Programming HP Calculators/Printable version

Programming HP Calculators

Contents

Introduction

This book is an unofficial guide to programming your HP calculator. It will teach you the basic commands mentioned in the official manual but will also include some other commands, such as Easter eggs and glitches.

The fact that HP calculators are programmable means that they are not calculators but computers. Although the language used by the calculators is simple it is similar to many of the complex languages used on PCs, making it a great starting point for people who want to learn more about programming.

'Hello World' Example

```
1►L:
DISP L; "Hello,":
DISP 2; "World!":
FREEZE:
```

This program basically does the following:

- Store 1 to the variable, 'L'
- On line 'L' (1) display 'Hello,'
- On Line 2 display 'World!'
- Freeze the calculator's display so that the text displayed doesn't disappear.

Type this example into a new program so you can see what it does.

BTW: 'Hello, World!' is a standard program for most programming tutorials and guides. It has no real significance that I know of.

Storing and Retrieving Variables

On HP calculators you can store numbers into variables. This is done with the '►' or '1►L: DISP L;' symbol, like in the example above. In the example above we store 1 in the variable, 'L'. This means that 'L=1''.

On HP calculators there are only prescribed variables, see the variable reference to see them all. Because all variables are prescribed there is no need to declare them at the start of the program like in most languages. Most of the time the A...Z and θ are the only variables used in programs.

Program Commands

Program Commands are instructions for the calculator. A program can contain any number of commands. When your calculator reads a command it will take the appropriate action; however, sometimes we might want that action to be slightly different each time we call it. This is why we have arguments after the command separated by a semicolon. Some commands have an unlimited number of arguments but most have somewhere between 0 and 8 arguments.

So that means that commands are formatted as:

```
COMMAND <arg1>; <arg2>; ... <arg#>
```

The arguments can be of various types. Sometimes they are numbers and sometimes they are strings (a collection of letters and symbols with quotation marks on each side). The 'DISP' command, for example, uses a number and a string for its arguments. It has the format:

```
DISP <line#>; <string>
```

The first argument is a number and the second argument is a string. This command basically tells the calculator to write 'string' on line 'line#'. If you want a more detailed description check the command reference.

Also, at the end of every command or variable assignment we need to put a colon so that the calculator knows that the command has ended.

Using Menus

Because it can be slow to type the commands and variables manually and they are easy to misspell you can select them from menus. These menus can be accessed by pressing the **MATH** button for commands and the **VAR** button for variables.

Commands

There are several subsets of the calculator commands. They are presented here along with a short summary of their features.

This section is currently incomplete. The status of each sub-section is written in italics below its link.

Aplet Commands

Current status: 0 of 4 commands completed. **(0%)**

The aplet commands are the commands used for building aplets (*not* applets) on the calculator. Most of these are fairly advanced, and you should have mastered other programming skills before you use the aplet commands, as they work on existing programs.

Commands included in this section			
CHECK	SELECT	SETVIEWS	UNCHECK

Branch Commands

Current status: 4 of 5 commands completed. **(80%)**

The branch commands are used to set the program to do different tasks depending on a certain situation. More information on branching (or "jumping") in programming can be found at Wikipedia: Jump instruction.

Commands included in this section			
IF	CASE	IFERR	RUN

Drawing Commands

Current status: 7 of 7 commands completed. **(100%)**

The drawing commands are used for drawing images on the screen. They are differentiated from graphic commands by the fact that they interface with the display, not graphic objects (GROBs). That is to say, using a command such as ARC will draw on the screen, not in a GROB.

Commands included in this section			
ARC	BOX	ERASE	FREEZE
LINE	PIXON/PIXOFF	TLINE	

Graphic Commands

Current status: 6 of 12 commands completed. **(50%)**

The graphic commands are used for creating and modifying GROBs - GRaphic OBjects. These are essentially variables on the calculator from G0 to G9 that can contain graphic information, ie. a 1-bit bitmap.

Commands included in this section			
DISPLAY→	→DISPLAY	→GROB	GROBNOT
GROBOR	GROBXOR	MAKEGROB	PLOT→
→PLOT	REPLACE	SUB	ZEROGROB

Loop Commands

Current status: 0 of 4 commands completed. **(0%)**

The loop commands are used to set the calculator to keep performing an action until a certain condition is satisfied. For example, to display all the numbers from 1 to 100, you could use a loop command, rather than individually writing code to display each number.

