses MULTI to expand and collect completely.

MULTI (Multiple Execution)

Given an object and a program that acts on the object, apply the program to the object repeatedly until the object is unchanged.

Arguments	Results
2: *object* 1: « *program* »	2: 1: *resulting object*

Techniques:

- DO . . . UNTIL . . . END (indefinite loop). The DO clause contains the steps to be repeated; the UNTIL clause contains the test that determines whether to repeat both clauses again (if false) or to exit (if true).
- Programs as arguments. Although programs are commonly named and then executed by calling their names, programs can also be put on the stack and used as arguments to other programs.
- Evaluation of local variables. The program argument to be executed repeatedly is stored in a local variable. It's handy to store an object in a local variable when you don't know beforehand how many copies you'll need.

  MULTI demonstrates one of the differences between global and local variables: if a global variable contains a name or program, the contents of the variable are evaluated when the name is evaluated; but the contents of a local variable are always simply recalled. Consequently, MULTI uses the local name to put the program argument on the stack and then executes an explicit EVAL command to evaluate the program.

Program	Comments
«	Begin the program.
→ p	Create a local variable *p* that contains the program argument.
«	Begin the defining program.
DO	Begin the DO clause.
DUP	Make a copy of the object.
p EVAL	Apply the program to the object, returning a new version. (The EVAL command is necessary to execute the program because local variables always return their contents to the stack unevaluated.)
UNTIL	Begin the UNTIL clause.
DUP	Make a copy of the new version of the object.
ROT	Move the old version to level 1.
SAME	Test whether the old version and the new version are the same.
END	End the UNTIL clause.
»	End the defining program.
»	End the program.
[ENTER]	Put the program on the stack.
'MULTI [STO]	Store the program as MULTI.

Example. MULTI is demonstrated in the next program.

EXCO (Expand and Collect Completely)

Given an algebraic object, execute EXPAN repeatedly until the algebraic doesn't change, then execute COLCT repeatedly until the algebraic doesn't change. In some cases the result will be a number.

Arguments	Results
1: '*algebraic*'	1: '*algebraic*'
1: '*algebraic*'	1: *z*

Techniques:

- Structured programming. EXCO calls the program MULTI twice. Even if you don't use MULTI anywhere else, the efficiency of repeating all the commands in MULTI by simply including its name a second time justifies writing MULTI as a separate program.

Required Programs:

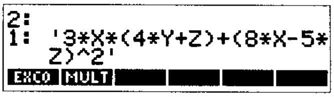
- MULTI (page 23) repeatedly executes the programs that EXCO provides as arguments.

Program	**Comments**
`«`	Begin the program.
`« EXPAN »`	Put EXPAN on the stack.
`MULTI`	Execute EXPAN until the algebraic object doesn't change.
`« COLCT »`	Put COLCT on the stack.
`MULTI`	Execute COLCT until the algebraic object doesn't change.
`»`	End the program.
ENTER	Put the program on the stack.
'EXCO STO	Store the program as EXCO.

Example. Expand and collect completely the expression $3x(4y+z)+(8x-5z)^2$.

Enter the expression.

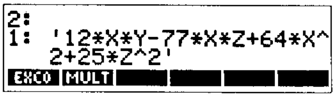
```
USER
'3*X
*(4*Y+Z)
+(8*X-5*Z)^2  ENTER
```



```
2:
1: '3*X*(4*Y+Z)+(8*X-5*
    Z)^2'
```

Expand and collect completely.

```
EXCO
```



```
2:
1: '12*X*Y-77*X*Z+64*X^
    2+25*Z^2'
```

Expressions with many products of sums or with powers can take many iterations of EXPAN to expand completely, resulting in a long execution time for EXCO.

