R3forth Programming Manual

A Concatenative Language Derived from ColorForth

Pablo H. Reda - 2025

English Translation and Corrections

Repository: https://github.com/phreda4/r3

Table of Contents

Programming a Computer

A computer is a mechanism that needs a program to function - that is, a description of what it has to do.

This description is in a language the machine understands. But the programming process begins earlier.

First comes an idea, for example, drawing a circle. Then we write code so the machine draws a circle with radius 1 in a specific language, for example:

```
:draw 1 circle ;
: draw ;
```

The computer first translates this code into another language called machine code, which is the only one it can really understand. This process is called **compilation**.

Once the code written in the program is compiled, the result is executed on the computer.

If the code we write cannot be compiled, then there's an error in the code.

```
:draw 1 cicle ;  | Error: 'cicle' not found
: draw ;
```

When we try to compile, the computer tells us there's an error and doesn't execute anything.

Another possibility is that the program is well written but doesn't do what we want.

To accomplish this we need:

1. Have a problem or task to solve
2. Have an idea of how to solve it
3. Translate the idea into the programming language, make the program or code
4. Compile the code without errors
5. Execute the code and have it do what we imagine

> **Important:** We can only program what we know how to solve or test some solution we invent. We cannot program what we don't know how it works.

R3 Language

Introduction

Programming a computer means building a mechanism that will produce behavior in the machine, making the "recipe" of this behavior. This RECIPE is called SOURCE CODE or PROGRAM.

The program consists of 2 types of definitions:

- **DATA**, which we can also call MEMORY, STATE, or VARIABLE
- **CODE**, which we also call ORDER, ROUTINE, FUNCTION, or ACTION

As DATA we need to represent places in memory where to store numbers, which can represent or be used as:

- **QUANTITY**, for example: 3 lives
- **ADDRESS or LOCATION**, for example: position 100 on the screen
- **STATE**, for example: jumping or falling

As CODE we need to build ACTIONS.

We can build any behavior with the following elements:

- **SEQUENCE**: one order follows the next
- **CONDITION**: only if a condition is met is an order followed
- **REPETITION**: repeat an order in some indicated way
- **RECURSION**: define an action by referring to the same action (least used)

Programming Concepts in R3

The program is a text file, the SOURCE CODE, where it's separated into words. A WORD is defined as a sequence of letters, digits, or characters, separated by spaces. Valid words are:

```
LIVES 134 -*jump*- lives
```

Case is not considered, so LIVES and lives are the same for the language.

> **Important:** Spaces define word boundaries. Often a code error is missing a space. The form of separation doesn't matter - it's the same if there's one space, several spaces, or it's on the next line.

The source code is read word by word:

- If the word is a number, it's pushed onto the DATA STACK, and moves to the next word
- If the word is not a valid number, it's searched in the DICTIONARY, if found it's EXECUTED and moves to the next word
- If the word has any of the valid prefixes, it's interpreted according to its meaning and moves to the next word
- If the word is not found, the program terminates indicating an error at that location

Prefix System

8 prefixes are recognized in words. These have meaning in the code we write:

Prefix	Meaning	Example	Description
`|`	Comment	`| This is a comment`	Not executed, ends at line end
`^`	Include	`^r3/lib/console.r3`	Include code from indicated file
`"`	String	`"Hello"`	Define text string, ends with `"`
`:`	Action	`:myword`	Define actions
`#`	Data	`#variable 5`	Define data/variables
`$`	Hexadecimal	`$FF`	Hexadecimal numbers
`%`	Binary	`%1010`	Binary numbers
`'`	Address	`'word`	Address of a word

> **Note:** The address prefix needs a valid word, otherwise it's an error. Base dictionary words don't have addresses.

Dictionary System

The language begins with a predefined dictionary. This dictionary has around 200 words that are the basic functions the computer performs. (See Appendix 1)

From here, new words are defined with prefixes `:` and `#`, and added to the dictionary to be used later.

