



:R4 Human Oriented Language

[http:// www.reda4.org](http://www.reda4.org)
pabloreda@gmail.com

Pablo Hugo Reda
Maria Jose Aguirre
Jose Maria Fantasia
Sebastian Desimone
Javier Gil Chica
Dutra da Lacerda

Translation by Manuel Cornes - June 2008

This Software is licenced under the CC-GNU GPL agreement

Introduction

The constant introduction of new computer-related technologies, each time more powerful and cheap has lead programmers, for market reasons, to use programming languages and other means of development that depart from the most basic microprocessor-based principles of programming. This has introduced complexity and high-cost, in terms of efficiency and development difficulties that are offset by a growing use of resources.

In the late 60s, Charles Moore created the Forth language using a stack machine paradigm, with a perspective different from most other languages. Forth forces you to see the construction of a system as a process of synthesis more than the addition of components. This goes against the common practice of hiding the hardware from the programmer with virtual layers, as this only makes problems worse.

Currently, the Forth programming language does not thrive in software development but it does so in hardware, as microcontrollers benefit from FORTH's unique characteristics and the brevity of the generated code. The software industry keeps ignoring it for reasons that are hard to define precisely.

End of 2005, the need for an alternative to the problematic C language, with its incoherent accepted practices (don't use global variables) and its goal of producing optimized code (use global variables), lead the author to the study of Forth.

At first the goal was to create a new language by adding features to FORTH, but after a while the perspective changed and the opposite route was considered : the goal became a search for something simpler.

:R4 is a language inspired by ColorForth, with a basic growable dictionary of 109 words, a data stack, a return stack and 6 prefixes with different functions.

This manual is the first documentation for the language. You will find here a description of how it works, the behaviour of each word, some examples and a quick reference to the language.

Human Manual for learning R4

We use a computer when we follow its rules, for example: you press the save button... to save something.

You program a computer when you create the way in which the machine will behave.

Just like human beings communicate with a language, to tell a machine what we want it to do we also need a language. The programming language is what

enables to express the desired behavior of the computer.

Programming in :R4 consists in creating a set of words representing actions and data to build that behavior.

So the program we are going to write will be composed of a set of words. A word, in :R4 is any set of letters and symbols separated by space. Example: + (plus) is a word, the distinction between signs and letters is irrelevant here. Also, you have to remember that :R4 does not distinguish between uppercase and lowercase in the definition and use of words, the only exception being the use of " (double quotes) that are used to represent this difference.

There is a base dictionary, with a starting set of words to be used for both the construction of a program and/or the definition of new words.

Code and the data stack

Code is written like a script to program the computer. :R4 code is stored in text files. When a number is specified in the program to be processed by the machine, this number is added to the stack (pushed on the stack). The stack is a sequence of numbers that allows adding and removing numbers from its top.

We start with an empty stack. The base of this empty task is drawn with no numbers.

```
(
```

Then, if a number is added to the code, 3 for example, the drawing of the stack becomes:

```
( 3
```

When you add another number, for example 2, the drawing of the stack is now:

```
( 3 2
```

The above example written in a source file is:

```
test.txt
```

```
3 2
```

It creates the following stack:

```
( 3 2
```

Even if the drawing of the stack and the source code are almost the same, with the exception that the stack base "(" is not drawn in the code, they are different things and in different places.

The source code is written by the programmer. The stack is built by the computer

at the moment when it is evaluated. So the programmer builds the stack only indirectly.

It is important to always have in mind the relation between code and the computer behavior. In fact, a script is written for the machine to execute, and the code from this script is the product of the programmers ingenuity to define the desired behaviour.

When the next instruction in the code is a number, it is pushed onto the stack. When instead, it is a word then the dictionary is used.

The dictionary is a list of words paired with their meaning. When one writes a word in the code, it is searched in the dictionary and if found, the meaning of that word is evaluated. It is important to remember that this evaluation does not happen when a word is written in the code, but rather when the computer executes it.

When a word does not exist in the dictionary but is used in a program, an error message indicates it. All words to be used must be defined before they are used.

When code execution starts, the dictionary contains a small set of basic words that allow to make the computer do what we need it to do. These words are classified by function : Stack manipulation, Arithmetical operations, Logical operations, Memory access, and so on.