Displaying a Binary Integer

This section contains three programs:

- PAD is a utility program that converts an object to a string for right-justified display.
- PRESERVE is a utility program for use in programs that change the calculator's status (angle mode, binary base, and so on).
- BDISP displays a binary integer in HEX, DEC, OCT, and BIN bases. It calls PAD to show the displayed numbers right-justified, and it calls PRESERVE to preserve the binary base.

PAD (Pad With Leading Spaces)

Convert an object to a string and, if the string contains fewer than 23 characters, add spaces to the beginning.

When a short string is displayed by using DISP, it appears *left-justified* – its first character appears at the left end of the display. The position of the last character is determined by the length of the string.

By adding spaces to the beginning of a short string, PAD moves the position of the last character to the right. When the string is 23 characters long, it appears *right-justified* – its last character appears at the right end of the display.

PAD has no effect on strings that are longer than 22 characters.

Arguments	Results
`1:` *object*	`1: "` *object*"

Techniques:

- WHILE ... REPEAT ... END (indefinite loop). The WHILE clause contains a test that determines whether to execute the REPEAT clause and test again (if true) or to skip the REPEAT clause and exit (if false).
- String operations. PAD demonstrates how to convert an object to string form, count the number of characters, and concatenate two strings.

Program	Comments
`«`	Begin the program.
`  →STR`	Make sure the object is in string form. (Strings are unaffected by this command.)
`  WHILE`	Begin WHILE clause.
`    DUP SIZE 23 <`	Does the string contains fewer than 23 characters?
`  REPEAT`	Begin REPEAT clause.
`    " " SWAP +`	Add a leading space.
`  END`	End REPEAT clause.
`»`	End the program.
ENTER	Put the program on the stack.
'PAD STO	Store the program as PAD.

Example. PAD is demonstrated in the program BDISP.

PRESERVE (Save and Restore Previous Status)

Given a program on the stack, store the current status, execute the program, and then restore the previous status.

Arguments	Results
1: « *program* »	1: *(result of program)*

Techniques:

- RCLF and STOF. PRESERVE uses RCLF (*recall flags*) to record the current status of the calculator in a binary integer and STOF (*store flags*) to restore the status from that binary integer.
- Local-variable structure. PRESERVE creates a local variable just to remove the object from the stack briefly; its defining program does little except evaluate the program argument on the stack.

Program	Comments
`«`	Begin the program.
`RCLF`	Recall a 64-bit binary integer representing the status of all 64 user flags.
`→ f`	Store the binary integer in a local variable *f* .
`«`	Begin the defining program.
`EVAL`	Execute the program argument.
`f STOF`	Restore the status of all 64 user flags.
`»`	End the defining program.
`»`	End the program.
ENTER	Put the program on the stack.
'PRESERVE STO	Store the program as PRESERVE.

Example. PRESERVE is demonstrated in the program BDISP.

BDISP (Binary Display)

Display a number in HEX, DEC, OCT, and BIN bases.

Arguments	Results
`1: #` *n*	`1: #` *n*
`1:` *n*	`1:` *n*

Techniques:

- IFERR . . . THEN . . . END (error trap). To accomodate real numbers, BDISP includes the command R→B (*real-to-binary*). However, this command causes an error if the argument is *already* a binary integer.

  To maintain execution if an error occurs, the R→B command is placed inside an IFERR clause. Because no action is required when an error occurs, the THEN clause contains no commands.

- Enabling LAST. In case an error occurs, LAST must be enabled to return the argument to the stack. BDISP sets flag 31 to programmatically enable the LAST recovery feature.
- FOR . . . NEXT loop (definite loop with index). BDISP executes a loop from 1 to 4, each time displaying *n* in a different base on a different line.

  The loop index (named *j* in this program) is a local variable. It's created by the FOR . . . NEXT program structure (rather than by a → command) and it's automatically incremented by NEXT.
- Subprograms. BDISP demonstrates three uses for subprograms.
  1. BDISP contains a main subprogram and a call to PRESERVE. The main subprogram goes on the stack and is evaluated by PRESERVE.
  2. When BDISP creates a local variable for *n*, the defining program is a subprogram.
  3. There are four subprograms that "customize" the action of the loop. Each subprogram contains a command to change the binary base and a marker (h, d, o, or b) to indicate the base. Each iteration of the loop executes one of these subprograms.