Commands included in this section			
DO	WHILE	FOR	BREAK

Matrix Commands

Current status: 1 of 10 commands completed. **(10%)**

The matrix commands are used to edit and modify matrices, ie. the variables M0 to M9.

Commands included in this section			
ADDCOL/ROW	DELCOL/ROW	EDITMAT	RANDMAT
REDIM	REPLACE	SCALE	SCALEADD
SUB		SWAPCOL/ROW	

Print Commands

Current status: 0 of 3 commands completed. **(0%)**

The print commands are used to send information to HP's separately-sold infrared printer. They are generally not very useful.

Commands included in this section		
PRDISPLAY	PRHISTORY	PRVAR

Prompt Commands

Current status: 7 of 10 commands completed. **(70%)**

Prompt commands are used to ask the user for information, or alert the user to some fact. These are amongst the most important commands available for your use.

Commands included in this section			
BEEP	CHOOSE	DISP	DISPTIME
EDITMAT	FREEZE	GETKEY	INPUT
MSGBOX		WAIT	

Stat Commands

Current status: 0 of 7 commands completed. **(0%)**

The stat commands are actually two groups of commands, Stat-One and Stat-Two. These two groups represent one-variable and two-variable statistics modes, respectively. They are used for editing statistics and performing statistical functions.

Commands included in this section			
DO1VSTATS	RANDSEED	SETFREQ	SETSAMPLE
D02VSTATS	SETDEPEND	SETINDEP	

Easter Egg Commands

Current status: 8 of 8 commands completed. **(100%)**

An "easter egg" is a common term used to refer to "hidden" or undocumented features in a program or operating system, in this case, the calculator. The commands documented here are not in the CMDS menu, and they are not documented by HP. Most of them don't serve any useful purpose.

Commands included in this section			
AMIGOS	DEMO	HELPWITH	LIBEVAL
RULES	SYSEVAL	VERSION	WSLOG

Commands/Aplet Commands

CHECK

SELECT

SETVIEWS

UNCHECK

Commands/Branch Commands

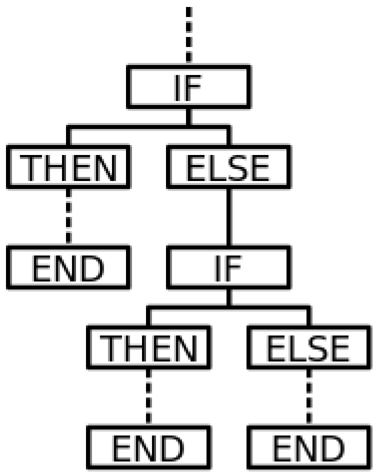
IF

Syntax

```
IF <test> THEN <clauses>:
ELSE <clauses>:END:
```

Or for "nesting" IF statements:

```
IF <test> THEN <clauses>:
ELSE IF <test> THEN <clauses>:
ELSE <clauses>:END:
ELSE <clauses>:END:
```



Using IF...THEN.

Or for running IF such that no action is performed should the test be false:

```
IF <test> THEN <clauses>:
ELSE END:
```

Detail

- IF statements are a set of commands used to test certain equations or inequations. If the result is "true", then a set of actions are performed. If not, no action is performed, or a different set of actions are performed as per ELSE.
- Tests can be combined with AND or OR.

Example

```
PROMPT A:
IF A==0 THEN
MSGBOX "You entered zero.":
ELSE MSGBOX "You didn't enter zero.":END:
```

This is a basic example, only checking for one result. Here's a more complex one, although multiple IF...THEN statements are usually best replaced with CASE.

```
PROMPT A:
IF A==0 THEN
MSGBOX "You entered zero.":
ELSE IF A>0 THEN
MSGBOX "You entered a positive number.":
ELSE END:ELSE IF A<0 THEN
MSGBOX "You entered a negative number."
ELSE END:ELSE END:
```