When the language searches for a word in the dictionary, this search is performed from last to first. Words with the same name can be defined, but only the last defined will be executed.

> **Programming is creating words:** Programming is defining new words in this dictionary and finally calling the first word that will execute the entire program.

The inclusion prefix `^` takes the indicated text file and adds to the dictionary all words defined in this text that are marked as exported: double `::` for code and

double `##` for data.

Example Program

```
#side 5
:square dup * ;
: side square ;
```

Line 1 defines a variable with value 5.
Line 2 defines a word that duplicates the top of stack and multiplies these two numbers.
Line 3 is the program start, pushes 5 (the value of variable 'side'), then calls square.

At the end of the program, the stack will have the number 25.

> **Important:** The semicolon doesn't indicate the word definition is finished, but that execution terminates. A word can have multiple end points and even no end, continuing in the next defined word.

Data Stack

The stack is a concept in computing to represent values within a determined order and functioning. The first number to enter the stack is the last to leave it (LIFO - Last In, First Out).

When numbers are found in source code, they will be stacked as the code executes.

This is where all operations are performed. Here values are stored that will be used for calculations and to indicate parameters to other words.

> **Stack Notation:** Each word can take and/or leave values on the data stack. As help to the programmer, a comment is put with a stack state diagram before and after the word, separated by two dashes.

Stack Manipulation Words

Word	Stack Effect	Description
`DUP`	`a -- a a`	Duplicate top of stack
`SWAP`	`a b -- b a`	Exchange top two items
`DROP`	`a --`	Remove top of stack
`ROT`	`a b c -- b c a`	Rotate three items
`OVER`	`a b -- a b a`	Copy second to top
`NIP`	`a b -- b`	Remove second item

Some words PRODUCE numbers (like DUP duplicating the top of stack), others CONSUME numbers (like DROP eliminating the top), and some don't affect the stack or both consume and produce.

> **Stack Balance:** If your definition makes the stack grow indefinitely or consumes all numbers, the program has an error. This cannot be detected before execution - the program will freeze or stop.

Arithmetic Operations

Addition and Subtraction

Word	Stack Effect	Description
`+`	`a b -- c`	c = a + b
`-`	`a b -- c`	c = a - b

Multiplication and Division

Word	Stack Effect	Description

Word	Stack Effect	Description
`*`	`a b -- c`	c = a * b
`/`	`a b -- c`	c = a / b
`MOD`	`a b -- c`	c = a mod b
`/MOD`	`a b -- c d`	c = a/b, d = a mod b

Bit Shifting

Word	Stack Effect	Description
`<<`	`a b -- c`	Left bit shift
`>>`	`a b -- c`	Right bit shift
`>>>`	`a b -- c`	Right bit shift without sign

Examples:

```
5 2 <<    | pushes 20
5 1 >>    | pushes 2
-2 1 >>   | pushes -1
-1 1 >>>  | pushes 9223372036854775807
```

Other Operations

Word	Stack Effect	Description
`NEG`	`a -- -a`	Negate value
`ABS`	`a -- |a|`	Absolute value
`SQRT`	`a -- b`	Square root
`*/`	`a b c -- d`	d = a*b/c without bit loss

Logical Operations

Word	Stack Effect	Example
`AND`	`a b -- c`	`$ff $55 AND` → $55
`OR`	`a b -- c`	`$2 $1 OR` → 3
`XOR`	`a b -- c`	`$3 2 XOR` → 1
`NOT`	`a -- b`	`0 NOT` → -1

Fixed Point Operations

To operate with fixed point, the language recognizes numbers with decimal points and translates them to that representation in 48.16 format.