Required Programs:

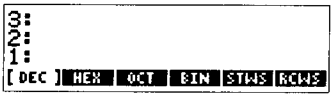
- PAD (page 26) expands a string to 23 characters so that DISP shows it right-justified.
- PRESERVE (page 27) stores the current status, executes the main subprogram, and restores the status.

```
Program                         Comments

«                               Begin the program.
 «                              Begin the main subprogram.
  DUP                           Make a copy of n .
  31 SF                         Set flag 31 to enable LAST.
  IFERR                         Begin error trap.
   R→B                          Convert n  to a binary integer.
  THEN                          If an error occured,
  END                           Do nothing (no commands in THEN
                                clause).
  → n                           Create a local variable n .
  «                             Begin the defining program.
   CLLCD                        Clear the display.
   « BIN "b" »                  Subprogram for BIN.
   « OCT "o" »                  Subprogram for OCT.
   « DEC "d" »                  Subprogram for DEC.
   « HEX "h" »                  Subprogram for HEX.
   1 4                          First and last index values.
   FOR j                        Start loop with index j .
    EVAL                        Evaluate one of the base subpro-
                                grams (initially the one for HEX).
    n →STR                      Make a string showing n  in the
                                current base.
    SWAP +                      Add the base marker.
    PAD                         Pad the string to 23 characters.
    j DISP                      Display the string in the j th line.
   NEXT                         Increment j  and repeat the loop.
  »                             End the defining program.
 »                              End the main subprogram.
 PRESERVE                       Store the current status, execute the
                                main subprogram, and restore the
                                status.
»                               End the program.

ENTER                           Put the program on the stack.
'BDISP STO                      Store the program as BDISP.
```

Example. Switch to DEC base, display # 100 in all bases, and check that BDISP restored the base to DEC.

Clear the stack and select the BINARY menu.

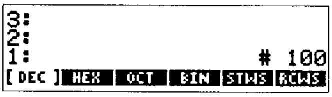
CLEAR
BINARY



```
3:
2:
1:
```

Make sure the current base is DEC and key in # 100.

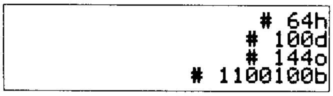
DEC
#100 ENTER



```
3:
2:
1:                 # 100
```

Execute BDISP. (Don't switch menus, since you'll want to see the BINARY menu in the next step.)

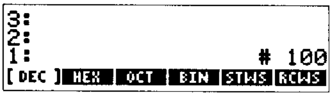
BDISP ENTER



```
             # 64h
            # 100d
            # 144o
        # 1100100b
```

Return to the normal stack display and check the current base.

ON

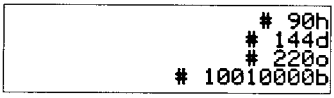


```
3:
2:
1:                 # 100
```

Although the main subprogram left the calculator in BIN base, PRESERVE restored DEC base.

To check that BDISP also works for real numbers, try 144.

USER
144 BDISP



```
             # 90h
            # 144d
            # 220o
       # 10010000b
```

Summary Statistics

For paired-sample statistics it's often useful to calculate the sum of the squares (Σx^2 and Σy^2) and the sum of the products (Σxy) of the two variables. This section contains five programs:

- SUMS creates a variable ΣCOV that contains the covariance matrix for the current statistics matrix ΣDAT.
- ΣGET extracts a number from the specified position in ΣCOV.
- ΣX2 uses ΣGET to extracts Σx^2 from ΣCOV.
- ΣY2 uses ΣGET to extracts Σy^2 from ΣCOV.
- ΣXY uses ΣGET to extracts Σxy from ΣCOV.