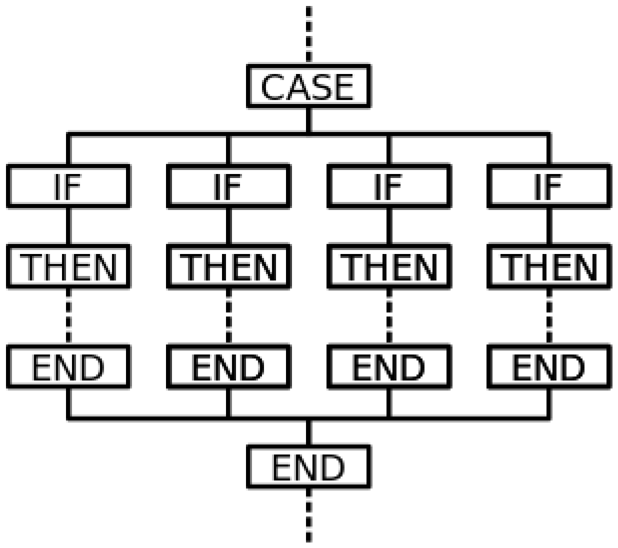
Note that the additional "ELSE IF" commands displayed are not actually commands per se. What is actually happening is that the result of the previous test, when evaluated as "false", moves on to the ELSE result, where it encounters IF, telling it to start *another* IF statement. This is what the "ELSE

END" commands do - they close off the secondary statements nested within the original statement.

CASE

Syntax

```
CASE IF <test> THEN <clauses>: END ... END:
```



Using CASE...END.

Detail

CASE is the command that is used in the place of multiple statements.

The best way of describing this is with a flowchart, seen right. Compare to the lower IF flowchart, and you can see how the CASE command is more efficient when there is more than one option.

Example

CASE is especially useful when accepting input from a CHOOSE list. For example:

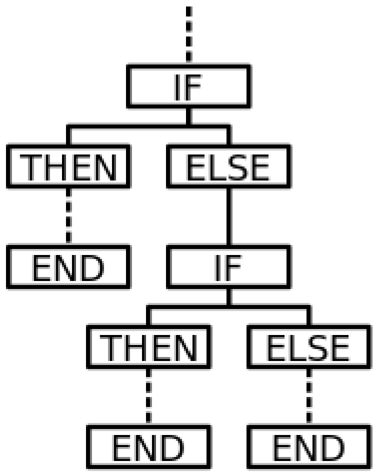
```
1?A:CHOOSE A;
"FAVOURITE APPLE?";
"Granny Smith";
"Golden Delicious";
"Pink Lady":
CASE

IF A==1 THEN
MSGBOX "Granny Smiths are my favourite, too!":
END

IF A==2 THEN
MSGBOX "Golden Delicious apples are OK, too.":
END

IF A==3 THEN
MSGBOX "Bleh, I can't stand Pink Ladies!":
END

END:
MSGBOX "Program finished.":
```



Using IF...THEN.

The first five lines initiate the CHOOSE menu. Of particular note is the 1?A at the start, this sets the default value of the menu to 1 (ie. the first option) and thus ensures that the menu operates correctly (if the value of A begins as outside the acceptable value for the menu, the highlighting will not operate correctly).

CASE initiates the CASE statement. You can then see three branches of this statement; one for each of the three menu options specified in CHOOSE. The basic syntax is:

```
CASE IF <clause 1> THEN <action set 1> END IF <clause 2> THEN <action set 2> END ... END:
```

Importantly (and counter-intuitively), the END at the end of an IF/THEN/END statement within CASE does **not** contain a full-colon at the end of it! However, the final END completing the CASE does require one.

You should be familiar with the MSGBOXs in each action set, these are a simple way of displaying text in a dialog and FREEZEing at the same time. The final MSGBOX is included to demonstrate what happens once the command completes; it skips to the final END:, so if you had somehow tricked the program into giving a non-valid answer (ie. changing the input for A) the program would just display the final MSGBOX. The same happens after the appropriate clause has been run for A==?.

IFERR

RUN

Syntax

```
RUN "<program-name>":
```

Detail

- Ceases running the current program, and runs <program-name> instead.
- You can self-reference using this command, creating an infinite loop.

Dangers

- The program name you reference is case-sensitive; that is, "ProGramNaME" is not the same as "programname".

Example

This program is called "INFINITE".

```
MSGBOX "You can't escape!":
RUN "INFINITE":
```

It should be noted that you can "escape", just press ON - this usually cancels the current command from running and this feature is designed specifically to allow users to cancel programs stuck in an infinite loop due to a programming error or malicious intent by the software author.

STOP

Syntax

```
STOP:
```

Detail

- Exits the program immediately. Compare it to BREAK.

