Programming in User RPL

Eduardo M Kalinowski, ekalin@iname.com

Version 2.0

Contents

1 Introduction **2**

2 First concepts **3**
2.1 Stack manipulation 4
2.2 Exercises 6

3 Local variables **8**
3.1 Exercises 9

4 Conditional tests **10**
4.1 Trues, Falses, Zeros and Ones 10
4.2 Flags 11
4.3 Basic conditionals: `IF...THEN...ELSE` 12
4.4 `CASE` structures 13
4.5 Exercises 15

5 Loop structures **18**
5.1 Indefinite Loop Structures 18
5.2 Definite loop structures 19
5.3 Exercises 20

6 Error handling **23**

7 Getting Input **25**
7.1 Enter input the by the Command Line: `INPUT` 25
7.2 Presenting options: `CHOOSE` 26
7.3 Input forms: `INFORM` 27
7.4 Getting key presses 29
7.5 Exercises 30

8 Displaying output **33**
8.1 Displaying message boxes: `MSGBOX` 33
8.2 Producing sounds 33
8.3 Displaying text strings 33
8.4 Using `FREEZE` 33
8.5 Clearing the display 33

A Answers to exercises **35**
Chapter 2 — First concepts 35
Chapter 3 — Local variables 36
Chapter 4 — Conditional tests 37
Chapter 5 — Loop structures 39
Chapter 7 — Getting input 43

1 Introduction

If you've payed 270 bucks for a calculator, it should do something really nice. Well, it does. It's a programmable calculator, which mean you can "teach" it to do something it does not automatically do. But in the HP48, with its complete and powerful programming language and structure, allows more than simple programs that repeat keystrokes, like in other calculators. You can create programs with graphics, user interfaces, and much more.

In this tutorial, you'll learn how to make simple (and some a bit more complicated) programs in UserRPL, the programming language of the HP48. Most examples are related to math, and most programs are totally useless: the calculator does the same the program does automatically. They just examples, to demonstrate the concepts you've learned.

A note about the exercises: some chapters have exercises. They are meant to be done, not just skipped. If you don't practice, how will you learn? There are answers to all of them, but the solution I give is only one possibility in many cases. There are many others. If your program worked, then you should consider the exercise as solved, even if the answer I give is different or even more efficient. But compare my solution to yours. Sometimes they may be completely different answers. Which one is more efficient? Which one is faster? Which one is smaller? If you have a different solution to a problem, send it to me at my e-mail: `ekalin@iname.com`.

If you have found any error in this tutorial, be it a spelling error, an error in a program, or anything else, please warn me at the above e-mail. I'd also like to hear comments and suggestions.

2 First concepts

For 270 bucks, you should expect a calculator that does everything you need. Several functions are already available, but probably you'll soon need something the HP cannot do automatically. Say you have to calculate the determinant of a symbolic matrix. Since the HP has built-in support only for numerical matrices, you'll have to calculate it manually. If you need to do that only once, that's OK, but what to do if you need several calculations, always doing the same thing, pressing the same keys, only for different inputs?

That's why the HP48 is different from other calculators. It is *programmable*. It means it can learn how to do something. In the example above, you could write a program that calculates the determinant of a symbolic matrix when you give its elements. So, to calculate a determinant, the only thing you would need to do is enter the elements and run the program. Want to calculate the determinant for different inputs? No problem, just key them in and run the program again, and you'll get the new results.

Basically, a programmable calculator can save you time.

On the HP48, programs are delimited by left and right guillemots (that's the « and » symbols.) To put enter those delimiters, press left-shift and minus. Inside, there is a sequence of objects, separated by spaces. Depending on the object type, some action is taken, according to table 1.

Object	**Action**
Command	Run
Number	Put into stack
Algebraic	Put into stack
String	Put into stack
List	Put into stack
Program	Put into stack
Global name (between '')	Put into stack
Global name (without '')	Put into stack and automatically evaluated
Local name (between '')	Put into stack
Local name (without '')	Put into stack and automatically evaluated

Table 1: Objects in programs and their actions

Let's see a simple example of a program. Suppose you have a 52-gallon cylindrical water heater and you wish to determine how much energy is being lost because of poor insulation. For that you can use the formula

$$q = hAT$$

where q is the heat loss from the heater (in btu per hour), h is the heat transfer coefficient (in our case 0.47), A is the total surface area of the cylinder (in our example 30) and T is the temperature difference.

The values of h and A will remain constant in our case, but we want to calculate the heat loss for various temperature differences. You could key in the values every time and multiply them, but this task can be simplified with a very simple program.

Let's create a program that calculates the heat loss for the heater when the temperature difference is given in the stack. Type the following program, and press ENTER (one more time: to get « » press Left-shift Minus):

```
« 30 * .47 * »
```

Let's examine this program carefully. The first thing after the « opening program delimiter is the number 30. All numbers are put into the stack, so the stack now contains the temperature difference (which you entered) and 30, the surface area (put by the program.) The next object is a command, *. It will be executed, multiplying the two numbers in the stack. This will put in the stack the result of $A \times T$. In sequence there is another number, 0.47, that will be put into the stack. The last object is another command, again *. In the stack there is now the result of the multiplication of $A \times T$ by h, or hAT, which is the wanted result q.

Now let's try the program and see if it works. If you have not already entered the program, do it now. You should see the program in level one of the stack. To save it in the memory, type `'HT'`, press ENTER, then STO. Now press VAR. You should see `HT` on the first menu key. Congratulations, you've just saved your first User RPL program.

To try it, enter 15 and press the first softkey, corresponding to the just stored `HT` program. You should get 211.5 in the stack. Now, if you wanted to calculate again for a temperature difference of 18 degrees, just enter 18 and run the program again. You should get 253.8.

To see how practical it would be if you had to calculate several times, calculate the heat loss for temperature differences of 10, 12, 14, 16, 18 and 20. The answers are 141, 169.2, 197.4, 225.6, 253.8 and 282.

2.1 Stack manipulation

Among the several commands that can be included in programs, some of the most important are the ones that manipulate the stack, ie, change the order of the elements, remove some elements or make copies of some of them. The HP48 provides 13 commands for manipulating the stack. It is not recommended that you know what they do. It is **essential** to know what they do. Was I clear? You **must** know how to use them and their effect. Here they are:

`DUP` Makes a copy of the object in level one. Example (in all subsequent examples, the left side represents the stack before the command, and the right side shows the stack after the command):

```
2:     | 2:  a
1:  a  | 1:  a
```

`DUP2` Makes copies of the objects in levels one and two. Example:

```
4:     | 4:  a
3:     | 3:  b
2:  a  | 2:  a
1:  b  | 1:  b
```

`DUPN` Makes copies of n objects, starting at level two. n is given in level one. Example:

```
6:     | 6:  a
5:     | 5:  b
4:  a  | 4:  c
3:  b  | 3:  a
2:  c  | 2:  b
1:  3  | 2:  c
```

`DROP` Drops (ie, removes) the object in level one. Example:

```
2:  a  | 2:
1:  b  | 1:  a
```

`DROP2` Drops the objects in levels one and two. Example:

```
3:  a  | 3:
2:  b  | 2:
1:  c  | 1:  a
```

`DROPN` Drops n objects from the stack, starting at level two. n is given in level one. Example:

```
5:  a  | 5:
4:  b  | 4:
3:  c  | 3:
2:  d  | 2:
1:  3  | 1:  a
```

`OVER` Returns a copy of the object in level two. Example:

```
3:     | 3:  a
2:  a  | 2:  b
1:  b  | 1:  a
```

`PICK` Returns a copy of the object in level n+1. n is given in level one. Example:

```
4:  a  | 4:  a
3:  b  | 3:  b
2:  c  | 2:  c
1:  3  | 1:  a
```

`SWAP` Exchanges levels one and two. Example:

```
2:  a  | 2:  b
1:  b  | 1:  a
```

`ROLL` Moves the object in level n+1 to level 1. As usual, n is in level one. Example:

```
5:  a | 5:
4:  b | 4:  b
3:  c | 3:  c
2:  d | 2:  d
1:  4 | 1:  a
```

`ROLLD` Moves the object in level two to level n+1. Example:

```
5:  a | 5:
4:  b | 4:  d
3:  c | 3:  a
2:  d | 2:  b
1:  4 | 1:  c
```

`ROT` Moves the object in level three to level one. This command is equivalent to `3 ROLL`. Example:

```
3:  a | 3:  b
2:  b | 2:  c
1:  c | 1:  a
```

`DEPTH` Returns the number of elements in the stack. Example:

```
4:    | 4:  a
3:  a | 3:  b
2:  b | 2:  c
1:  c | 1:  3
```

Now let's see another very simple program. It calculates the hypotenuse of a right triangle, given the two legs in the stack.