If ΣDAT contains n columns, ΣCOV is an $n \times n$ matrix. The programs ΣX2, ΣY2, and ΣXY refer to ΣPAR (*statistics parameters*) to determine which columns contain the x data (called C_1) and the y data (called C_2).

Techniques:

- Matrix operations. These programs demonstrate how to transpose a matrix, how to multiply two matrices, and how to extract one element from a matrix.
- Programs usable in algebraic objects. Because ΣX2, ΣY2, and ΣXY conform to algebraic syntax (no arguments from the stack, one result put on the stack), you can use their names like ordinary variables in an expression or equation.
- ΣPAR convention. Several paired-sample statistics commands use a variable named ΣPAR to specify a pair of columns in ΣDAT. ΣPAR contains a list with four numbers, the first two specifying columns. (The other two numbers are the slope and intercept from linear regression.)

  SUMS ensures that ΣPAR exists by executing 0 PREDV DROP; the command PREDV (*predicted value*) creates ΣPAR with default values if ΣPAR doesn't already exist, and DROP removes the predicted value computed for 0.

  ΣX2, ΣY2, and ΣXY use the values stored in ΣPAR to determine which element to extract from ΣCOV.

SUMS (Summary Statistics Matrix)

Create a variable ΣCOV that contains the covariance matrix of the statistics matrix ΣDAT.

As an example, if ΣDAT is the $n \times 2$ matrix

$$\begin{bmatrix} x_1 & y_1 \\ x_2 & y_2 \\ \cdot & \cdot \\ \cdot & \cdot \\ \cdot & \cdot \\ x_n & y_n \end{bmatrix},$$

then ΣCOV will contain the covariance matrix

$$\begin{bmatrix} \Sigma x^2 & \Sigma xy \\ \Sigma xy & \Sigma y^2 \end{bmatrix}.$$

Arguments	Results
1:	1:

Program	**Comments**
«	Begin the program.
RCLΣ	Recall the contents of the $n \times m$ statistics matrix ΣDAT.
DUP	Make a copy.
TRN	Transpose the matrix. The result is an $m \times n$ matrix.
SWAP *	Multiply the matrices to produce the $m \times m$ covariance matrix. (Without swapping the matrices, the product would be an $n \times n$ matrix.)
'ΣCOV' STO	Store the covariance matrix in a variable ΣCOV.
0 PREDV DROP	Make sure ΣPAR exists.
»	End the program.
ENTER	Put the program on the stack.
'SUMS STO	Store the program as SUMS.

ΣGET (Get an Element of ΣCOV)

Given p and q, each indicating either the first or second *position* in ΣPAR, extract the rs element from ΣCOV, where r and s are the corresponding first or second *elements* in ΣPAR.

ΣGET is called by ΣX2, ΣY2, and ΣXY with the following arguments.

- For ΣX2, $p = 1$ and $q = 1$.
- For ΣY2, $p = 2$ and $q = 2$.
- For ΣXY, $p = 1$ and $q = 2$.

Arguments	Results
2: 1 *or* 2 1: 1 *or* 2	2: 1: *rs element of ΣCOV*

Program	Comments
«	Begin the program.
ΣCOV	Put the covariance matrix on the stack.
ΣPAR	Put the list of statistics parameters on the stack.
DUP	Make a copy.
5 ROLL	Move p to level 1.
GET	Get r, the p th element in ΣPAR.
SWAP	Move ΣPAR to level 1.
4 ROLL	Move q to level 1.
GET	Get s, the q th element in ΣPAR.
2 →LIST	Put { r,s } on the stack.
GET	Get the rs element from ΣCOV.
»	End the program.
ENTER	Put the program on the stack.
'ΣGET STO	Store the program as ΣGET.

ΣX2 (Sum of Squares of *x*)

Calculate Σx^2, where the *x*'s are the elements of C_1 (the column specified by the first parameter in ΣPAR).