Example

This program is called "EXAMPLE".

```
MSGBOX "Want to exit? 1 = yes, 0 = no.":
PROMPT A:
IF A==1 THEN STOP:
ELSE RUN "EXAMPLE":
```

This command is practically self-explanatory.

Commands/Drawing Commands

Important note

The drawing commands are somewhat confusing in that they reference parts of the 'plot', not parts of the physical screen.

It is intuitive to assume that (knowing that the screen is 131x64 pixels wide, and that the calculator references points by a Cartesian-like system) a pixel at (1,1) will **always** be the most bottom-right pixel on the screen.

However, this is not the case. (1,1) is the pixel at (1,1) on the infinite 'plot'. This means that it will **only** be the bottom-right pixel, as expected, when the plot is set to the dimensions 1?X?131 and 1?Y?64. This can be done by setting the (X/Y)(MIN/MAX) commands to appropriate values:

```
1?XMIN:131?XMAX:
1?YMIN:64?YMAX:
```

Adding this code to the start of your program will set the calculator's plot size to these specifications automatically, ensuring smooth operation of your program. However, upon the program's completion, any graph the user might have set up will be displayed on a (presumably) incorrect plotsize, which may frustrate the user. You could work around this like follows:

```
XMIN?A:XMAX?B:
YMIN?C:YMAX?D:
1?XMIN:131?XMAX:
1?YMIN:64?YMAX:

<program code goes here>

A?XMIN:B?XMAX:
C?YMIN:D?YMAX
```

Of course, this adds bloat to the program somewhat, but will resolve the problem **as long as** you don't use the variables you selected in the program for other purposes. An alternate method would be to store the values in a list.

ARC

Syntax

```
ARC <cent-x>;<cent-y>;<radius>;<start>;<end>:
```

Detail

- <cent-x> and <cent-y> are the x- and y-coordinates of the center of the arc.
- <radius> is simply the radius of the arc.
- <start> is the starting angle of the arc, and <end> the finishing angle.

- Angles are measured mathematically, that is, the same as the unit circle; with 0° being the equivalent of due east on the compass, not due north, and counter-clockwise being the positive direction.

Dangers

- The ARC command utilises the current angle mode for arguments <start> and <end>. Therefore, if you design your arcs using degrees, it may be wise to set the angle mode to degrees programmatically like so:

```
1?HAngle:
```

- Alternately, an HAngle value of 2 or 3 can be used to set the calculator in radians or grads, respectively.
- The ARC command is particularly slow, so if you need to draw extensively, it may be wise (although time-consuming) to use MAKEGROB.

Example

```
ARC 65;32;15;0;360:
```

Draws a full circle of radius 15px, from the center of the screen.

```
ARC 65;32;15;10;80:
ARC 65;32;15;100;170:
ARC 65;32;15;190;260:
ARC 65;32;15;280;350:
```

Draws a full circle comprised of four arcs, but the circle is missing a arc of 20° at each cardinal point. Represented more concisely using a loop:

```
FOR X=0 TO 270 STEP 90;
ARC 65;32;15;10+X;80+X:
END:
```

BOX

Syntax

```
BOX <x1>;<y1>;<x2>;<y2>:
```

Detail

- (<x1>,<y1>) are the coordinates of the box's starting corner.
- (<x2>,<y2>) are the coordinates of the corner that completes the box.
- There are no limitations on the corners used, as long as they are binary opposites, that is top-left and bottom-right or bottom-left and top-right. Additionally, the order they are used in is not significant.

Example

```
BOX 1;1;131;64:
```

Draws a box around the border of the screen.

```
BOX 1;1;131;64:
BOX 2;2;130;63:
```

Draws a border 2 pixels thick around the screen. More concisely:

```
FOR X=0 TO 1;
BOX 1+X;1+X;131-X;64-X:
END:
```

This allows you to easily adjust the thickness of your border by simply adjusting the FOR command; for "FOR X= TO Z" the border will always be (Z-1) pixels thick.

ERASE

Syntax

```
ERASE:
```

Detail

- Erases the contents of the screen, as simple as that.
- Adding ERASE: before any of the drawing commands is usually a good idea to prevent the program from drawing over the programs catalog, or any other view that the user happens to be in at the time.

FREEZE

Syntax

```
FREEZE:
```

Detail

- Stops the program, and waits for a keypress to continue.
- Use it after you finish your drawing commands, so that the user has time to look at the screen!