`« SQ SWAP + √ »`

Let's examine this program closely. First, it squares the number in level one. Then it swaps levels one and two, and squares the object now at level one, the other side of the triangle, which was before in level two. The program then adds them, and takes the square root. Simple, isn't it?

2.2 Exercises

1. What happens when the following programs are run?

   (a) `« « « 1 2 + » » EVAL »`

   (b) `« « « 1 2 + » EVAL » »`

   (c) `« « « 1 2 + » EVAL » EVAL »`

   (d) `« « « 1 2 + » » EVAL EVAL »`

   (e) `« « 1 « 2 + » » EVAL »`

(f) `« « 1 « 2 + » EVAL » EVAL »`

2. Design a program that, when the radius of a sphere is in level one, calculates its volume. ($V = \frac{4}{3}\pi r^3$)

3. Write a program to convert from Fahrenheit degrees to Celsius degrees. ($^\circ C = \frac{9^\circ F}{5} + 32$)

4. Write a program that does the inverse conversion (that is, from Celsius to Fahrenheit).

5. Write a program to calculate the parallel resistance R_P of two resistors R_1 and R_2, whose values are in the stack, using the formula $\frac{1}{R_P} = \frac{1}{R_1} + \frac{1}{R_2}$.

6. Write a program that does the same thing as the above, but now using the formula $R_P = \frac{R_1 R_2}{R_1 + R_2}$, derived by isolating R_P in the above formula.

7. Write a program to calculate the determinant $\begin{vmatrix} a & b \\ c & d \end{vmatrix}$, assuming a, b, c and d are in the stack, in this order.

8. Write a program to calculate the distance of two points (x_1, y_1) and (x_2, y_2), assuming x_1, x_2, y_1 and y_2 are in the stack, in this order.

9. Write a program to find one root of a quadratic equation, given a, b and c in the stack.

10. Modify the above program so that it finds both roots of the equation.

11. Write a program to find the surface area of a rectangular box, using the equation $S = 2(hw + hd + wd)$.

3 Local variables

Let's examine again the program given at the end of the last chapter, the one that calculates the hypotenuse of a right triangle:

```
« SQ SWAP + √ »
```

In the above program, each of the two inputs is used only once, so stack manipulation is not complicated. But in some cases the inputs are used many times, so keeping the stack organized can be quite difficult, and fetching a specific value is something complicated. So, instead of using lots to commands to get the element you want, why not store it somewhere, and them simply recall item when need arises? It would be a lot easier.

To do that, you could use variables. Just enter a name, and use the `STO` command, and it will be stored. It can later be recalled with `RCL`. But this brings some problems: what if the user already has a variable with the name you want to use? And you must take care to erase all variables used when the program ends. But what happens if the program terminates unexpectedly, because of an error, for example? The user's VAR menu would be full of unnecessary stuff.

To help solve this problems, the HP48 calculator has a nice feature called *local variables*. They work like normal variables, but they don't appear in the VAR menu, they are purged automatically when the program ends, they can't be accessed by the user and there is no problem in using a name that's already a user's variable.

Basically, one could say that local variables can exist only when a program is running and are accessible only to the program running, and any other programs called by that program.

Local variables are created with the → command (the arrow is above the 0 key.) The use is very simple: anywhere you want, use → followed by as many names of variables you want, separated by spaces. The value will be taken from the stack and stored in the variables.

Let's clarify the above: suppose you have 1, 2 and 3 in the stack, in this order. If you use → `a`, the local variable `a` will now contain the value 3. If you use → `a b`, then `a` will contain 2 and `b` will contain 3. Finally, if you use → `a b c`, `a` will contain 1, `b`, 2 and `c`, 3.

OK, but how do I use this nice feature in a program? Very simple: add the → command followed by the variable names and then either an algebraic expression that will be evaluated substituting any local variables by their values or a program that will be run, where you can use local variables as any other variable.

Let's see an example: the above hypotenuse program could be written this way:

```
« → a b '√ (a^2+b^2)' »
```

using algebraic notation or

```
« → a b « a SQ b SQ + √ » »
```

using a sub-program.

In this case, the first version (without local variables) is more efficient. But in more complicated cases, local variables generally much better. At least, they make the programs easier to understand, because you can see what's being done, and not a lot of stack manipulation commands such as `SWAP 4 PICK OVER ROT ROT DUP`. However, you must know how the stack works, and the basic stack manipulation commands before you go further in this tutorial. Alas, by this time, you should already know them. If you don't, go back to the previous chapter and learn what they do.

3.1 Exercises

1. Given a, b, u, c, d and v in the stack, write a program to solve the system

$$\begin{cases} ax + by = u \\ cx + dy = v \end{cases}$$

   To make things easier:

$$x = \frac{du - bv}{ad - bc} \quad \text{and} \quad y = \frac{av - cu}{ad - bc}$$

2. Write a program, using local variables, to calculate the determinant of the matrix

$$\begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix}$$

   using:

   (a) Algebraic notation (ie, $\rightarrow$ `a '4*a'`)

   (b) RPN notation (ie, $\rightarrow$ `a « a 4 * »`)

3. Write a program to calculate the inverse matrix of the matrix in the previous exercise.

4 Conditional tests

One of the most powerful abilities of the HP is that programs can make decisions, and execute actions based on these decisions. Of course it can't, for example, choose a good wine for you, but it could recommend the type of wine based on the type of meat. This kind of decision, based on some given factor (type type of meat) is called *conditional*, and is what we'll see in this chapter. But before we start studying HP48 commands, let's study a bit of Boolean algebra. It's not going to hurt so much.

4.1 Trues, Falses, Zeros and Ones

We will now study some operations of *boolean* data type. A boolean has only two possible values: *true* and *false*. In the HP48, there is not a special boolean object type, and real numbers are used to represent them. A zero is a *false*, and any other value (generally one) is *true*.

There are four basic operations with boolean values: NOT, AND, OR and XOR. (Actually, XOR is not a basic operation, but a derived operation, as we'll see later.)

Let's start with the easiest: NOT. This is the only unary operation (takes only one argument.) In the HP, it is represented by `'NOT A'`, in textbooks by $\neg A$ or $\overline{A}$ (we'll use the latter notation in this tutorial.) This function inverts its operand: a true becomes a false, and vice-versa.

See table 2, which describes the result of NOT for its possible inputs. This kind of table is called a *truth table*.

A	$\overline{\mathbf{A}}$
0	1
1	0

Table 2: NOT truth table

The next operation we'll see is AND. It is represented in the HP48 by `'A AND B'` and in textbooks by $A \wedge B$, $A.B$ or simply AB. The AND function is true only if both inputs are true. Its truth table is shown in table 3.

A	**B**	**A.B**
0	0	0
0	1	0
1	0	0
1	1	1

Table 3: AND truth table

The third function is OR. As you might have guessed, the HP48 represents it as `'A OR B'`. The textbook representations are $A \vee B$ or $A + B$. This function is true if at least of the inputs is true. See table 4 for the truth table.

A	**B**	**A + B**
0	0	0
0	1	1
1	0	1
1	1	1

Table 4: OR truth table

Last, but not less important, there is XOR, which stands for eXclusive OR. It returns true if the inputs are different. You'll see it represented on the HP48 as `'A XOR B'` and in textbooks as $A \underline{\vee} B$ or $A \oplus B$. Its truth table is in table 5.

A	**B**	**A $\oplus$ B**
0	0	0
0	1	1
1	0	1
1	1	0

Table 5: XOR truth table

XOR is derived from the other functions. There are several ways to represent it using only the first three operations. One of them is

$$A \oplus B = \bar{A}B + A\bar{B}$$

That was all you needed to know about Boolean algebra. But before we really start programming, let's see a little more theory.