With these numbers, addition and subtraction are the same as integers, but multiplication and division need special words defined in `r3/lib/math.r3`:

```
^r3/lib/math.r3

| Fixed point operations
*.     | f f -- f  multiply two fixed point numbers
/.     | f f -- f  divide two fixed point numbers
int.   | f -- a    convert to integer

| Trigonometric functions (angle in turns: 180 degrees = 0.5)
cos    | f -- cos(f)
sin    | f -- sin(f)
tan    | f -- tan(f)
sqrt.  | f -- sqrt(f)
ln.    | x -- r
exp.   | x -- r
root.  | base root -- r
```

> **Fixed Point Format:** R3 uses 48.16 fixed point format - 48 bits for integer part, 16 bits for fractional part. This provides good precision while avoiding floating point complexity.

The language also provides special operations for any bit distribution:

Word	Stack Effect	Description
`*>>`	`a b c -- d`	d = (a*b)>>c without bit loss
`<</`	`a b c -- d`	d = (a<<c)/b without bit loss

Conditionals

Parentheses are used to indicate code blocks. Note that they are words too and must be separated by spaces:

```
( code block )
```

Along with code blocks, we have conditional words that make comparisons to build conditionals and repetitions.

Stack-Only Conditionals

These conditionals only check the top of stack without modifying it:

Word	Stack Effect	Description
`0?`	`a -- a`	True when a = 0
`1?`	`a -- a`	True when a ≠ 0
`-?`	`a -- a`	True when a < 0
`+?`	`a -- a`	True when a ≥ 0

Comparison Conditionals

These compare TOS (top of stack) with NOS (next of stack), consuming TOS:

Word	Stack Effect	Description
`=?`	`a b -- a`	a = b?
`<?`	`a b -- a`	a < b?

Word	Stack Effect	Description
<=?	a b -- a	a ≤ b?
>?	a b -- a	a > b?
>=?	a b -- a	a ≥ b?
<>?	a b -- a	a ≠ b?

Building Conditionals

A condition is built with a word indicating the condition followed by a code block that executes only if the condition is met:

```
3
4 >? ( "Greater than 4" .print )
4 <? ( "Less than 4" .print )
drop
```

Line 2 asks if there's a number greater than 4 on top of stack, and if so prints the text.
The second of stack is not consumed, so conditions can be chained for a determined value.

Early Exit Example

```
:min | a b -- c
    over >? ( drop ; ) nip ;
```

> **No IF-ELSE:** R3 doesn't have IF-ELSE construction. This absence encourages factorization - it forces you to separate logic or find another way to solve the algorithm.

To replicate IF-ELSE behavior:

```
| Instead of: A ?? ( B ) else ( C ) D
| Use:
:condition ?? ( B ; ) C ;
A condition D
```

Switch-Case Alternatives

For sequential integers, use jump tables:

```
:a0 "action 0" ;
:a1 "action 1" ;
:a2 "action 2" ;
:a3 "action 3" ;
:a4 "action 4" ;

#list 'a0 'a1 'a2 'a3 'a4

:action | n -- string
    3 <<        | 8 * (cell size)
    'list + @ ex ;
```

For non-sequential values, use comparison chains:

```
:cases | value -- value string
    5 <? ( "less than 5" ; )
    6 =? ( "is 6" ; )
    7 =? ( "is 7" ; )
    111 <? ( "between 8 and 110" ; )
    "greater or equal to 111" ;
```

> **Stack Balance:** Ensure all branches produce the same stack behavior, unless intentionally different.

Repetition

When the conditional is inside the block, a repetition is constructed. While this condition is met, the block repeats. When false, execution jumps to the next word after the block.

```
( condition code )
```

Counting Up

```
1 ( 10 <?
    dup "%d " .print
    1 + ) drop
```

This code prints numbers 1 to 9. When TOS becomes 10 on line 2, it jumps to line 3 after the closing parenthesis.