LINE

Syntax

```
LINE <x-start>;<y-start>;<x-end>;<y-end>:
```

Detail

- Draws a line from point (<x-start>,<y-start>) to point (<x-end>,<y-end>).

Example

```
LINE 1;1;131;64:
LINE 1;64;131;1:
```

Draws a cross across the screen.

PIXON/PIXOFF

Syntax

```
PIXON <x>;<y>:
PIXOFF <x>;<y>:
```

Detail

- Turns the pixel at (<x>,<y>) on or off.

Dangers

- PIXON and PIXOFF are reasonably fast, but when inside a FOR loop (see below), can be appreciably slow. If you want to do something like a 'pretend' loading bar (or even a real one if it's applicable), you should use LINE or TLINE as they are faster.

Example

```
FOR X=1 TO 131 STEP 1;
FOR Y=64 TO 1 STEP -1;
PIXON X;Y:
END:END:
FOR X=131 TO 1 STEP -1;
FOR Y=1 TO 64 STEP 1;
PIXOFF X;Y:
END:END:
```

Fills the screen with pixels from left-to-right, top-to-bottom, then reverses the process.

TLINE

Syntax

```
TLINE <x-start>;<y-start>;<x-end>;<y-end>:
```

Detail

- Same as LINE, except the command performs a slightly different task - instead of turning a line of pixels on, it toggles them. That is to say, if all the pixels you're drawing across are off already (say you just used ERASE:), it will function as a regular LINE, but if there are already pixels turned on, it will turn those off, and conversely turn off pixels on.

Example

```
ERASE:
DISPLAY? G1:
GROBNOT G1:
?DISPLAY G1:
TLINE 1;1;131;64:
TLINE 1;64;131;1:
```

Draws a cross in white across the screen, after turning on all the pixels. Therefore, it acts the same as:

```
ERASE:
LINE 1;1;131;64:
LINE 1;64;131;1:
DISPLAY? G1:
GROBNOT G1:
?DISPLAY G1:
```

Commands/Graphic Commands

DISPLAY→

Syntax

```
DISPLAY→ <name>:
```

Detail

- Stores the display in GROB <name>. (Note that the arrow is going **out of** DISPLAY.)
- Usually used to generate GROBs on which to perform additional graphic commands.

Example

```
DISPLAY→ G1:
ERASE:WAIT 2:
→DISPLAY G1:
FREEZE:
```

Stores the display in GROB G1, clears the screen, waits two seconds, then reloads the display on the screen and pauses. (Effectively useless without other graphic commands).

→DISPLAY

Syntax

```
→DISPLAY <name>:
```

Detail

- Sends GROB <name> to the display. (Note that the arrow is going **in to** DISPLAY.)
- Usually used to output GROBs to the display after altering them using other graphic commands.

Example
See example for DISPLAY→.

→GROB

GROBNOT

Syntax

```
GROBNOT <name>:
```

Detail

- Inverts the colors in GROB <name>, that is, switches all ON pixels OFF and vice-versa.
- Could be used to select buttons, for example, but is generally too slow for this purpose.

Example

```
ERASE:
DISPLAY→ G0:
GROBNOT G0:
→DISPLAY G0:
FREEZE:
```

This program will erase the screen (ie. turn all pixels OFF), then invert the colours (ie. turn all pixels ON), giving you a black screen.

GROBOR

GROBXOR

MAKEGROB

Syntax

```
MAKEGROB <name>; <width> <height>
<hex-data>:
```

Detail

- Allows you to make a custom grob, pixel-by-pixel using hexadecimal.
- Much faster than regular drawing commands, so if you have the time to write your GROBs manually, it's strongly recommended. However, it does usually increase the size of the program far more than just using other drawing commands, although this does not affect compile time - so you need to decide which is more important, program size or program speed.
- Each hexadecimal digit (a number between 0-9, or a letter between A-F) represents a four-digit binary number, which in turn represents four pixels on the screen (where 1 is on and 0 is off). The following table illustrates the pixel on/off sequences produced by each hex digit:

0: 0000 or □□□□	**1:** 1000 or ■□□□	**2:** 0100 or □■□□	**3:** 1100 or ■■□□
4: 0010 or □□■□	**5:** 1010 or ■□■□	**6:** 0110 or □■■□	**7:** 1110 or ■■■□
8: 0001 or □□□■	**9:** 1001 or ■□□■	**A:** 0101 or □■□■	**B:** 1101 or ■■□■
C: 0011 or □□■■	**D:** 1011 or ■□■■	**E:** 0111 or □■■■	**F:** 1111 or ■■■■

Table detailing pixel patterns created by using different hex digits.