4.2 Flags

A flag is like a reminder. If you had to remember to, say, pick up dinner on the way home, you would do something as a reminder to yourself, like tying a string to your finger. OK, you would never do that, but this is just an example.

Sometimes, a program also needs to remember something. But a program can't tie a string to its finger. Alas, a program doesn't have fingers. But since anatomy is not one of my favorite subjects, let's stop here. Back to programming, what could a program do to remember something?

The answer is: storing some information in some place that can be read back later. A normal (or local) variable could be used, but there are better things: *flags*. A flag is like a variable, but can only contain two values: set and clear. Sounds like the boolean thing above, doesn't it?

The HP48 has 128 flags. Of these, 64 are reserved for its own use (not all are used, however.) You can change their value, but this would mess the way the calculator operates. For example, system flag 40 controls the the clock display. If your program altered it, it would also change the displaying of the clock.

Because of this, there is another group of flags: the *user flags*. These 64 flags aren't used by the HP48. You can use them to do whatever you want.

The 64 system flags are represented by negative numbers, and the user flags by positive numbers. So, -40 means system flag 40, and 55, user flag 55.

OK, but what can I do with flags? Simple: instead of tying a string to your finger to remember to pick up dinner, set flag 4 (or any other.) Then, later, see whether the flag is set. If it is, then pick up dinner; otherwise don't. A program does exactly that, but to remember other things (after all, calculators don't eat.)

There are several commands to work with flags. They set flags, clear flags, and return information about whether a flag is set or clear. They are summarized in table 6.

Command	**Action**
`SF`	Sets the flag specified in level 1
`CF`	Clears the flag specified in level 1
`FS?`	Returns 1 if the flag is set; 0 otherwise
`FC?`	Returns 1 if the flag is cleared; 0 otherwise
`FS?C`	Returns 1 if the flag is set; 0 otherwise, then clears flag
`FC?C`	Returns 1 if the flag is cleared; 0 otherwise, then clears flag
`RCLF`	Puts a list with the values of all flags in the stack
`STOF`	Sets the flags according to the list in the stack

Table 6: Flag-related commands

4.3 Basic conditionals: `IF...THEN...ELSE`

The above commands let you check whether a flag is set or not. Now we might need to do some action if it is set, and some other if it is not. Or we might want to do something in case a generic condition is met, and some different thing if not. That's what conditional structures are for. The simples one is `IF`. It's form is:

```
« IF test condition THEN
     actions to be taken if condition is true
  END
»
```

First, the *test condition* will be evaluated. If the result is true (ie, non-zero), then the commands between `THEN` and `END` will be executed. Otherwise, they will be skipped and execution will continue after `END`.

This structure has another clause, that allows execution of certain commands only if the ***test condition*** is false (ie, zero). The new form is:

```
« IF test condition THEN
     actions to be taken if condition is true
  ELSE
     actions to be taken if condition is not true
  END
»
```

OK, in theory that's easy to understand. But in practice, how can one use that? Normally, we use comparison commands. They are: == (yes, to check equality you must use == and not just =), <, >, ≠, ≤ and ≥. All take two arguments from the stack, and return 1 if the condition is true and 0 if not. Ready for `THEN`. You can use `AND` and the like to check for multiple conditions.

Let's see a simple example of a program using `IF` structures. The program takes three numbers from the stack, and returns the smallest of them.

```
« → a b c @ Store the numbers for easy access
  « IF a b < a c < AND THEN
       a @ If a is smaller than b and c, it's the wanted number
    ELSE
       IF b c < THEN
          b @ If b is smaller than c, it's the number
       ELSE
          c @ otherwise it's c
       END
    END
  »
»
```

(In User RPL, `@` represents a comment. Everything after it until the end of the line is ignored.

4.4 CASE structures

Now let's write a program that gets a number as input and outputs it as a string. Let's restrict it to numbers up to four, so the program does not get excessively long.

```
« → x
  « IF 1 == THEN
       "One"
    ELSE
       IF x 2 == THEN
          "Two"
       ELSE
```

```
          IF x 3 == THEN
             "Three"
          ELSE
             IF x 4 == THEN
                "Four"
             ELSE
                "Other"
             END
          END
       END
    END
  »
»
```

The method above works, but is quite inefficient. Notice that the `IF`'s look like a stairway, descending each time more. There is a structure that makes this kind of structure much more efficient: the `CASE` structure. The above program, rewritten to use `CASE` would be:

```
« → x
  « CASE
        x 1 == THEN
            "One"
        END
        x 2 == THEN
            "Two"
        END
        x 3 == THEN
            "Three"
        END
        x 3 == THEN
            "Four"
        END
        "Other"
    END
  »
»
```

The general form of the structure is:

```
CASE
     test clause 1 THEN
          set of actions 1
     END
     test clause 2 THEN
          set of actions 2
     END
```

```
    ⋮
    test clause n THEN
        set of actions n
    END
    default set of actions
END
```

You can have any number of test clauses. When one evaluates to true (non-zero), the corresponding set of actions is executed, until `END`. The rest of the `CASE` structure is skipped until another `END`. If one clause evaluates to false (zero), the set of actions is skipped until the first `END`, and the next clause is evaluated. If none of the clauses evaluate true, the default action are taken. Note it isn't necessary to include a default set of actions.

4.5 Exercises

1. What's the value of each of the following expressions?

   (a) $0 \wedge 1 \vee 0$

   (b) $1 \wedge 1 \vee 0 \wedge \neg 0$

   (c) $(1 \oplus 1) \times \overline{1 + 1}$

   (d) $(1 \oplus (1 \times (0 \times \overline{0 \oplus 1}))) \oplus (\overline{1} + (\overline{0} \times \overline{1}))$

2. Assuming $a = 0$, $b = 1$, $c = 1$ and $d = 0$, what is the value of the expressions?

   (a) $a \wedge (\neg b)$

   (b) $a + \overline{b \oplus d}$

   (c) $c + (d + (a + (c \times (a \oplus d))))$

   (d) $(b \oplus \overline{c}) \times (\overline{c} + \overline{(a \times b) \oplus c})$

3. Supposing the variables `a`, `b`, `c`, `d` and `e` contain, respectively, 127, 10, 5, 0 and 1, which do each of the following programs return?

   (a) `« d NOT »`

   (b) `« d e AND »`

   (c) `« a b > b c < OR »`

   (d) `« a b < NOT »`

   (e) `« d e AND a b == OR »`

   (f) `« d e XOR a b < AND »`

   (g) `« a b + c < d AND e XOR d NOT AND »`

   (h) `« a b + c b / * 3 == a b OR NOT AND »`

4. Which value will be returned when this program is run?

```
« 0 1 0 2.5 3.5 → a b c x y
  « IF c x y + r > a NOT b AND OR OR THEN
       0
     ELSE
       1
     END
  »
»
```

5. Given the program below:

```
« → a b c
  « IF a THEN
       Command1
     ELSE
       IF b THEN
          IF c THEN
             Command2
          ELSE
             Command 3
             Command 4
          END
       END
     END
     Command5
  »
»
```

   (a) Which commands will be executed if the stack contains 1, 1 and 0?

   (b) Which commands will be executed if the stack contains 0, 1 and 0?

   (c) What should the stack contain so that only `Command5` is executed?

   (d) Which commands will be executed if the stack contains 0, 1 and 1?

6. After running this program, what will be the output?

```
« 32 2 5 → a c h
  « a 5 xroot c 3 4 / * → b j
    « IF b j > THEN
         8 h 6 SQ c / / *
      ELSE
         a h a / + h -
      END
    »
  »
»
```

7. Write a program that toggles the value of a flag (ie, if it is set it is cleared, and vice-versa) specified in level one.

8. Write a program that reads a flag number and a true/false from the stack. It it's true, it sets the flag, if not is clears the flag.

9. Write a program that reads three numbers form the stack and outputs them in increasing order.

10. Write a program that, given the month, day and year, in this order, calculates the day of the week, and outputs its name ("Sunday", "Monday", etc.) Use the formula

$$d_w = \left(\lfloor 2.6m - 2 \rfloor + \left\lfloor \frac{a}{4} \right\rfloor + \left\lfloor \frac{s}{4} \right\rfloor + d + a - 2s \right) \mod 7$$

where

- m is the month. January and February as months 11 and 12 of the previous year. March is month 1, and December is month 10.
- d is the day of the month.
- a are the two last digits of year.
- s are the first two digits of year.
- d_w is the day of the week. Sunday is 0, Monday is 1 and so on.