Nested Loops - 10x10 Table

```
#table * 800  | 10 x 10 x 8 (8 bytes per number)

'table
0 ( 10 <? 1 +
    0 ( 10 <? 1 +
        rot @+ "%d " .print
        rot rot
    ) drop
    .cr
) drop
```

Preferred Pattern - Countdown

Conditionals that don't consume stack execute faster. Zero becomes the preferred end marker:

```
10 ( 1? 1 - ) drop
```

Whether we need 0 in the loop determines if we put our code before or after the decrement.

Memory Traversal

When traversing memory, it's practical to have a termination marker, usually 0:

```
"hello" ( c@+ 1?
    use_each_character ) 2drop
```

> **Careful:** When exiting the loop, besides the address, we have 0 pushed on stack.

Using a count would require:

```
"hello" 4 ( 1? 1 - swap c@+
    use_each_character swap ) 2drop
```

Multiple Exit Conditions

Valid to perform multiple comparisons to exit the loop:

```
( c@+ 1?
    13 <>?
    drop )  | here we have 0 or 13
```

Each loop exit must have the same stack behavior.

Recursion

Recursion is built by calling the word being defined from within itself.

Fibonacci Example: The Fibonacci sequence starts with 1 at positions 0 and 1, with each subsequent term being the sum of the two previous ones.

```
:fibonacci | n -- f
    2 <? ( 1 nip ; )
    1 - dup
    1 - fibonacci swap fibonacci + ;
```

As soon as a word definition starts, it can be invoked, so recursion occurs naturally.

> **Recursion Guidelines:** As with all recursive definitions, special care must be taken with the termination condition and stack state when the word terminates.

Tail Call Optimization

When a word is called before `;` (end of word), the language internally doesn't return from this word to terminate, but terminates in the called word. This is called "tail call" optimization.

In recursion, if the last word calls itself, it transforms into a loop:

```
:loopback | n -- 0
    0? ( ; )
    1 -
    loopback ;
```

Variables and Memory

Variables define memory for data storage. Variables have a name, a memory address, and a value stored at that memory location. The actual address where each variable is located is obtained when executed - it's not necessary to know this address value, just use its name to represent it.

```
#lives 3
#positionX #positionY
#map * $400  | 1KB
#list 3 1 4
#energy 1000
```

Using hex address $1000 as an example, this code reflects in memory as:

Line	Name	Address	Value
1	lives	$1000 (4096)	3
2	positionx	$1008 (4104)	0
2	positiony	$1010 (4112)	0
3	map	$1018 (4120)	0 0 0 ... 0
4	list	$1418 (5144)	3 1 4
5	energy	$1430 (5168)	1000

Memory Access Words

Word	Stack Effect	Description
`!`	`value address --`	STORE - write value to address
`@`	`address -- value`	FETCH - read value from address
`+!`	`val address --`	Add val to value at address

Word	Stack Effect	Description
~~!+~~	~~address value -- address+8~~	~~Store and increment address~~
`@+`	`address -- address+8 value`	Fetch and increment address

> **Memory Layout:** Each number in memory is stored in 8 bytes (64 bits). If there's a sequence of numbers, they will be at this distance apart.

Example Usage

```
:listshow
    'list
    @+ "%d " .print  | prints 3
    @+ "%d " .print  | prints 1
    drop ;

'list 8 + @         | pushes 1
1 'positionX +!     | add 1 to positionX
listshow            | prints 3 1

5 6                 | 5 6
'list               | 5 6 $1418
!+                  | 5 $1420
!                   |
listshow            | prints 6 5
```

Different Memory Access Sizes

Size	Fetch	Store	Incr	Description
8 bits	`c@`	`c!`	`c@+` `c!+`	byte/character
16 bits	`w@`	`w!`	`w@+` `w!+`	word
32 bits	`d@`	`d!`	`d@+` `d!+`	dword
64 bits	`@`	`!`	`@+` `!+`	qword (default)

Memory Buffer Management

```
| Memory buffer with pointer
#buffer> 'buffer

:+element | element --
    buffer> !+ 'buffer> ! ;

:traverse
    'buffer ( buffer> <?
        @+ "%d " .println
    ) drop ;

3 +element
4 +element
5 +element
traverse | prints 3 4 5
```

Text and Memory

Double quotes handle text within the program. Text is identified by this prefix and terminated with another double quote.