Words to manipulate the stack

DROP

removes the top of the stack

3 2 drop

(3

given that 2 is removed by drop.

SWAP

Swaps the first and second numbers on the stack

3 2 swap

(2 3

DUP

Duplicates the top of the stack

3 2 dup

(3 2 2

OVER

Adds a copy of the second number on the stack

```
3 2 over
```

```
( 3 2 3
```

Notice that the code and the stack are now different. Furthermore the computer will process our words or numbers step by step, which means that each word acts on the stack at the moment when it is evaluated by the machine and at the end of the evaluation of all words the indicated stack is obtained.

It is important to know at each moment the stack status, as each word can produce change. It is convenient to have a way to represent this change and for this purpose one can use a stack change notation.

This change notation is the following :

```
| before -- after
```

One adds the | symbol to indicate that starting from this character in the code and until the end of the line, anything the programmer writes is going to be a comment and will be ignored by the machine, in other words, it's a comment for the programmer.

We will write in 'before', the status of the stack BEFORE the execution of the word followed by -- and then what it will be left AFTER execution.

The values on the stack are represented by letters and the quantity of letters before and after will depend on the action of the word.

The full listing of the stack manipulating words follows hereafter

```
DUP | a -- a a
```

```
DROP | a --
```

```
OVER | a b -- a b a
```

```
PICK2 | a b c -- a b c a
```

```
PICK3 | a b c d -- a b c d a
```

```
PICK4 | a b c d e -- a b c d e a
```

```
SWAP | a b -- b a
```

```
NIP | a b -- b
```

```
ROT | a b c -- b c a
```

```
2DUP | a b -- a b a b
```

```
2DROP | a b --
```

```
3DROP | a b c --
```

```
4DROP | a b c d --
```

2OVER | a b c d -- a b c d a b
2SWAP | a b c d -- c d a b

One can say that the stack is the short term memory of the computer. Eventhough at the beginning it results awkward to think about the stack, it is the key to the whole language, don't despair if at the beginning you find it a bit complex, practice makes this mechanism transparent and converts it into a common operation.

Exercices (Solutions are given at the end of the document)

With numbers, DROP, SWAP, DUP and OVER write the code to obtain the following stacks using the least number of numbers.

- 1.- (2 3 2 3 2 3
- 2.- (2 2 2 2 2 2
- 3.- (1 2 3 4 5 6
- 4.- (
- 5.-)

Words for calculations

Words used for arithmetic calculations are commonly called signs, but as we have seen already this distinction is not necessary in our case.

+ (sum sign)

2 3 +
(5

The + word (sum) takes the first two numbers on the stack, adds them and puts the result on the stack (after the two previous values have been removed). The rest of the basic operations work identically with their respective meanings.

+	a b -- c	c = a + b
-	a b -- c	c = a - b
*	a b -- c	c = a * b
/	a b -- c	c = a / b
*/	a b c -- d	d = a * b / c (64 bits mult)
*>>	a b c -- d	d = (a * b) >> c (64 bits mult)
/MOD	a b -- c d	c = a / b d = a remainder b
MOD	a b -- c	c = a remainder b
NEG	a -- b	b = -a
1+	a -- b	b = a + 1
1-	a -- b	b = a - 1
2/	a -- b	b = a / 2

```

2*   | a -- b   b = a * 2
<<  | a b -- c   c = a << b shifts bits to the left
>>  | a b -- c   c = a >> b to the right(carry sign)

```

Logic words

Much like arithmetic words, one can use a number of words for logic calculations.

```

AND  | a b -- c   c = a AND b
OR   | a b -- c   c = a OR b
XOR  | a b -- c   c = a XOR b
NOT  | a -- b     b = NOT a

```

Arithmetic words and logic words do their calculations on the data stack, as we can see, any calculation happens there.

Growing the dictionary

As previously said, the dictionary comprises a set of basic words. Programming in :R4 consists in defining words and for this purpose one uses the : prefix (colon), any word starting with a : defines itself and is added to the dictionary.