5 Loop structures

In the previous chapter, you learned some of User RPL control structures: the conditionals. They allow part of a program to be executed only if a certain condition is met. Now, we'll see other types of control structures: the *loop structures*. They allow part of a program to be executed more than once.

5.1 Indefinite Loop Structures

The first kind of loop structure you'll see is the *indefinite loop*. As the name says, it repeats an indefinite number of times, while (or until) a condition is met. There are two forms of indefinite loops in the HP48: `WHILE...REPEAT...END` loops and `DO...UNTIL...END` loops. Basically, they do the same thing: repeat the code while a certain condition is met, or until a condition is met.

The form of the `WHILE` structure is:

```
WHILE
      test condition
REPEAT
      code to be repeated
END
```

When the HP sees `WHILE` in a program, it evaluates ***test condition***. If it is true, then the ***code to be repeated*** is executed, and when `END` is reached, the program restarts execution at `WHILE`. If the ***test condition*** evaluates false, then the program resumes after the `END`. Since the condition is evaluated before the main code, the loop may never be executed, if the condition evaluates false the first time.

Let's see a simple example of `WHILE`. The program below calculates the sum of the even numbers between 100 and 200 (inclusive):

```
« 0 @ Initial sum
  100 @ First number
  → sum n @ Save in variables to make things simpler
  « WHILE
          n 200 ≤
    REPEAT @ Repeat while the number hasn't reached 200
          n 'sum' STO+ @ Increment sum
          'n' 2 STO+ @ Yes, STO+ accepts its arguments in any order
    END
    sum @ Output the sum
  »
»
```

The other form of indefinite loop is `DO...UNTIL...END`. It works like `WHILE`, but instead of repeating *while* a condition is true, it repeats *until* a condition is true (which can also be said as while a condition is false). Its form is:

```
DO
   code to be executed
UNTIL
   test condition
END
```

The *code to be executed* will be executed at least once, until *test condition* evaluates true.

A note valid for both forms of loops. If you use something like

```
WHILE 1 REPEAT ...  END
```

the program will be executed until your batteries drain completely or until it is stopped by the user. So be careful to always put a condition to stop the loop.

Now let's see how the above program would look like rewritten with DO. The only change is that the test condition has been reverted.

```
« 0 100 → sum n
  « DO
       'sum' n STO+
       2 'n' STO+
    UNTIL
       n 200 >
    END
    sum
  »
»
```

5.2 Definite loop structures

Different from the above structures, which will run a unknown number of times, the *definite loop structures* run a predefined number of times. There are four different variations of loop structures in the HP48, but the differences are minimal.

The simplest possible is START...NEXT. It's form is:

```
start stop START
      commands to be executed
NEXT
```

START reads to numbers from the stack, the initial number and the final number. Then, the *commands to be executed* are executed. When NEXT is reached, the initial number is incremented by one. If it is smaller than or equal to the final number, then the commands will be executed again. If it is greater, then execution will continue after NEXT.

The second type of definite loop structure is the FOR...NEXT loop. It is exactly like START, with one difference: you can access the number of that execution and use it as a normal variable. The syntax is:

```
start stop FOR var
      commands to be executed
NEXT
```

The variable can have any name, and it works like any local variable. You can even assign a new value to it. Traditionally, the variable is called `i`, `j` or `k`, but this is just a convention.

In both kind of loops, you can change `NEXT` by *increment* `STEP`. The difference is that the count will be incremented by *increment* instead of 1.

Now let's see our program in using a `FOR` loop:

```
« 0 @ Initial sum
  100 200 @ Goes from 100 to 200
  FOR i @ The current number is i
      i + @ Increment sum by i
  2 STEP @ Increment count by 2, since we want only even numbers
»
```

5.3 Exercises

1. Running the following program, which results do you get?

   ```
   « { } 10
     DO
        DUP 1 - 3 ROLLD SQ + SWAP
     UNTIL
        DUP 1 ==
     END
     DROP
   »
   ```

2. Modify the above program to use a `WHILE` structure (change as little as possible).

3. Write a program to calculate the factorial of a number given in the stack.

4. Suppose Country A has a population of 30,000,000 inhabitants and an annual growth rate of 3%; and Country B has a population of 200,000,000 inhabitants and an annual growth rate of 1.5%. Write a program (using `DO`) to calculate the number of years necessary for the population of Country A to get equal of greater than the population of Country B. Output the number of years and the population of each country.

5. A radioactive chemical element loses half of its mass every 50 seconds. Given the initial mass (in grams) in the stack, write a program to determine the time necessary for the mass to get lass than 0.5 grams. Output the final mass and the time in hours, minutes and seconds in the form "*x*h *y*min *z*s".

6. Write programs to calculate the following sums:

   (a) $S = \dfrac{1}{1} + \dfrac{3}{2} + \dfrac{5}{3} + \dfrac{7}{4} + \cdots + \dfrac{99}{50}$

   (b) $S = \dfrac{2^1}{50} + \dfrac{2^2}{40} + \dfrac{2^3}{48} + \cdots + \dfrac{2^{50}}{1}$

   (c) $S = \dfrac{37 \times 38}{1} + \dfrac{36 \times 37}{2} + \dfrac{35 \times 36}{3} + \cdots + \dfrac{1 \times 2}{37}$

   (d) $S = \dfrac{1}{1} - \dfrac{2}{4} + \dfrac{3}{9} - \dfrac{4}{16} + \cdots + \dfrac{10}{100}$

   (e) $S = \dfrac{1000}{1} - \dfrac{997}{2} + \dfrac{994}{3} - \dfrac{991}{4} + \cdots - \dfrac{853}{50}$

   (f) $S = \dfrac{480}{10} - \dfrac{475}{11} + \dfrac{470}{12} - \dfrac{465}{13} - \cdots + \dfrac{430}{20}$

7. The following programs calculate sums of fractions. Write the sums as in the above exercise, and also using sigma notation.

   (a)
```
« 0 DUP
  14 FOR a
     2 a ^ 15 a - SQ /
     -1 a ^ * +
  NEXT
»
```

   (b)
```
« 100
  1 99 FOR b
    100 b - b !  +
  NEXT
»
```

   (c)
```
« 63
  1 31 FOR i
    63 i 2 * - i !  / +
  NEXT
»
```

   (d)
```
« 1
  2 20 FOR j
    j !  2 j ^ 1 - /
    -1 j ^ * -
  NEXT
»
```

8. Write a program to calculate π using

$$S = \frac{1}{1^3} - \frac{1}{3^3} + \frac{1}{5^3} - \frac{1}{7^3} + \cdots$$

   where $\pi = \sqrt[3]{32S}$. Use 51 terms.

9. Write a program that calculates the value of π, with a precision of 0.01. Use `DO` and:
$$\pi = 4 - \frac{4}{3} + \frac{4}{5} - \frac{4}{7} + \frac{4}{9} + \cdots$$

10. Write a program that, when x is in the stack, returns e^x, with a precision of 10^{-11}. Use `WHILE`.
$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots$$

11. Supposing x is in the stack, write programs to calculate the following sums:
    (a) $S = 1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \cdots + \frac{1}{x}$
    (b) $S = x - \frac{x^2}{3!} + \frac{x^4}{5!} - \frac{x^6}{7!} + \cdots + \frac{x^{40}}{41!}$
    (c) $S = \frac{1}{x} + \frac{2}{x-1} + \frac{3}{x-2} + \cdots + \frac{x-1}{2} + \frac{x}{1}$

12. Write a program that calculates the approximate square root of a number Y in the stack, using Newton-Raphson's method:
    - the first approximation of $\sqrt{Y}$ is $X_0 = \frac{Y}{2}$
    - the successive approximations are $X_{n+1} = \frac{X_n^2 + y}{2X_n}$

13. Modify the above program so that it calculates the square root with a precision of 10^{-11}.

14. Write a program to find a root of an equation using Newton-Raphson's method:
$$X_{n+1} = X_n - \frac{f(X_n)}{f'(X_n)}$$
    Assume the equation is in level three, the variable in level two and an initial guess X_0 is in level one. Calculate 30 approximations.