Arguments	Results
`1:`	`1:` Σx^2

Program	**Comments**
«	Begin the program.
`1 1`	Specify C_1 twice.
`ΣGET`	Extract Σx^2.
»	End the program.
`ENTER`	Put the program on the stack.
`'ΣX2` `STO`	Store the program as ΣX2.

ΣY2 (Sum of Squares of *y*)

Calculate Σy^2, where the *y*'s are the elements of C_2 (the column specified by the second parameter in ΣPAR).

Arguments	Results
`1:`	`1:` Σy^2

Program	**Comments**
«	Begin the program.
`2 2`	Specify C_2 twice.
`ΣGET`	Extract Σy^2.
»	End the program.
`ENTER`	Put the program on the stack.
`'ΣY2` `STO`	Store the program as ΣY2.

ΣXY (Sum of Products of x and y)

Calculate Σxy, where the x's and y's are corresponding elements of C_1 and C_2 (the columns specified by the first and second parameters in ΣPAR).

Arguments	Results
1:	1: Σxy

Program	Comments
«	Begin the program.
1 2	Specify C_1 and C_2.
ΣGET	Extract Σxy.
»	End the program.
`ENTER`	Put the program on the stack.
'ΣXY `STO`	Store the program as ΣXY.

Example. Calculate ΣX2, ΣY2, and ΣXY for the following statistics data:

$$\begin{bmatrix} 18 & 12 \\ 4 & 7 \\ 3 & 2 \\ 11 & 1 \\ 31 & 48 \\ 20 & 17 \end{bmatrix}$$

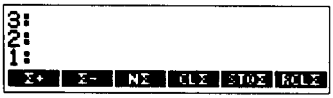
The general steps are as follows.

1. Enter the statistical data.
2. Execute SUMS to create the covariance matrix ΣCOV.
3. Execute ΣX2, ΣY2, and ΣXY.
4. If ΣDAT contains more than two columns (that is, if each data point contains more than two variables):
    a. Execute COLΣ to specify new values for C_1 and C_2. The values are stored in ΣPAR.
    b. Execute ΣX2, ΣY2, and ΣXY.

Now try the example given above.

Clear the stack, select the STAT menu, and clear ΣDAT.

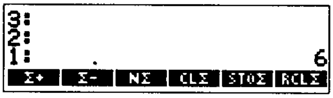
CLEAR
STAT
CLΣ



```
3:
2:
1:
```

Enter the data and then check that you entered all six data points.

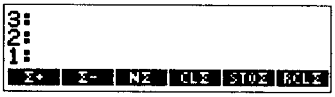
```
[18,12 Σ+
[4,7 Σ+
[3,2 Σ+
[11,1 Σ+
[31,48 Σ+
[20,17 Σ+
NΣ
```



```
3:
2:
1:                    6
```

Drop the number of data points.

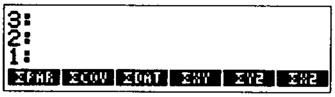
DROP



```
3:
2:
1:
```

Create the covariance matrix ΣCOV.

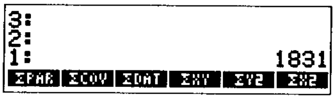
USER
SUMS



```
3:
2:
1:
```

Calculate Σx^2.

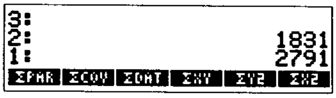
ΣX2



```
3:
2:
1:                 1831
```

Calculate Σy^2.

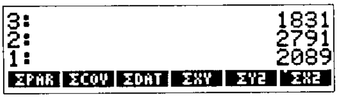
ΣY2



```
3:
2:                 1831
1:                 2791
```

Calculate Σxy.

ΣXY

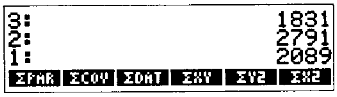


```
3:                1831
2:                2791
1:                2089
```

If the statistics matrix had more than two columns, you could specify new values for C_1 and C_2. For practice, specify $C_1=1$ and $C_2=2$ (the current values).