The word :hello written in the code doesn't mean look for :hello but that the HELLO word should be added to the dictionary (without the : prefix) and that its definition will follow until the ; word (semi-colon). It is very important to remember about the spaces. Let's have a look at a definition in a piece of code.

```
:s3 3 + ;    | a -- b    ( b will be a+3 )
```

This doesn't mean that with this code the stack will contain a number resulting from the addition of 3 to its top value, but rather that the s3 word will be defined in the dictionary. The behaviour will be computed when called AFTER it has been defined.

```
:s3 3 + ;
```

```
2 s3
```

Then yes, the sum is calculated. Note that eventhough the s3 word does not exist in the basic dictionary, once defined it can be used (it always has to be defined before it is used).

: (colon), exists in :R4 as a word and as a prefix, we have seen already that it can be used to define new words when used as a prefix. At the end of the code, one can define where execution of the program will start with the word : (colon), the code defined before then will be ready when the program starts.

```
:s3 3 + ;
```

```
: 2 s3 ;
```

```
( 5
```

Names are assigned to definitions : (colon) which are actions, programming is creating those names and their definitions. Each definition takes or produces numbers that are managed on the stack, that's what the stack notation documents.

It is important that the correct names are found for each situation.

Exercices (Solutions are at the end of the document)

Define the words to obtain the following stack changes :

1.- Define square of a number with the following behaviour:

```
:**2 | a -- a*a
```

2.- Define the sum of 3 numbers at the top of the stack with the following behaviour:

```
| a b c -- a+b+c
```

3.- Suppose that 2drop and 2dup don't exist in our dictionary. Come up with a definition of these words validating your answer with the stack change notation.

4.- Write a word that inverts the order of the numbers on the following stack

```
| 1 2 3 4 --- 4 3 2 1
```

5.- Write the word 3dup that duplicates the 3 last numbers of the stack

```
| 1 2 3 -- 1 2 3 1 2 3
```

6.- Define -rot as follows

```
:-rot | a b c -- c a b
```

7.- Define the following formulas and show the effects on the stack,

a) a^2+b^2 | a b -- c

b) a^2+ab+c | c a b -- d

c) $(a-b)/(a+b)$ | a b -- c

8.- Draw a picture you like to forget about all these calculations.

Program variables

Just as the : (colon) prefix is used to define words that will perform some type of action, the # prefix (hash) is used to define words that will keep data, also called variables.

```
#datum
```

The line above defines the word datum or the datum variable as a memory cell that enables keeping a number, which can be initialized as follows :

```
#datum 22
```

One needs two words in the dictionary to write a number in a variable and read the value that is stored, as follows :

Read Memory or Fetch

```
@ | address -- value
```

Returns the value that is stored in the specified memory address.

Write Memory or Store

```
! | value address --
```

Writes the value in the specified address.

It is useful at this stage to learn about another prefix ' (simple quote). It is used to get the address of an already defined word, as assigned by the language. This address is used to put or get values of variables.

```
#datum 8
```

```
'datum @ | pushes an 8 on the stack
```

```
2 'datum !
```

```
'datum @ | pushes a 2 this time
```

A word defined with a # works differently than a word defined with a : . A : word is called when its name is written in the code, however, a variable (word defined with a #) puts its value on the stack when written in the code.

It follows that @ is not necessary most of the time, given that datum will push its value, not its address on the stack. The two following lines have the same result, because the second line pushes the address and obtains the value with @.

```
datum | pushes datum's value
```

```
'datum @ | pushes datum's value
```

In the example above the word datum, that contains a value, is defined. For the sake of simplicity one says that the variable datum is defined. Variable datum's

value is the number that this variable contains and the address of this variable is the physical position in the memory at which this variable is kept. Note that the address is also a number but we will never be needing its value other than through its name (in this case 'datum').

The code:

```
0 10 !
```

is valid, but utterly useless (unless memory address 10 actually means something on the computer that is being programmed).

A very common construct consists in changing the value of a variable with a variation of its own value, for example, to add 1 to the value of variable datum, one should write :

```
#datum 0  
:add1 datum 1 + 'datum ! ;
```

This definition can be rewritten as follows

```
#datum 0  
:add1 1 'datum +! ;
```

Given that +! increments the value of the memory's address.

The dictionary definitions and :R4's variables are in the memory, when asked for the address of a word the system refers to the memory where it was located by the language.