15. Modify the above program to search for a root with a precision specified by the user in level one. All other inputs are shifted one level up.

6 Error handling

The HP48 User RPL programs are safe. Error handling is done automatically and the programmer doesn't have to worry about that. And, if some error condition occurs, for example, if there aren't enough arguments for a command, the only thing that will happen is that an error beep will be heard (unless disabled by the user) and a message will be displayed. The program will be stopped and the local variables will be removed. There is no reason to worry about that.

However, sometimes it is necessary to worry about errors that might happen. If your program, for example, needs a positive number as input, but a negative number is entered, probably no error will occur in the program, but the result might be wrong or meaningless. Because of this, there are some structure to handle errors or generate errors.

The first of these is `IFERRR...THEN...ELSE`. It's syntax is:

```
IFERR
      detection clause
THEN
      clause if error
ELSE
      normal clause
END
```

First, the ***detection clause*** is evaluated. If an error occurs, then ***clause if error*** is executed, and then the program continues after `END`. If no error occurred, ***normal clause*** is executed instead. You may omit the `ELSE` part, in this case no action will be taken if there was no error.

Tip: Stopping loops

It isn't written on the manual, but you can exit loops (`START`, `DO` and the others.) To do that, include the loop in an `IFERR` structure. Then, when you want to stop the loop, generate an error. Divide something by zero, or better yet, use `DOERR` (to be seen shortly.) The loop will be exited. The example below will not output all numbers:

```
« IFERR
        1 9 FOR i
          i
          IF i 5 > THEN
             0 DUP /
          END
        NEXT
  THEN
  END
»
```

But what if the program must take different actions based on the kind of error that occurred? Suppose the program should do something if there was an "Undefined Result" error and another action if there was an "Infinite Result" error. How can this be done? With the commands `ERRN`, `ERRM` and `ERR0`.

The first, `ERRN`, returns the number (in hexadecimal string format) of the last error. You can find a table of error values and their messages in Appendix B of the HP48 manual. `ERRM` returns a string with the error message. And `ERR0` clears the last error.

Last but not less important than the others, there is the command `DOERR`. As the name says, it is used to produce an error. It accepts different kinds of arguments: the number of the error (either as a real number or as an hexadecimal string), a string with a custom message, or 0. If you give a number (different from 0), that error will be generated. If you give a string, that string will be shown in the status area. And 0 can be used to stop a program, with no error message.

Tip: Another way to stop a program
Although `0 DOERR` can be used to stop a program while it is running, there is a smaller and faster way: simply use the `KILL` command where you want the program to be stopped.

7 Getting Input

The most simple and obvious way of getting input for HP programs is the stack. The user should put the arguments in the stack, and the results are returned to the stack. However, sometimes this is not the best way. When there are many arguments, it may be difficult for the user to remember the order, the type, and so on. So, there are better ways of getting inputs. And so there are better ways of displaying output, which we'll see on the next chapter.

7.1 Enter input the by the Command Line: INPUT

The first command to get input is **INPUT** (somewhat obvious, isn't it?) This command allows you to display a message and the user can edit a string using the command line. It takes two arguments: in level two, the message to be displayed at the top of the screen. It may be empty, and it may contain newline characters for multi-line messages.

In level one, there are two possible arguments: you can put a string with the initial text, which the user will modify (it can obviously be blank), or a list in the form

```
{ "string" { row column } mode(s) }
```

The arguments may be in any order, and you don't have to specify all the arguments above, just the ones you need.

- **"string"** is the initial text that will be at the command line.
- **row** and **column** specify where the cursor will appear (the default is at the end of the string) and whether it is in insert or overwrite mode (the default is insert). Row numbers start at one for the topmost row, and column numbers start at one for the leftmost character. Row 0 means the bottom row, and column 0 means the last character. Instead of specifying the row and column, you may substitute the list for a single real number, the character position counting from the first character. To make the cursor start in overwrite mode, make the first number negative.
- **mode(s)** are zero or more of the following:
  - A Greek alpha symbol (alpha left-shift a) will start the editor with the alphabetic keyboard on. This is very useful if the user is supposed to enter a string.
  - **ALG** will cause the editor to start with algebraic entry mode on.
  - **V** will check the syntax of the entire command line when ENTER is pressed, in the same way that the command line editor normally does, disallowing an edit if there is an RPL syntax error.

INPUT returns what the user entered as a string. Normally, you would use **OBJ→** to make it the way you need, for example a real number.

When the user presses CANCEL the first time during `INPUT`, the input line is cleared. If the user presses CANCEL again, the rest of the program is canceled.

An example of a program using `INPUT`:

```
« "Enter A, B, C"
   ":A:    :B:
  INPUT
»
```

(Insert newlines between the three lines of the input string.)

When executed, the program will show a screen similar to the one of picture 1.



Figure 1: `INPUT` example

The return string will be something like

```
":A:1
:B:2
:C:3"
```

which you can then use `OBJ→` on and you'll get

```
 4:
 3:  A: 1
 2:  B: 2
 1:  C: 3
```

7.2 Presenting options: CHOOSE

Another command related to input is `CHOOSE`. You may see an example of `CHOOSE` by pressing right-shift 7. Yes, it's a box with several options. They are very easy to create.

To use `CHOOSE`, you just need three arguments. In level three, a string with the title. If it's an empty string, the choose box will have no title and it will be

one line bigger. In level two, a list containing the options. And in level one a number corresponding to the initial position where the highlight should be.

If the user chooses an item, then CHOOSE will return the chosen item in level two and 1 (true) in level one. If the choose box was canceled, only 0 (false) will be returned. The chosen item will be returned as it was entered on the list. However, if you substitute each element of the list with a two-element list, then the first one will be shown, but the second will be returned.

Let's see an example of CHOOSE. The following program will display a choose box which lets you select your angle mode.

```
« "ANGLE MODE" @ Title
  { { " Degrees" DEG }
    { " Radians" RAD }
    { " Gradians" GRAD } } @ Items
  1 @ Initial position
  IF CHOOSE THEN EVAL END
»
```

7.3 Input forms: INFORM

And now, we'll see *the* command for getting input: INFORM. You may see an example of INFORM by pressing right-sift 7 ENTER. Yes, those menus are created by INFORM. It is the most complicated command of User RPL, but it is not so difficult.

INFORM takes five arguments. In level five, the title to be displayed at the top of the screen. In level four, the specification of the fields. In level three, the format. In levels two and one, respectively, lists with the values that will be used if the fields are reset and when they are first shown.

The field specification is a list in the form `{ field1 field2 ... fieldn}`, where each field is one of the following:

- `"label"`
- `{ "label" "help" }`
- `{ "label" "help" type1 type2 ... typen }`
- `{ }`

`"label"` is a string which will server as the title of the field, and which will be displayed near the field. `"help"` is the text that will be displayed on the bottom line when the field is specified (which should be a specification of what the field means.) The type specifications are zero or more real numbers representing the types of objects allowed in the field. If unspecified, all types of objects are valid. You can find a list of object types in Appendix H of the manual, under command TYPE. If a field specification is an empty list, the field immediately to the left is expanded to occupy the unspecified field space.

The format specification is one of the following objects:

- `{ }`
- `columns`
- `{ columns }`
- `{ columns width }`

Where `columns` is the number of columns that the form has. The default value is one. `width` is the tab width between the left edge of each title and each field. This allows a vertical alignment of the fields. The default is three.

The reset and initial values are either empty lists or list containing exactly one object for each field. These values will be used when one (or all) the fields are reset, or the first time the display is shown. If you want a field to be left blank, specify `NOVAL` as its value.

If the user filled in the form, a list with the values will be returned to level two, and 1 (true) to level one. The the form was canceled, 0 (false) will be returned to level one. A field that was left blank will be returned as `NOVAL`.

Let's see an example of `INFORM`, because I'm sure you didn't understand a thing of the above. But you will understand if you see an example. The program will display a dialog box like the one in picture 2, and will then calculate the distance of the points if the user doesn't cancel the form.