Text in source code is called a string or character chain.

Text can be thought of as a sequence of bytes in memory - text handling is the same as memory handling, but the unit is bytes instead of qwords (8 bytes).

String Examples

```
"example text"
" example text "  | with spaces
"Say ""HELLO"" to everyone"  | embedded quotes
```

When R3 encounters text, it stores it in memory. Each character occupies one byte corresponding to ASCII code. A zero byte is added at the end indicating the text's end. The ADDRESS of the text start is then pushed onto the stack.

> **String Storage:** Text is stored in separate memory from variables, unless the text is in a variable definition, in which case it will be in the variables memory area.

```
#lives 3
#text "BEWARE of the dog"
#dogs 4
```

Character-by-Character Processing

```
:printascii | t --
    ( c@+ 1? "%d " .print ) 2drop ;

"AB" printascii  | prints: 65 66
```

Formatted Output with sprint

The `sprint` word (in r3/lib/mem.r3) processes text with % placeholders:

Format	Description
`%d`	Print number in decimal
`%b`	Print number in binary
`%h`	Print number in hexadecimal
`%s`	Print text from address
`%%`	Print % sign

```
253 254 255 "%d %b %h" .print
| prints: 255 11111110 fc
```

> **Careful:** The text unpushes requested values, sometimes we can forget an over or dup to maintain these numbers on the stack.

String Processing Libraries

Useful definitions for text handling are in libraries r3/lib/str.r3 and r3/lib/parse.r3.

For example, to count characters in text:

```
::count | s1 -- s1 cnt
    0 over ( c@+ 1?
        drop swap 1 + swap ) 2drop ;
```

This traverses until finding a 0 byte, adding to a counter on stack. When finished, removes the last 0 and address, leaving the count on stack.

Registers A and B

We have something like two privileged variables called registers A and B. These registers are very useful for traversing memory, for reading or writing values. Additionally, they are maintained between word calls.

Word	Stack Effect	Description

Word	Stack Effect	Description
`>A`	`a --`	Load register A
`A>`	`-- a`	Push register A value
`A+`	`a --`	Add to register A
`A@`	`-- a`	Fetch from memory at A
`A!`	`a --`	Store in memory at A
`A@+`	`-- a`	Fetch from A and increment A
`A!+`	`a --`	Store at A and increment A

> **Register Usage:** When using registers, be careful that whoever calls these words, if they also use them, will be modified. It's necessary to save them somehow to prevent this.

Example: Finding Minimum Values

```
#list1 * $ffff  | 64kb
#list2 * $ffff  | 64kb
#minimum

:search |
    a@+ minimum <? ( 'minimum ! ; ) drop ;

'list1 >a
a@ 'minimum !
1000 ( 1? 1- search ) drop

'list2 >a
a@ 'minimum !
1000 ( 1? 1- search ) drop
```

The word `search` will use the address in register A. On line 8 it will be list1, on line 12 it will be list2. This could be passed as a parameter on the stack but would require moving the element counter. The variable minimum will have the minimum value of each list when the loop terminates.

Nested Loops with Registers

The nested loop example with registers avoids the stack manipulation needed to retrieve address, recover value, and advance:

```
#table * 800  | 10 x 10 x 8 (8 bytes per number)

'table >a
0 ( 10 <? 1 +
    0 ( 10 <? 1 +
        a@+ "%d " .print
    ) drop
    .cr
) drop
```

Return Stack

There exists a second stack that handles calls between words, storing the return address that will be used each time a word termination `;` is executed.