- The syntax is quite counter-intuitive; the grob name (G0 for example) comes first, followed by a semicolon and a space, then the width and height measurements for the grob, separated by spaces (rather than the usual semicolons). After that, a new line **must** be created, and the hex-data begins, which is finally ended with the regular :. No newlines must be created whilst typing the hex-data, although the calculator will naturally wrap the data onto the next line anyway.

- **Useful tip:** To generate the MAKEGROB code for the entire screen, use other methods to draw the image you desire (eg. SKETCH mode), then press ON+PLOT to save a screenshot to G0. Now, go into HOME view and enter G0, then hit ENTER. Finally, select the text that has just appeared starting with MAKEGROB, and press F5 (COPY), then hit ENTER again. The code is now stored in the *Editline* program, and you can insert it into any other program by pressing VARS, scrolling down to Programs, selecting Editline, pressing F4 (VALUE), and pressing F6 (OK).
  - Another idea is to crop this grob using SUB to create a sub-GROB, then typing the variable name (eg. G1) for the cropped GROB in HOME view. Use the above procedure to copy this MAKEGROB code into your program, and you will be able to save substantial amounts of space in the program that would otherwise have been taken up by 0s or other erroneous data. If only the calculator had some form of RLE..

Example

```
MAKEGROB G1; 12 12
FFF10810899999910850A999168108FFF:
```

```
■■■■■■■■■■■■
■□□□□□□□□□□■
■□□□□□□□□□□■
■□□■■□□■■□□■
■□□■■□□■■□□■
■□□□□□□□□□□■
■□■□□□□□□■□■
■□□■■□□■■□□■
■□□□□■■□□□□■
■□□□□□□□□□□■
■■■■■■■■■■■■
```

The above MAKEGROB code translated into the creation of the GROB roughly approximated by the ASCII sketch above.

PLOT→

→PLOT

REPLACE

SUB

ZEROGROB

Commands/Loop Commands

DO

WHILE

FOR

BREAK

Commands/Matrix Commands

ADDCOL/ADDROW

DELCOL/DELROW

EDITMAT

Syntax

```
EDITMAT <matrix>:
```

Detail

- Opens the edit matrix screen, allowing the user to input data into a matrix. <matrix> can be one of the ten matrix variables from M0 to M9.
- Essentially the same as PROMPT - but used for matrix input.

Example

```
MSGBOX "This program multiplies two matrices.":
MSGBOX "Please enter the first matrix.":
EDITMAT M1:
MSGBOX "Please enter the second matrix.":
EDITMAT M2:
M1*M2?M3:
MSGBOX "The result of the multiplication is stored in M3."
```

This program will multiply two matrices, as detailed in the MSGBOX commands.

RANDMAT

Syntax

```
RANDMAT <matrix>;<#rows>;<#columns>:
```

Detail

- Generates a matrix in variable <matrix> of size <#rows> x <#columns>, setting each element to a random integer (apparently) from -9 to 9, inclusive.
- Essentially the same as PROMPT - but used for matrix input.

Example

```
RANDMAT M1;50;50:
```

Makes a really big matrix filled with random integers that wastes a fair bit of your calculator's memory, as well as drains the batter power writing all that info to memory. Useful!

REDIM

REPLACE

SCALE

Note: This information was obtained via trial and error and may be incorrect, pending verification. Splintax 12:57, 8 February 2006 (UTC)

Syntax

```
SCALE <matrix>;<factor>;<row>:
```

Detail

- Multiplies all elements in the row <row> of matrix <matrix> by scalar <factor>.

Example

```
[[1,2],[3,4]] |STO| M1:
SCALE M1;5;1:
EDITMAT M1:
```

Generates the 2x2 matrix with [1 2] on the top and [3 4] on the bottom row, then multiplies the top row (row 1) by 5 to make it [5 10].

SCALEADD

Note: This information was obtained via trial and error and may be incorrect, pending verification. Splintax 13:03, 8 February 2006 (UTC)

Syntax

```
SCALEADD <matrix>;<factor>;<rowa>;<rowb>:
```

Detail

- Adds each element of <rowa>, multiplied by <factor>, to each element of <rowb>.