It is important to practice using variables, values and addresses, a drawing always helps understand the differences.

When defining variables, one can use several numbers, for example :

```
#list 10 20 30 40 50
```

Here list is the first number, that is 10. How do you think the rest can be obtained?

Well done, with the address of the list variable.

```
'list 4 + @
```

In this case the second number, or 20, will be pushed on the stack. If we replace the 4 with an 8 we will obtain the third number etc etc sequentially. To

understand why 4 instead of 1, we need to understand how numbers are kept in memory.

:R4 defines each number with 4 bytes, but it is possible to use less bytes in memory. Just like @ takes 4 bytes, c@ takes 1 byte and w@ takes two bytes. There is also c! that writes one byte and w! that writes 2 bytes.

One should keep these quantities in mind as we will be working with addresses and they will be useful to calculate correct memory addresses.

Number bit sizes can be indicated with square brackets and parentheses.

```
#32bits 1 2 3 | 32 bits numbers  
#16bits [ 1 2 3 ] | 16 bits numbers  
#8bits ( 1 2 3 ) | 8 bits numbers
```

The first variable occupies $4*3=12$ memory bytes, the second one 6 bytes and the third 3 bytes

Another notation can be used to ask for specific quantities of memory bytes, for example to define a 1k memory location :

```
#of1KBYTE )( 1024
```

Whereas numbers are kept on 4 bytes, characters use 1 byte.

A further detail regarding handling of numbers in :R4, two prefixes exist to specify binary and hexadecimal number bases, they are % and \$ respectively :

Decimal	Binary	Hexadecimal
0	%0	\$0
2	%10	\$2
10	%1010	\$A
15	%1111	\$F
16	%10000	\$10

All what the computer does, keeps or shows is in memory, there also is an empty memory space to work, without it the computer could not create but only show.

The only word that is in the base dictionary is MEM, it leaves on the stack the free memory starting address, you can do what you want with this memory, it's all yours, it came with the computer.

Control structures

Decision is the mechanism by which an option or a path is chosen. To represent

a decision one specifies what the condition is and then which path or words will be called according to this condition.

Guess where the condition is computed, ... yes, on the data stack. Four words enable that.

- 0? Jump if the top of the stack is 0
- 1? Jump if the top of the stack is NOT 0
- +? Jump if the top of the stack is positive
- ? Jump if the top of the stack is negative

These words use the top of the stack for their comparisons, without modifying it.

A second set of words compare the top two numbers of the stack, removing the first one as a side effect.

- =? | a b -- a a = b ?
- <>? | a b -- a a <> b ?
- >? | a b -- a a > b ?
- <? | a b -- a a < b ?
- <=? | a b -- a a <= b ?
- >=? | a b -- a a >= b ?
- and? | a b -- a test b bit in a
- nand? | a b -- a test not b bit in a

Conditional jump

Parentheses delimit the aforementioned jump, but don't forget that parentheses also are words, for example :

0? (drop)

This piece of code removes the top of the stack if it is 0, it won't do anything otherwise as the drop won't be executed. This construct is called IF, to reflect the type of condition.

Another type of conditional jump specifies two possible actions, when the condition is true and when it isn't.

+? (dup)(drop)

One should think in terms of word blocks in-between parentheses. Although "(" (open parenthesis), ")" (close-open parenthesis) and ")" (close parenthesis) are three words. In the previous example, when the top of the stack contains a negative number, then a JUMP to the next word of ")" happens, otherwise (when it is positive) execution goes on until ")" and then jumps to the ")". As a

result, if the top of the stack contains a positive number then the code above duplicates it, otherwise it is removed. Word blocks should always be closed, leaving a block open is an error and will be detected by the language.

Repeat forever

To repeat a list of words one should enclose them inside parentheses, for example :

```
:clean | ... clean is defined ;  
:dirty | .. dirty is defined ;  
: ( dirty clean ) ;
```

This program, when executed, will repeat the two words dirty and clean for ever.