We have the following words to manipulate the return stack:

Word	Stack Effect	Description
`>R`	`a -- rstack: -- a`	Push to return stack
`R>`	`-- a rstack: a --`	Pop from return stack

Word	Stack Effect	Description
R@	r- a nstack: a -- a	Copy top of return stack

When a word is called (a code definition, not data), the language pushes onto the return stack the location where it should return once the called word's execution terminates.

> **Return Stack Usage:** It's advisable not to use this stack since an imbalance between calls will cause the code to break. However, it can be used as an alternative place to save or retrieve values, always being careful that each word that pushes values here must pop them precisely.

It's also possible to alter this stack to change execution flow, but again, you must think carefully about what will happen.

Operating System Connection

The computer only works with numbers, and any communication with the user or the rest of the world is done through words that the operating system resolves, independent of the language but dependent on its connection to it.

Access to external libraries is fundamental because it allows us to access hardware capabilities not directly available, either by manufacturer criteria that doesn't want its internal workings seen, or by resource complexity that doesn't want to be reprogrammed.

Generally, library documentation is needed to know what functions to import, and the use and functioning of these words will be the library's responsibility. Once we have access to the library, we can build our program from there.

Console Access Example

```
^r3/lib/console.r3

: "hello world" .println ;
```

The key word is `.println` whose function is to take the text address and print it to terminal and go to the next line. This word is built from OS calls and necessary processing to make it easier to use.

Advanced Operating System Connection

The OS connection is built through function calls to dynamic libraries. In Windows these are called .DLL (dynamic link libraries).

The current distribution, besides connecting to the OS, uses the hardware's graphics capabilities through SDL version 2 libraries, since it's multiplatform.

Windows DLL	R3 Library
SDL2.dll	`^r3/lib/sdl2.r3`
SDL2_image.dll	`^r3/lib/sdl2image.r3`
SDL2_mixer.dll	`^r3/lib/sdl2mixer.r3`
SDL2_net.dll	`^r3/lib/sdl2net.r3`
SDL2_ttf.dll	`^r3/lib/sdl2ttf.r3`

> **Platform Support:** R3 is prepared for use on Linux, MAC, or RPI, though these are not yet available.

Library Loading

The way to connect to a library is through 2 words: one to load the library and one to get the address of the function to execute, plus a set of words that make the call.

Word	Stack Effect	Description
`LOADLIB`	`"name" -- liba`	Load dynamic library
`GETPROC`	`liba "name" -- aa`	Get function address

Words to call these functions take parameters from the stack according to parameter count and leave the OS response:

Word	Stack Effect	Description
`SYS0`	`aa -- r`	Call function with 0 parameters
`SYS1`	`a aa -- r`	Call function with 1 parameter

Word	Stack Effect	Description
SYS2	a b -- r	Call function with 2 parameters
...	...	...
`SYS10`	`a b c d e f g h i j aa -- r`	Call function with 10 parameters

Libraries

Most common words from main libraries.

Operating System Communication

^lib/core.r3

Word	Stack Effect	Description
`::msec`	`-- msec`	Push system milliseconds since program start
`::time`	`-- hms`	Push current time (hours, minutes, seconds) in one value
`::date`	`-- ymd`	Push current date (year, month, day) in one value

Time Display Example

```
:.time | -- ; print time in hh:mm:ss format
    time
    dup 16 >> $ff and "%d:" .print  | hour
    dup 8 >> $ff and "%d:" .print   | minute
    $ff and "%d" .print            | second
    ;

:.date | -- ; print date in yyyy-mm-dd format
    date
    dup 16 >> $ffff and "%d-" .print  | year
    dup 8 >> $ff and "%d-" .print    | month
    $ff and "%d" .print              | day
    ;
```

Console Printing

The language starts in a window called console where only text can be written. The writing position is called cursor and moves as characters are placed. When it reaches the console end, it scrolls up to make room.