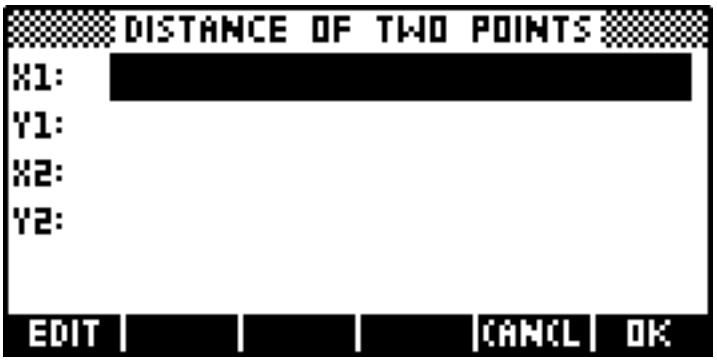


Figure 2: `INFORM` example

```
« "DISTANCE OF TWO POINTS" @ Title
  { { "x1:" "" 0 } @ Title, help, and only real numbers
    { "y1:" "" 0 }
    { "x2:" "" 0 }
    { "y1:" "" 0 } }
  { } @ The default format is good for us
  DUP DUP @ No reset or initial values
  IF
```

```
    INFORM @ Calculate only if not canceled
  THEN
    OBJ→ DROP @ Split the values
    ROT - SQ 3 ROLLD - SQ + √ @ Calculate
  END
»
```

Now, let's change a little our program, so that it displays with two columns, like the image shown in picture 3.

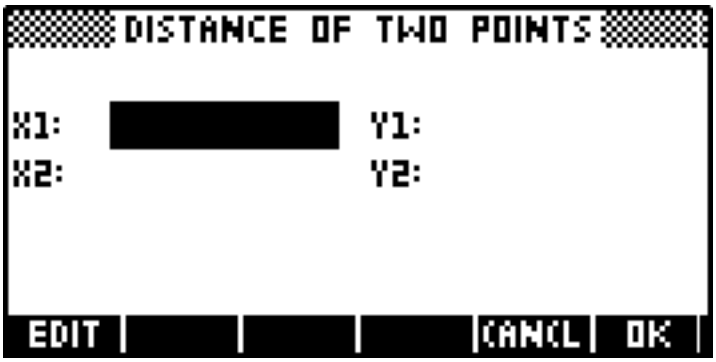


Figure 3: `INFORM` with two columns

```
« "DISTANCE OF TWO POINTS" @ Title
  { { } @ Blank line at top
    { "x1:" "" 0 } @ Title, help, and only real numbers
    { "y1:" "" 0 }
    { "x2:" "" 0 }
    { "y1:" "" 0 } }
  2 @ Two columns
  { } DUP @ No reset or initial values
  IF
  ⋮ @ The rest is equal
»
```

7.4 Getting key presses

To get individual key presses, we use a special form of the command `WAIT`. Normally, this command takes the number of seconds (from the stack) to delay program execution. However, if you give 0 as input for it, it will wait until a key is pressed, and will return the code of that key.

Key codes have the form `rc.p`, where `r` is the row number and `c` is the column number, counting from the top left key (the first softkey — A). `p` is a

number from one to six, representing the key modifiers. The values are listed in table 7.

Plane	Meaning
1	No modifier
2	Left-shift
3	Right-shift
4	Alpha
5	Alpha and left-shift
6	Alpha and right-shift

Table 7: Values for the plane

This way, MTH is key 21.1, RCL is key 32.3, x^2 is key 44.2, and so on.
The following program will wait for a key, and exit if ENTER is pressed:

```
« 0 WAIT
  IF
     51.1 ==
  THEN
     KILL
  END
  ⋮ @ Rest of program
»
```

Another command related to keys is `KEY` (no comments.) It is used in loops, when you want a loop to stop when a key is pressed. If no key is pressed while this command runs, it will return 0, otherwise it will return the code of the key. But the code contains no plane information, because the modifier keys also return a value (and thus probably stop the loop). The following program will loop until a key is pressed:

```
« DO
     ⋮ @ Your code here
  UNTIL
     KEY
  END
»
```

7.5 Exercises

1. Using three `INPUT`s, write a program to calculate the roots of a quadratic equation, return the values on the stack. The user should be prompted for a, b and c.

2. Write a program that uses `INFORM` to let the user enter a, b and c, and calculates the roots of the quadratic equation. The program then uses `INFORM` again to display the results. The screens should be like the ones in pictures 4 and 5.

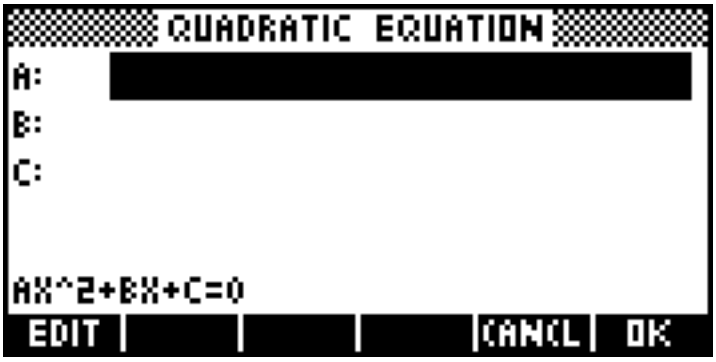


Figure 4: Input screen for exercise 2



Figure 5: Output screen for exercise 2

3. Write a program that creates a screen like the one shown in pictures 6 and 7 (see the different help) and then solves the system

$$\begin{cases} ax + by = u \\ cx + dy = v \end{cases}$$

and returns x and y to the stack.

4. Write a program that creates a screen like the one in picture 8 and calculates the inverse matrix. After calculations are done, use a screen like the input one to give the results.

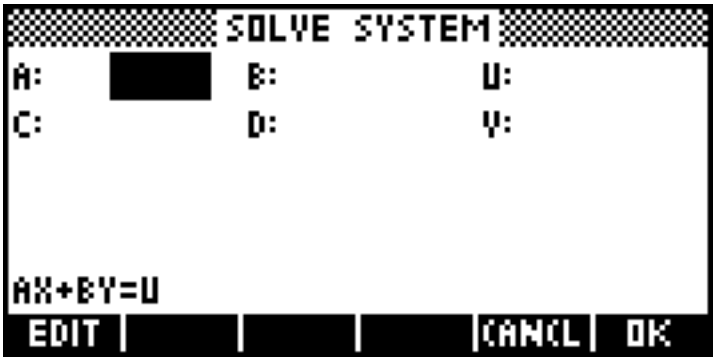


Figure 6: Input form for exercise 3

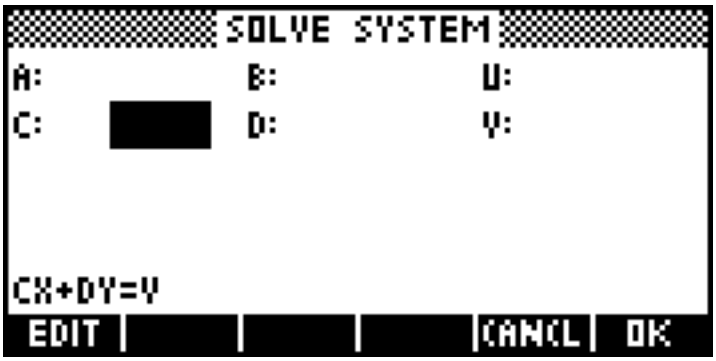


Figure 7: Input form for exercise 3

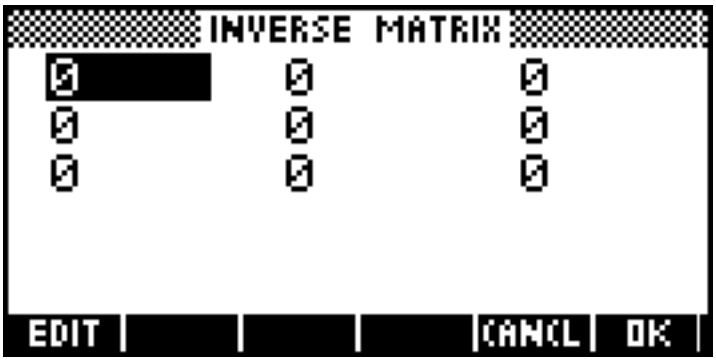


Figure 8: Input form for exercise 4

8 Displaying output

In this last chapter, you'll learn how to display the output of your programs in a nice way. Since your program has a nice way of getting input (by means of the commands seen on last chapter), it should also have a nice way of displaying output.