Repeat while

This loop is directly named after how it works, it is more useful to have a stop condition than to have something repeat indefinitely, notice that the position of the condition defines the type of loop :

```
:printn | n -- Print the top of the stack and remove it ;  
: 5 ( 1? )( 1 – dup printn ) drop ; |print 4 3 2 1 0
```

The last line reads: push 5 on the stack, while the top of the stack is not 0 REPEAT...subtract 1, copy it on the top of the stack and print it. It should be duplicated because printing an element removes it from the (top of) the stack.

As you can see, the status of the stack should be taken care of at each repetition, conditions that remove from the stack are generally easier to read:

```
: 0 ( 5 <? )( 1 + dup printn ) drop ; |print 1 2 3 4 5
```

This reads : push 0 on the stack and WHILE it is less than 5 add 1 to it and print it (we know already why we duplicate it before we print it).

Repeat until

```
: 5 ( 1 – dup printn 0? ) drop ; | print 4 3 2 1 0
```

This loop will repeat UNTIL the top of the stack is 0.

These control structures have to be nested correctly, in other words, there should be the same number of opening and closing parentheses, furthermore a “;” (semi colon) inside parentheses does not finish the definition but only the execution of the corresponding word.

Vectors

A vector is a variable that contains an address, but the code of a word is stored at this address not data, this enables changing its meaning at any time and is a powerful construct. An example follows:

```
:sum1 1 + ;  
:sum2 2 + ;
```

```
#sum
```

```
:define1 'sum1 'sum ! ;  
:define2 'sum2 'sum ! ;
```

```
: 3 define1 sum exec define2 sum exec ;
```

(6

There is a way to define actions without giving them names, they are called anonymous words, these definitions are meaningful when we need to define an action that is used only once, it is then referenced by its address. Notice here that this address is called with EXEC, either immediately or in another definition

Square brackets are used to build this type of definition

```
[ exit ; ] >esc< | action assigned to the escape key
```

'[' and ']' are both words, the first one means that the code that follows won't execute immediately, the second one closes the code and pushes on the stack the address of the code so defined.

Prefixes

The 6 existing language prefixes are :

^ Include vocabulary

^module.txt | include the module.txt file

The ^ prefix is used to insert code into a file, this is useful to create words in a file and use them in another file, this way you don't have to re-write code, this enables the creation of re-usable lexicons of words or specific vocabularies.

Not all words from included files are available, only those that are exported.

: (colon) Define action

:new | define new as the list until the ;
::newg | define and export this definition

The definition of words with : (colon) assigns the list of words that follow to the name defined with the prefix.

:: (Double colon) defines a word that's exported, it is usable in other files when the file in which it is defined is included by another program.

(numeral) Define data
#memory | defines memory variable
#:memoryg | define and export a variable

Variable definitions with a '#' (hash sign) take the optional numbers that follow as their contents, square brackets “[]” change the width to 16 bits and parentheses “()” to 8 bits. Finally the word)(allows to specify the quantity in bytes that are to be reserved for the defined variable.

#: (hash-colon) exports the variable.

' (simple quote) Pushes the address of the word on the stack
'hello | pushes the address of the already defined hello

| (pipe sign) Comment
|blabla Comment, ignore everything until end of line

“ (double quote) Defines text string memory (that can contain spaces) and pushes its address on stack
"bla bla" | text (including spaces), push address on stack

Up to here basic structures of the language have been described, the other words described in what follows complete the base dictionary.

Fixed Point

Decimal numbers are awolled trught fixed point representation,
for this :R4 convert a number-dot-number to a 16.16 bits fixed point.

2.0 is \$20000
1.0 is \$10000
0.5 is \$7ffff

add and substract not need an adjust

2.0 1.5 + 0.03 -

The multiply need 16 righth shift.

```
.*      | f f -- f  
        16 *>> ;
```

Division need 16 left shift the one parameter

```
:/      | f f -- f  
        swap 16 << swap / ;
```

The System

The following group of words handle system related information, the first three words obtain information and the following two control the virtual machine :

```
MSEC      | -- a  
Pushes the current system milliseconds on the stack
```

```
TIME      | -- h m s  
Pushes the system's hour, minutes and seconds on the stack
```

```
DATE      | -- d m a  
Pushes the day, month and year of the system on the stack
```

```
END        | --  
Stops the virtual machine, when :R4 eventually runs on a real machine this will shutdown the computer, for now it simply returns to the underlying Operating System that executes the virtual machine.
```

```
RUN        | d -  
Loads the code specified by its file name and executes it. This enables chaining execution programs.
```

The screen

The computer screen is a matrix of lights in which each light, called a pixel, can take any color.