The command COLΣ is available in the STAT menu, but here it's easier to spell out the command name and stay in the USER menu.

1 ENTER
2 COLΣ ENTER



```
3:                1831
2:                2791
1:                2089
```

You could now execute ΣX2, ΣY2, and ΣXY for the new pair of columns C_1 and C_2.

Don't forget the execute SUMS again whenever you add or delete data from the statistics matrix ΣDAT.

Median of Statistics Data

This section contains three programs:

- SORT orders the elements of a list.
- LMED calculates the median of a sorted list.
- MEDIAN uses SORT and MED to calculate the median of the current statistics data.

SORT (Sort a List)

Sort a list into ascending order.

Arguments	Results
`1:` { *list* }	`1:` { *sorted list* }

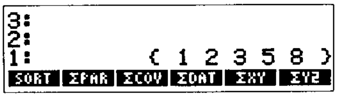
Techniques:

- Bubble sort. Starting with the first and second numbers in the list, SORT compares adjacent numbers and moves the larger number toward the end of the list. This process is done once to move the largest number to the last position in list, then again to move the next largest to the next-to-last position, and so on.
- Nested definite loops. The outer loop controls the stopping position each time the process is done; the inner loop runs from 1 to the stopping position each time the process is done.
- Nested local-variable structures. SORT contains two local-variable structures, the second inside the defining program of the first. This nesting is done for convenience; it's easier to create the first local variable as soon as its value is computed, thereby removing its value from the stack, rather than computing both values and creating both local variables at once.
- FOR . . . STEP and FOR . . . NEXT (definite loops). SORT uses two indexes: −1 STEP decrements the index for the outer loop each iteration; NEXT increments the index for the inner loop by 1 each iteration.

Program	**Comments**
`«`	Begin the program.
`DUP SIZE 1`	From the last position to the first position,
`FOR j`	Begin the outer loop with index *j* .
`1 j`	From the first position to the *j* th position,
`FOR k`	Begin the inner loop with index *k* .
`k GETI → n1`	Get the *k* th number in the list and store it in a local variable n_1.
`«`	Begin outer defining program.
`GETI → n2`	Get the next number in the list and store it in a local variable n_2.
`«`	Begin inner defining program.
`DROP`	Drop the index.
`IF n1 n2 >`	If the two numbers are in the wrong order,
`THEN`	Then do the following:
`k n2 PUTI`	Put the second one back in the *k* th position.
`n1 PUT`	Put the *k* th one back in the next position.
`END`	End of THEN clause.
`»`	End inner defining program.
`»`	End outer defining program.
`NEXT`	Increment *k* and repeat the inner loop.
`-1 STEP`	Decrement *j* and repeat the outer loop.
`»`	End the program.
ENTER	Put the program on the stack.
`'SORT` STO	Store the program as SORT.

Example. Sort the list { 8, 3, 1, 2, 5 }.

```
USER
{8,3,1,2,5 SORT
```



```
3:
2:
1:            { 1 2 3 5 8 }
```

LMED (Median of a List)

Given a sorted list, calculate the median. If the list contains an odd number of elements, the median is the value of the center element. If the list contains an even number of elements, the median is the average value of the elements just above and below the center.

Arguments	Results
1: { *sorted list* }	1: *median of sorted list*

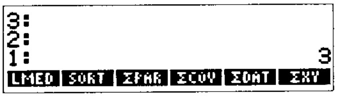
Techniques:

- FLOOR and CEIL. For an integer, FLOOR and CEIL both return that integer; for a non-integer, FLOOR and CEIL return successive integers that bracket the non-integer.