Example

```
[[1,2],[3,4]] |STO| M1:
SCALEADD M1;5;1;2:
EDITMAT M1:
```

Generates the 2x2 matrix with [1 2] on the top and [3 4] on the bottom row, then adds the top row multiplied by 5 (that is, [5 10]) to the bottom row to produce [8 14].

SUB

SWAPCOL/SWAPROW

Commands/Print Commands

PRDISPLAY

PRHISTORY

PRVAR

Commands/Prompt Commands

BEEP

Syntax

```
 BEEP <freq>;<time>:
```

Detail

- Uses the calculator's piezoelectric beeper to beep at a frequency of <freq> Hz for <time> seconds.

Example

```
INT(RANDOM*9+1)?R:
MSGBOX "Guess the number between 1 and 10."
WHILE A?R REPEAT
PROMPT A:
IF A==R THEN
BEEP 400;.2:
BEEP 600;.5:
MSGBOX "You win!":
STOP:
ELSE BEEP 600;.2:
BEEP 400;.5:
IF A>R THEN MSGBOX "Lower..":
ELSE MSGBOX "Higher..":
ELSE END:
ELSE END:END:
```

This program generates a random number from 1 to 10, and prompts the user for a guess. If the user guesses correctly, the calculator beeps twice with rising pitch (typical "correct" sound). Otherwise, it beeps twice with lowering pitch (typical "wrong" sound), and tells the user if the answer is higher or lower than what they specified.

CHOOSE

Syntax

```
CHOOSE <variable>;
"<title-name>";
"<option-1>";
"<option-2>";
...
"<last-option>":
```

Detail

- Allows the user to select from a set of options given on the screen.
- After two ;s, CHOOSE starts allocating spaces on the on-screen menu and assigning values to those spaces. Therefore, in the syntax example, <option-1> would have a value of 1 - if it were selected, <variable> would be set to 1.

- This option is best used in conjunction with CASE to define the effect of each option.

Dangers

- CHOOSE starts with the current value of <variable> as the item selected. **Therefore, it will be impossible to use the menu should the variable have a value that is not a positive integer, or is greater than the number of options.**
- A simple workaround for the following is always inserting 1?<variable> at the start of the CHOOSE command, to ensure that the first option is selected by default.

DISP

Syntax

```
DISP <line>;"<text>":
```

Detail

- Displays <text> on line number <line> (1 to 7).
- Although there is room for 8 lines after an ERASE, setting <line> as less than 1, non-integer, or more than 7 will result in the text simply not printing (no error message).
- DISPXY is able to handle printing on the "8th line", should you require more room.

Dangers

- DISP erases the contents of the entire line that it is about to print on, regardless of whether or not pixels have already been drawn in that area. Therefore, you should always use DISP first, should you intend to draw over it. DISPXY corrects this problem to a degree, although use of it still results in *some* whitespace being forced over your drawing.
- This can be used to your advantage should you wish to blank out a certain line of text, simply use DISP <line>;" ":. This can be advantageous as it will obfuscate your code (if you want to limit the alterations people make to your code), and because it is much faster than any other method (although it allows the least control).

Example

```
ERASE:DISP 1;"Hello World!":
FREEZE:
```

Displays the ubiquitous "Hello World" on the first line on the screen, after erasing the screen, and freezes the program at the end so that the user has time to read the message.

```
1?A:1?B:ERASE:
FOR X=1 TO 500;
A+B?C:B?A:C?B:
B/A?P:
DISP 1;B":"A:
DISP 2;P:
```

Advanced: Approximates phi using the ratio between increasing Fibonacci numbers. The ability of DISP to erase the line before displaying its payload creates an interesting-looking "reiterative" effect.

DISPTIME

Syntax

```
DISPTIME:
```

Detail

- Displays the date and time in a message box.
- The time is stored in the TIME and DATE real variables.
  - **TIME** syntax: hh.mmss (with hh in 24-hour time). For example, 2:30PM is 14.3000. Since the variable is an integer, the 0s at the end actually go on for 13 characters; this is because the TIME variable actually stores fractions of seconds, too. (Thus you could set 1PM as just 13, or 12:05:30.136 as 12.0530136.)
  - **DATE** syntax: mm.ddyyyy. For example, 21st September, 2005 is 9.212005.