8.1 Displaying message boxes: MSGBOX

The command **MSGBOX** is very easy to use, for it takes only a single argument. It displays the string in a nice message box. The message box has at least two lines, so if your message is very short there will be a blank line. You may enter line-breaks in your string. If you don't, the text will be automatically wrapped.

8.2 Producing sounds

To make the HP produce a sound, use the **BEEP** command. However, If the beep has been deactivated, the beep will not sound. The command takes two arguments: in level two, the frequency of the sound, in Hertz. In level one, the duration of the sound. Quite simple.

8.3 Displaying text strings

The command used to display strings on the screen is **DISP**. It takes a string as argument (or any other object, which will be converted to a string) and the line number (from one to seven) from the stack. The object is displayed at the specified line. You may include line breaks in the string, this way you may display more than one line each time. Only the line(s) where the text will be displayed are cleared.

8.4 Using FREEZE

If you tried **DISP** (if you haven't, this is a very good time to do so), you probably noticed that the text only remains in the display for a very short period. To help this, there is the command **FREEZE**. It takes one parameter: a real number representing the area to freeze. The specified area(s) are not updated until a key is pressed. The possible values are listed in table 8.

8.5 Clearing the display

And the last command on this subject is **CLLCD**. OK, I should have finished with a big command, with many arguments, etc., but **CLLCD** is very simple. It takes no arguments and returns no values. It clears the whole screen (that's what the name stands for: clear LCD.) The stack, or **PICT** isn't actually cleared, just the display. It is normally used before **DISP**.

Number	**Area(s) frozen**
1	Status area
2	Stack
3	Status area and stack
4	Menu
5	Menu and status area
6	Menu and stack
0 or 7	Whole display

Table 8: Arguments for `FREEZE`

A Answers to exercises

Chapter 2 — First concepts

1. (a) `« 1 2 + »`
   (b) `« « 1 2 + » EVAL »`
   (c) 3
   (d) 3
   (e) 1 and `« 2 + »`
   (f) 3
2. `« 3 ^ π 4 3 * / →NUM »`
3. `« 32 - 5 * 9 / »`
4. `« 9 * 5 / 32 + »`
5. `« INV SWAP INV + INV »`
6. `« DUP2 * 3 ROLLD + / »`
   If you have `LASTARG` enabled, there is a shorter version:
   `« * LASTARG + / »`
7. `« 4 ROLL * 3 ROLLD * - »`
8. `« ROT - SQ 3 ROLLD - SQ + √ »`
9. `« 3 PICK 4 * * NEG OVER SQ + √ SWAP NEG + SWAP 2 * / »`
10. The most obvious solution is `« 3 PICK 4 * * NEG OVER SQ + √ SWAP NEG ROT 2 * DUP 3 ROLLD / 3 ROLLD / DUP2 + 3 ROLLD NEG + »`
    But Peter Karp sent two solutions that are shorter. Both work by changing the well known Bhaskara's formula into

$$x = \frac{-b}{2a} \pm \sqrt{\left(\frac{-b}{2a}\right)^2 - \frac{c}{a}}$$

    The first solution is: `« 2 →LIST SWAP / EVAL SWAP -2 / DUP SQ ROT - √ DUP2 + ROT ROT - »`
    and the other is: `3 PICK / SWAP ROT -2 * / DUP SQ ROT - √ DUP 2 + ROT ROT I »`
11. The most obvious solution is: `« 3 DUPN * SWAP ROT * + SWAP ROT * + 2 * »` . However, Joe Horn has a better solution. It is based on factoring the equation into $S = 2(d(h+w)+hw)$. Here is the program: notice the use of `LASTARG`: `« * LASTARG + ROT * + 2 * »`

Chapter 3 — Local variables

1.
```
« →a b u c d v
  « 'a*d-b*c' EVAL
    '(d*u-b*v)' EVAL OVER /
    '(a*v-c*u)' EVAL ROT /
  »
»
```

   Of course, you can use RPN notation if you prefer:

```
« →a b u c d v
  « a d * b c * -
    d u * b v * OVER /
    a v * c u * ROT /
  »
»
```

2. (a)
```
« →a b c d e f g h i
  'a(e*i-f*h)+b*(f*g-d*i)+c(d*h-e*g)'
»
```

   (b)
```
« →a b c d e f g h i
  « a e i * f h * - *
    b f g * d i * - * +
    c d h * e g * - * +
  »
»
```

3.
```
« →a b c d e f g h i
  « e i * f h * - *
    b i * c h * - * NEG
    b f * c e * - *
    d i * f g * - * NEG
    a i * c g * - *
    a f * c d * - * NEG
    d h * e g * - *
    a h * b g * - * NEG
    a e * b d * - *
    9 →LIST
    a e i * f h * - *
    b f g * d i * - * +
    c d h * e g * - * +
    / EVAL
  »
»
```

Chapter 4 — Conditional tests

1. (a) 0
   (b) 1
   (c) 0
   (d) 1
2. (a) 0
   (b) 0
   (c) 1
   (d) 0
3. (a) 1
   (b) 0
   (c) 1
   (d) 0
   (e) 0
   (f) 1
   (g) 0
4. 0
5. (a) `Command1` and `Command5`
   (b) `Command3`, `Command4` and `Command5`
   (c) 0, 0 and any other value
   (d) `Command2` and `Command 5`
6. 2.22222....
7. ```
   « IF DUP FS? THEN
        CF
     ELSE @ Since the flag is set, we don't need to set it again,
        DROP @ just drop its number
     END
   »
   ```
8. ```
   « IF THEN @ If the value is true
        SF @ Set
     ELSE
        CF @ Else clear
     END
   »
   ```

9.
```
« → l m n
  « IF l m > l n > OR THEN
       IF m n < THEN
          l m 'l' STO 'm' STO
       ELSE
          l n 'l' STO 'n' STO
       END
    END
    IF m n > THEN
       m n 'm' STO 'n' STO
    END
    l m n
  »
»
```

10.
```
« ROT 2 -
  IF DUP 0 < THEN @ January or February?
     12 + @ Correct month
     SWAP 1 - SWAP @ Correct year
  END
  3 ROLLD
  100 / DUP FP 100 * SWAP IP @ Split year
  DUP 4 / IP SWAP 2 * - SWAP DUP 4 / IP
  + + + SWAP 2.6 * .2 - IP + 7 MOD @ Apply formula
  → dw
  « CASE
        dw 0 == THEN "Sunday" END
        dw 1 == THEN "Monday" END
        dw 2 == THEN "Tuesday" END
        dw 3 == THEN "Wednesday" END
        dw 4 == THEN "Thursday" END
        dw 5 == THEN "Friday" END
        dw 6 == THEN "Saturday" END
        "Other day?!"
    END
  »
»
```

This program can be improved. First, we can make Saturday the default clause because there will never be another day. But it can be improved even further, without `CASE`. But that remove the purpose of the exercise. Anyway, you can replace `→ dw « ...»` with

```
{ "Sunday"
  "Monday"
  "Tuesday"
  "Wednesday"
```

```
  "Thursday"
  "Friday"
  "Saturday" }
SWAP 1 + GET
```

Chapter 5 — Loop structures

1. `{ 100 81 64 49 36 25 16 9 4 }`

2. ```
   « { } 10
   WHILE
         DUP 1 ≠
   REPEAT
         DUP 1 - 3 ROLLD SQ + SWAP
   END
   DROP
   »
   ```

3. ```
   « 1 @ Number that will be multiplied
   2 @ Count
   → n f i
   « WHILE
           i n ≤
     REPEAT
           'f' i STO*
           'i' 1 STO+
     END
     f @ Output the factorial
   »
   »
   ```

   Or using `DO`:

   ```
   « 1 2 → n f i
     « DO
          'f' i STO*
          'i' 1 STO+
       UNTIL
          i n >
       END
       f
     »
   »
   ```