Width and Height of the screen can be set by the :R4 user, for now three versions exist : 640x480, 1024x768 and 1280x1024.

When more computing power is needed a smaller screen can be used so that less effort is required for the drawing.

```
SW         | -- w ScreenWidth
```

SH	-- h	ScreenHeight
CLS	--	Erase screen with background color
REDRAW	--	Copy virtual screen to real screen
FRAMEV	-- a	Start of Video Memory
UPDATE	--	Update operating system internal events

Drawing

The following words can be used to draw on the screen. Coordinates are passed through the stack and the numbers correspond to physical positions on the screen, it is not appropriate to use constants here, but rather variables calculated by taking into account the size of the screen so that the drawing does not depend on a particular resolution.

OP	x y --	Origin point
CP	x y --	Curve control point
LINE	x y --	Draw a line
CURVE	x y --	Draw a curve
PLINE	x y --	Draw polygon line
PCURVE	x y --	Draw polygon curve
POLI	--	Draw polygon
PAPER	a --	Define background color
INK	a --	Define drawing color
INK@	-- a	Pushes drawing color on the stack
ALPHA	a --	Define color transparency

Interaction

For interaction, actions are assigned to the keyboard and the mouse.

XYMOUSE	-- x y	Mouse position
BMOUSE	-- b	Mouse status
KEY	-- s	Last key presses
IPEN!	v --	Set Mouse event (every change in mouse call this vector)
IKEY!	v --	Set Key event (every change in keyboard call this vector)

Files

:R4 handles external memory with the following words :

DIR	"" -	Change current directory
FILE	nr -- ""	Get name from number
LOAD	's "" -- 'h	Load a file in memory

SAVE | 's c "" –

Write a memory chunk

Extending the vocabulary

The base words dictionary is built in the virtual machine that executes the language, specific domain vocabularies can be created to ease programming, the language is precisely in this phase.

Install procedure ?

No installation is needed for :R4, one only needs to extract the downloaded ZIP archive (in any place) and run r4.exe

How does :R4 run ?

When r4.exe is called, the main.txt file is loaded and executed, please, modify this file to add, delete or modify programs, Don't be afraid to break anything, just keep a copy of the archive if you want to restore everything to its original state.

Notice that from a given program, one can call another one by using:

```
"other.txt" RUN
```

notice that other.txt is the name of the source code archive you want to call, the new program will start running or an error message will be printed along with the corresponding error line. The code of each program is the text file as extracted from the archive..just open and edit it with NOTEPAD for example.

Final words

The author's wish is to meet with people who use, improve and increase the number of programs for :R4.

Don't despair if at the beginning you don't know how to start a program, this is normal, as much in FORTH as in :R4 the terseness of the language implies thinking about all the details.

If a group of words repeats a lot, create another word and see how code is reduced, sometimes some words disappear, get used to deleting as much code as you write.

Use only integers, if you need fractions multiply the unit to divide it into parts, don't use floating point, as you would be adding uncertainty to your numbers representation.

Try to always reduce source code, less words is always easier to understand than many words, and it will of course be faster as a result.

As someone said, simplicity is the final goal, not the starting point.

Programming in :R4

Not so long ago, 64 kb was a lot of memory, programs were published in magazines, one could key them in and they were kept in tapes, the latter was not exactly fun but the former was, given that one would learn as the code was keyed in.

Have a look at the following example, the numbers are used to reference lines and should not be copied.

```
01]^reda4.txt
02]^gui.txt
03]:start
04]   'exit >esc<
05]   show cls
06]           16 16 screen blanco
07]           "Hello world" print ;
08]
09]: start ;
```

Line 1 and 2 is used to include words for graphics, animation, keyboard and fonts, this line is almost always present at the top of an :R4 source code file.

Line 3 defines the start word that is called at the beginning of the program at line 9.

Line 4 associates the exit action to the ESC key.

Line 5 uses SHOW to introduce the words that will be used to draw the screen until the ; (semi colon) of line 7, the first word 'CLS' clears the screen.