Program	Comments
`«`	Begin the program.
`DUP SIZE`	The size of the list.
`1 + 2 /`	The center position in the list (fractional for even-sized lists).
`→ p`	Store the center position in local variable *p*.
`«`	Begin the defining program.
`DUP`	Make a copy of the list.
`p FLOOR GET`	Get the number at or below the center position.
`SWAP`	Move the list to level 1.
`p CEIL GET`	Get the number at or above the center position.
`+ 2 /`	The average of the two numbers at or near the center position.
`»`	End the defining program.
`»`	End the program.
[ENTER]	Put the program on the stack.
' LMED [STO]	Store the program as LMED.

Example. Calculate the median of the list you sorted using SORT.

USER
LMED



```
3:
2:
1:                    3
LMED SORT ΣPAR ΣCOV ΣDAT ΣXY
```

LMED is called by MEDIAN.

MEDIAN (Median of Statistics Data)

Return a vector representing the medians of the columns of the statistics data.

Arguments	Results
1:	1: [$x_1 x_2 \cdots x_m$]

Techniques:

- Arrays, lists, and stack elements. MEDIAN extracts a column of data from ΣDAT in vector form. To convert the vector to a list, MEDIAN puts the vector elements on the stack and then combines them into a list. From this list the median is calculated using SORT and LMED.

  The median for the *m* th column is calculated first, and the median for the first column is calculated last, so as each median is calculated, it is moved to the stack level above the previously calculated medians.

  After all medians are calculated and positioned correctly on the stack, they're combined into a vector.

- FOR . . . NEXT (definite loop with index). MEDIAN uses a loop to calculate the median of each column. Because the medians are calculated in reverse order (last column first), the index is used to reverse the order of the medians.

Required Programs:

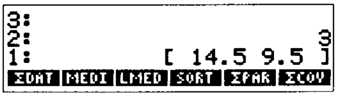
- SORT (page 39) arranges a list in ascending order.
- LMED (page 41) calculates the median of a sorted list.

Program	Comments
«	Begin the program.
RCLΣ	Put a copy of the current statistics matrix ΣDAT on the stack for safe-keeping.
DUP SIZE	Put the list { *n* *m* } on the stack, where *n* is the number of rows in ΣDAT and *m* is the number of columns.
LIST→ DROP	Put *n* and *m* on the stack. Drop the list size.
→ n m	Create local variables for *n* and *m* .
«	Begin the defining program.
'ΣDAT' TRN	Transpose ΣDAT. Now *n* is the number of columns in ΣDAT and *m* is the number of rows.
1 m	The first and last rows.
FOR j	For each row, do the following:
Σ-	Extract the last row in ΣDAT. Initially this is the *m* th row, which corresponds to the *m* th column in the original ΣDAT.
ARRY→ DROP	Put the row elements on the stack. Drop the index list { *n* }, since *n* is already stored in a local variable.
n →LIST	Make an *n* -element list.
SORT	Sort the list.
LMED	Calculate the median of the list.
j ROLLD	Move the median to the proper stack level.
NEXT	Increment *j* and repeat the loop.
m 1 →LIST	Make the list { *m* }.
→ARRY	Combine all the medians into an *m* -element vector.
»	End the defining program.
SWAP	Move the orginal ΣDAT to level 1.
STOΣ	Restore ΣDAT to its previous value.
»	End the program.
$\boxed{\text{ENTER}}$	Put the program on the stack.
'MEDIAN $\boxed{\text{STO}}$	Store the program as MEDIAN.

Example. Calculate the median of the data on page 36. (This example assumes you've keyed in the data.) There are two columns of data, so MEDIAN will return a two-element vector.

Calculate the median.

[USER]
≣ MEDIAN ≣



```
3:
2:                    3
1:       [ 14.5 9.5 ]
ΣDAT MEDI LMED SORT ΣPAR ΣCOV
```

The medians are 14.5 for the first column and 9.5 for the second column.

Index

Contents

HEWLETT PACKARD

00028-90099 English
Printed in U.S.A. 6/87