4. ```
   « 0 @ Elapsed years
   3E7 2E8 @ Initial populations
   ```
   ```

```
  →ny pa pb
  « DO
       'pa' 1.03 STO*
       'pb' 1.015 STO*
       'ny' 1 STO+ @ Increase populations and number of years
    UNTIL
    pa pb ≥
    pa "Pop. A" →TAG
    pb "Pop. B" →TAG
    ny "No. years" →TAG
  »
»
```

5.
```
« 0 @ Time elapsed
  →m t
  « WHILE
         m .5 ≥
    REPEAT
         'm' 2 STO/ @ Halve mass
         't' 50 STO+ @ Increment time
    END
    @ Cute display for mass
    m 1_g →UNIT "Mass" →TAG
    t @ Let's convert no. seconds to hours, minutes & seconds
    3600 / IP "h " + @ Hours
    t 3600 MOD DUP 't' STO @ Seconds - hours
    60 / IP "mină" + + @ Minutes
    t 60 MOD "s" + + @ Finally
  »
»
```

6. (a)
```
« 1 @ First item
  2 50 FOR i
    i 2 * 1 - i / + @ Calculate term and add
  NEXT
»
```

(b)
```
« .04 @ First item
  2 50 FOR j
    2 j ^ 51 j - / +
  NEXT
»
```

(c)
```
« 1406 @ First item
  36 1 FOR k
     k k 1 + * 38 k - / +
  -1 STEP
»
```

You could also change it to calculate in reverse order, so that it isn't necessary to use `-1 STEP`.

(d)
```
« 1
  2 10 FOR i
    i i SQ / @ Absolute value
    -1 i ^ * - @ Calculate sign and add
  NEXT
»
```

(e)
```
« 1000
  2 50 FOR a
    1000 a 1 - 3 * - a /
    -1 k ^ * -
  NEXT
»
```

(f)
```
« 48
  1 10 FOR n
    480 n 5 * - n 10 + /
    -1 n ^ * +
  NEXT
»
```

7. (a) $S = \dfrac{1}{196} - \dfrac{2}{169} + \dfrac{3}{144} - \cdots - \dfrac{14}{1}$

$$S = \sum_{a=1}^{14} \frac{(-1)^a(-a)}{(15-a)^2}$$

(b) $S = \dfrac{100}{1} + \dfrac{99}{1} + \dfrac{98}{2} + \dfrac{97}{6} + \dfrac{96}{24} + \cdots + \dfrac{1}{99!}$

$$S = \sum_{b=0}^{99} \frac{100-b}{b!}$$

(c) $S = 63 + \dfrac{61}{1!} + \dfrac{59}{2!} + \dfrac{57}{3!} + \cdots + \dfrac{1}{30!}$

$$S = \sum_{i=0}^{31} \frac{63-2i}{i!}$$

(d) $S = \dfrac{1!}{1} - \dfrac{2!}{3} + \dfrac{3!}{7} - \dfrac{4!}{14} + \dfrac{5!}{31} - \cdots + \dfrac{20!}{1048575}$

$$S = \sum_{j=1}^{20} \frac{(-1)^j(-j!)}{2^j-1}$$

8.
```
« 1
  3 105 FOR k
    k 3 ^ -1 k 2 / CEIL ^ * INV -
  2 STEP
  32 * 3 XROOT
```

```
»
```

9.
```
« 4 3 →d
  « DO
       4 d / -1 d 2 / CEIL ^ * - 2 'd' STO+
    UNTIL
       DUP π →NUM - ABS .01 <
    END
  »
»
```

10.
```
« 1 →x d
  « 1 WHILE
      DUP x EXP - ABS 1E-11 >
    REPEAT
      x d ^ d !  / +
      'd' 1 STO+
    END
  »
»
```

11. (a)
```
« 1 2 ROT
  FOR n
      n INV +
  NEXT
»
```

(b)
```
« DUP →x
  « 1 20 FOR i
      x i 2 * ^ i 2 * 1 + !  /
      -1 i ^ * +
    NEXT
  »
»
```

(c)
```
« DUP INV SWAP →x
  « 2 x FOR n
      n x n 1 - - / +
    NEXT
  »
»
```

12.
```
« DUP 2 / SWAP →y
  « 1 20 START
    DUP SQ y + SWAP 2 * /
  NEXT
  »
»
```

13.
```
« DUP DUP √ SWAP 2 / 3 ROLLD →y r
  « DO
       DUP SQ y + SWAP 2 * /
    UNTIL
       DUP r - ABS 1E-11 <
    END
  »
»
```

14.
```
« ROT DUP DUP STEQ @ Save copy of equation in EQ
  EQ→ - @ Split and subtract, to zero one side
  DROP ROT DUP2 DUP PURGE ∂ @ Calculate derivate
  →f v d @ Save function, variable and derivate
  « 1 30 START
      DUP v STO f EVAL d EVAL / - @ Calculate approximation
    NEXT
  »
»
```

15.
```
« 4 ROLL DUP DUP STEQ OBJ→ SWAP DROP
  EQ→ -
  DROP 4 ROLL DUP2 DUP PURGE ∂ →e f v d
  « DO
       DUP v STO
       f EVAL DUP d EVAL / ROT SWAP -
    UNTIL
       SWAP ABS e <
    END
  »
»
```

Chapter 7 — Getting input

1.
```
« "Enter A:" "" INPUT OBJ→
  "Enter D:" "" INPUT OBJ→
  "Enter C:" "" INPUT OBJ→
  3 PICK 4 * * NEG OVER SQ + √  SWAP NEG ROT
  2 * DUP 3 ROLLD / 3 ROLLD / DUP2 + 3 ROLLD NEG +
»
```

2.
```
« "QUADRATIC EQUATION"
  { { "A:" "AX^2+BX+C=0" 0 1 9 13 }
    { "B:" "AX^2+BX+C=0" 0 1 9 13 }
    { "C:" "AX^2+BX+C=0" 0 1 9 13 } }
  { } DUP DUP
  IF INFORM THEN
     OBJ→ DROP 3 PICK 4 * * NEG OVER SQ + √
```

```
      SWAP NEG ROT 2 * DUP 3 ROLLD / 3 ROLLD / DUP2 +
      3 ROLLD NEG +
      2 →LIST
      "SOLUTION"
      { "X':" "X'':" }
      { } DUP
      5 ROLL
      IF INFORM THEN DROP END
   END
»
```

3.
```
« "SOLVE SYSTEM"
  { { "A:" "AX+BY=U" 0 1 9 13 }
    { "B:" "AX+BY=U" 0 1 9 13 }
    { "U:" "AX+BY=U" 0 1 9 13 }
    { "C:" "CX+DY=V" 0 1 9 13 }
    { "D:" "CX+DY=V" 0 1 9 13 }
    { "V:" "CX+DY=V" 0 1 9 13 } }
  3 { } DUP
  IF INFORM THEN
     OBJ→ DROP
     →a b u c d v
     « 'a*d-b*c' EVAL
       'd*u-b*v' EVAL OVER /
       'a*v-c*u' EVAL ROT /
     »
  END
»
```

4.
```
« "INVERSE MATRIX" DUP
  { { "" "" 0 1 9 13 }
    { "" "" 0 1 9 13 }
    { "" "" 0 1 9 13 }
    { "" "" 0 1 9 13 }
    { "" "" 0 1 9 13 }
    { "" "" 0 1 9 13 }
    { "" "" 0 1 9 13 }
    { "" "" 0 1 9 13 }
    { "" "" 0 1 9 13 } } DUP 3 ROLLD
  { 3 1 } DUP 4 ROLLD
  { 0 0 0 0 0 0 0 0 0 } DUP DUP 6 ROLLD
  IF INFORM THEN
     OBJ→ DROP
     →a b c d e f g h i
     « e i * f h * - *
       b i * c h * - * NEG
       b f * c e * - *
```

```
      d i * f g * - * NEG
      a i * c g * - *
      a f * c d * - * NEG
      d h * e g * - *
      a h * b g * - * NEG
      a e * b d * - *
      9 →LIST
      a e i * f h * - *
      b f g * d i * - * +
      c d h * e g * - * +
      / EVAL
    »
    9 →LIST
    IF INFORM THEN DROP END
  END
»
```