Line 6 defines the size of the letters that will be used, screen specifies the numbers of columns and lines that will be used on the screen, then 'blanco' is used to specify that any subsequent drawing will be white ('blanco' in Spanish)

Line 7 prints on the screen the text in-between quotes.

Animation example in :R4

Let's see an example of an animation that draws a bouncing ball.

```

01]^reda4.txt
02]^gui.txt
03]#ball $cc004 $7493FE85 $6BCE50D7 $22DC7 $93D250D7
04]$8B73FE87 $9435B557 $1CF37 $6C31B3C7 $7493FE87 0
05]#xb 0 #yb 0 #vx 8 #vy 0 #ay 2
06]:toc          vx neg 'vx ! ;
07]:tic          vy neg 'vy ! ;
08]:screen
09]  'exit >esc<  show cls
10]  100 100 dim xb yb pos 'ball sprite
11]  xb vx + sw >? ( toc ) 0 <? ( toc ) 'xb !
12]  ay 'vy +!
13]  yb vy + sh >? ( tic drop sh ) 'yb ! ;
14]
15]: screen ;

```

Line 1 and 2 includes extensions.

Lines 3 and 4 define space for eleven 32 bits numbers, these numbers represent the drawing of the ball, this drawing can be created with the GARABATOR program and then the yourname.inc file in the r4inc folder that this program generates can be used.

Line 5 defines the variables that will be used, notice that XB YB are coordinates that are used at line 10 to specify the position of the drawing (sprite in the parlance of old micro computers). VX VY represent the speed of each coordinate and AY is vertical acceleration, something like gravity.

Line 6 defines TOC as sign change for VX (when it bounces against the borders)

Similarly, line 7 defines TIC as the sign change for VY (when it bounces against the "floor")

Notice here how line 15 is the start of the program as indicated with a ':' , a space and then the word defined between lines 8 and 13.

Line 9 associates pressing the ESC key to the action of exiting the screen drawing.

Assigning actions to keys is direct, there are no faster and simpler actions than assigning an action to each external event

Line 9 uses SHOW to specify the start of the screen drawing words, this drawing is performed 30 times per second by repeating the words in between SHOW and the end of the definition, CLS erases the screen, just try removing CLS and see

what happens.

Line 10 first defines the screen size, change the 100 value before the DIM (that specify dimensions) and see what happens, then the position is specified by variable XB and YB as already seen, before POS (for position) and then the SPRITE drawing defined at line 3 is drawn.

Line 11 calculates horizontal movement, studied step by step this is what happens:

```
xb   | push value of XB on the stack
vx   | push VX on the stack
+    | add the two
sw   | push the screen width on the stack
>?   | is the calculated sum bigger than the width ?
(    | if yes then do what's in between ( )
toc  | bounce ( defined above)
)    |
0    | push 0 on the stack
<?   | Is the pushed value below 0 ?
(    | If yes then do what's in between ( )
toc  | bounce
)    |
'xb  | Push address of XB
!    | Keep in XB the sum produced at the start
```

Line 12 sums up AY in VY

Line 13 is like line 12 but only tests for the floor, notice here that the sum is replaced when it is bigger than the floor.

To test:

Add the line that follows to the definition of the toc word :

Rand 4 << 4 or 'ball !

What happens to the ball ? What if I want the ball to do the same when it falls ?

Solutions

Exercice 1

2 3 over over over over

2 dup dup dup dup dup

1 2 3 4 5 6

¿?

Exercise 2

```
:**2 dup * ;  
:sum3 + + ;  
:2drop drop drop ;  
:2dup over over ;  
:inv4 1 2 3 4 swap 2swap swap ;  
:3dup dup 2over rot ;  
:-rot rot rot ;  
:resa **2 swap **2 + ;  
:resb over + * + ;  
:resc      2dup      -      -rot      +      /      ;
```