Example

This program is called "TimeUtils".

```
1?M:CHOOSE M;
"Display time";
"Set time":
CASE
IF M==1 THEN
DISPTIME:STOP:END
IF M==2 THEN
MSGBOX "Hour? (24h time)":
PROMPT H:
MSGBOX "Minute?":
PROMPT M:
H+M/100?TIME:
MSGBOX "Set. Date?":
PROMPT D:
MSGBOX "Month?":
PROMPT M:
MSGBOX "Year?":
PROMPT Y:
D+M/100+Y/1000?DATE:
MSGBOX "Set.":RUN "TimeUtils":END
END:
```

This program is a multi-function application that allows you to both display and edit (correct) the current time. Most of it should be self-explanatory, given the information about the DATE and TIME variables in Details.

EDITMAT

See EDITMAT under the matrix commands section for information on this command.

FREEZE

Syntax

```
FREEZE:
```

Detail

- Pauses the running of the program, waiting for input from a key.
- Best used after using drawing commands, or else the drawings will disappear from the screen straight away.
- For the readers who know more advanced programming languages, it is the same as GETKEY, but it does not return a value to the calculator (ie. cannot detect which key was pressed).

GETKEY

INPUT

MSGBOX

Syntax

```
MSGBOX "<message>":
```

Detail

- The simplest way of displaying a message on screen, requiring only the text to be displayed; however, this is its weakness - it offers almost no control.
- Automatically incorporates a FREEZE, that is, you need to press a key to get out of the MSGBOX.

Example

```
PROMPT A:
MSGBOX "You entered "A".":
```

WAIT

Commands/Stat Commands

Stat-One Commands

DO1VSTATS

RANDSEED

SETFREQ

SETSAMPLE

Stat-Two Commands

D02VSTATS

SETDEPEND

SETINDEP

Commands/Easter Egg Commands

AMIGOS

Syntax

```
AMIGOS:
```

Details

- Displays a crossword of people associated with the development of the calculator.

DEMO

Syntax

```
DEMO:
```

Details

- Gives a demo of all the features of your calculator.

HELPWITH

Syntax

```
HELPWITH <cmd>:
```

Details

- Displays help for the command <cmd>. It is useful when teaching oneself programming, although not useful *in* programs themselves.

LIBEVAL

Syntax

```
LIBEVAL <library number>; <object number>:
```

Details

- This command allows access to libraries on the calculator. Not many libraries or objects are known, but a short list is provided here:
  - **2;0:** Shows you the sources from all the commands of the menu math.

- **2;149:** This command allows the finding of new functions of math.

RULES

Syntax

```
RULES:
```

Details

- Displays a crossword of the calculator's developers.

SYSEVAL

Syntax

```
SYSEVAL <address>:
```

Details

- Can access many functions of the calculator that are otherwise hidden. A short list of some of the working values for <address> is provided below:
  - **171591** A piece of left-over code for 39g users.
  - **151587** Puts the keyboard in ALPHA mode.
  - **151607** Puts the keyboard again in his normal state.
  - **19027** Turns the calculator off.
  - **258091** Let you receive Aplets, Notes, Programmes, lists, matrices from in the homescreen.
  - **258092** Let you send Aplets, Notes, Programmes, lists, matrices from in the homescreen. This function doesn't work well.
  - **258960** Gives the version of METAKERNEL in a grob.
  - **259723** Makes some new menus that surely comes from METAKERNEL.
  - **260385** Tetris for the 39g users.
  - **260387** Dynablaster for the 39/40g users. Press DEL to get out of it.
  - **260243** Built-in Filer for 39/40g users.
  - **259588 + X** Clear list X. To clarify, clearing list 0 (L0) would be SYSEVAL 259588, list 1 SYSEVAL 259589, list 2 SYSEVAL 259590 and so on.

Dangers

- If you use an incorrect address, there is a high chance that your calculator's memory will be cleared. **Be careful while using this command!**

VERSION

Syntax

```
VERSION:
```

Details

- Gives the ROM version of your calculator.

WSLOG

Syntax

```
WSLOG:
```

Details

- Shows the calculator log.

Variables

Real variables are the letters A through Z, plus θ (Greek lowercase theta). They act just like regular numbers stored in the calculator.

Plot-View Variables

Symbolic-View Variables

Numeric-View Variables

Home Variables

Note Variables

Sketch Variables