Reference base :R4

Prefixes					
^modulo.txt			include modulo.txt file		
:new			define new as the list until the ;		
::newg			define and export this definition		
#memory			define memory as a variable		
#:memoryg			define and export the variable		
'hello			push the address of hello (previously defined)		
blabla...			comment, ignore until end of line		
"bla bla"			text (include spaces), push address		
Control					
(...)			Repeat WORDS in between parentheses		
?? (.v.)(.f.)			Condition		
(.f. ??)			Repeat until condition (run at least once)		
(.. ??)(.v.)			Repeat while condition		
[...]			Anonymous definition(push its address)		
EXEC	d --		Call the address on the stack		
Conditionals					
0?	--	1?	--	Is the top of the stack null/(not null) ?	
+?	--	-?	--	Is the top of the stack positive/negative ?	
=?	ab - a	<?>	ab - a	a = b ? a <> b ?	
>?	ab - a	<?	ab - a	a > b ? a < b ?	
<=?	ab - a	>=?	ab - a	a <= b ? a >= b ?	
and?	ab - a	nand?	ab -- a	Test bits	
Data stack					
DUP	a -- aa	ROT	abc -- bca		
DROP	a --	2DUP	ab -- abab		
OVER	ab -- aba	2DROP	ab --		
PICK2	abc -- abca	3DROP	abc --		
PICK3	abcd -- abcda	4DROP	abcd --		
PICK4	abcde - abcdea	2OVER	abcd -- abcdeb		
SWAP	ab -- ba	2SWAP	abcd -- cdab		
NIP	ab - b				
Logic					
AND	ab -- c	c = a AND b	XOR	ab -- c	c = a XOR b
OR	ab -- c	c = a OR b	NOT	a -- b	b = NOT a
Arithmetic					
+	ab -- c	c=a+b	NEG	a -- b	b= -a
-	ab -- c	c=a-b	ABS	a -- b	b= a
*	ab -- c	c=a*b	1+	a -- b	b=a+1
/	ab -- c	c=a/b	1-	a -- b	b=a-1
*/	abc -- d	d=a*b/c	2/	a -- b	b=a/2
/MOD	ab -- c d	c=a/b	2*	a -- b	b=a*2
MOD	ab - d	d=a modulus b	<<	ab -- c	c=a<<b
*>>	abc - d	d=(a*b)>>c	>>	ab -- c	b=a>>b

Memory					
@	a - b	b=32(a)	@+	d - d+4 v	
C@	a -- b	b=8 (a)	C@+	d -- d+1 byte(v)	
W@	a -- b	b=16(a)	W@+	d -- d+2 word(v)	
!	vd --	32(d) =v	!+	vd -- d+4	
C!	vd --	8(d) =v	C!+	vd -- d+1	
W!	vd --	16(d)=v	W!+	vd -- d+2	
+	vd --	32(d)=32(d) + v	W+!	vd -- 16(d)=16(d) + v	
C+!	vd --	8(d)=8(d) + v	MEM	-- d free memory	
Return stack					
>R	a --	R: --a	R<	-- a	R:a --
R	-- a	R:a--a	R+	v --	R:a - a+v
R@+	-- v	R:a--a+4	R!+	v --	R:a -- a+4
			RDROP	--	R:a --
System					
MSEC	-- a	Milliseconds of the system			
TIME	-- h m s	Hour, minutes and seconds			
DATE	-- d m a	Day, month and year			
RESET	--	Exit from :r4, finalize and shutdown virtual machine			
RUN	d --	Load, compile and execute the file which name is in d			
Screen					
SW	-- w	Push screen width on the stack			
SH	-- h	Push screen height on the stack			
CLS	--	Erase screen			
REDRAW	--	Draw real screen with virtual one			
FRAMEV	-- a	Push start of virtual screen on the stack			
UPDATE	--	Update internal SO events			
Drawing					
OP	xy --	Point of origin			
CP	xy --	Control point for the curve			
LINE	xy --	Draw line			
CURVE	xy --	Draw curve			
PLINE	xy --	Draw polygon line			
PCURVE	xy --	Draw polygon curve			
POLI	--	Draw polygon			
Color					
PAPER	a --	Set background color			
INK	a --	Set current drawing color			
INK@	-- a	Push current drawing color on the stack			
ALPHA	a --	Set alpha (0-255)			
External storage					
DIR	"" --	Change current directory			
FILE	n -- ""	Get the name given the number			
LOAD	a"" -- b	Load a file in memory			
SAVE	ac"" --	Write a memory chunk			