^lib/console.r3

Word	Stack Effect	Description
`::cls`	`--`	Clear console, erase all characters
`::.write`	`"text" --`	Write text to console
`::.print`	`.. "text with %" --`	Write formatted text, extract values from stack for %
`::.home`	`--`	Move cursor to console start
`::.at`	`x y --`	Position cursor at row y, column x
`::.fc`	`color --`	Set foreground color for text

Word	Stack Effect	Description
`::.bc`	`color --`	Set background color for text
`::.input`	`--`	Wait and save a line of text entered from keyboard
`::.inkey`	`-- key`	Return pressed key, zero if none

> **Input Buffer:** The `##pad` variable contains the entered text after `.input`.

Random Numbers

^lib/rand.r3

Word	Stack Effect	Description
`::rerand`	`s1 s2 --`	Initialize random generator with two seeds
`::rand`	`-- rand`	Push a 64-bit random number
`::randmax`	`max -- value`	Push random number between 0 and MAX-1

Random Number Usage

```
time msec rerand        | Initialize with variable time values
10.0 randmax            | Random number 0 to 9.999...
5.0 randmax 5.0 -       | Random number -5.0 to 0
5.0 randmax 5.0 +       | Random number 5.0 to 10.0
```

Graphics with SDL2 Library

The SDL2 library allows access to graphics capabilities. https://www.libsdl.org/

This library allows starting a graphics window and drawing on it. Besides the graphics window, mechanisms are needed to respond to KEYBOARD and MOUSE buttons.

Graphics Window - ^lib/sdl2.r3

Word	Stack Effect	Description
`::SDLinit`	`"title" w h --`	Start graphics window w×h pixels with title
`::SDLfull`	`--`	Set window to fullscreen
`::SDLquit`	`--`	Exit graphics window
`::SDLcls`	`color --`	Clear screen with chosen color
`::SDLredraw`	`--`	Refresh screen
`::SDLshow`	`'word --`	Execute WORD each time screen redraws
`::exit`	`--`	Exit from SHOW

Input Variables

Variable	Description
`##SDLkey`	Code of pressed key, zero if no key
`##SDLchar`	Character code representation
`##SDLx`, `##SDLy`	Mouse cursor position x and y in window
	Mouse button state, zero when none

##SDLb Variable	pressed Description

Loading Graphics Files - ^lib/sdl2image.r3

Word	Stack Effect	Description
`::loadimg`	`"file" -- img`	Load image file (PNG with transparency, JPG without)
`::unloadimg`	`img --`	Remove image from memory

Drawing Graphics - ^lib/sdl2gfx.r3

Word	Stack Effect	Description
`::SDLColor`	`col --`	Set drawing color
`::SDLPoint`	`x y --`	Draw a pixel
`::SDLLine`	`x1 y1 x2 y2 --`	Draw line from x1,y1 to x2,y2
`::SDLFRect`	`x y w h --`	Draw filled rectangle
`::SDLRect`	`x y w h --`	Draw rectangle outline
`::SDLFEllipse`	`rx ry x y --`	Draw filled ellipse
`::SDLEllipse`	`rx ry x y --`	Draw ellipse outline
`::SDLTriangle`	`x y x y x y --`	Draw filled triangle

Image Drawing

Word	Stack Effect	Description
`::SDLimagewh`	`img -- w h`	Get image width and height
`::SDLImage`	`x y img --`	Draw image at position
`::SDLImages`	`x y w h img --`	Draw image at position with size
`::SDLImageb`	`box img --`	Draw image in defined box
`::SDLImagebb`	`box box img --`	Draw part of image in defined box
`::SDLspriteZ`	`x y zoom img --`	Draw image at position with scale
`::SDLSpriteR`	`x y ang img --`	Draw image at position with rotation
`::SDLspriteRZ`	`x y ang zoom img --`	Draw image with rotation and scale

Tile Sheets

Word	Stack Effect	Description
`::tsload`	`w h filename -- ts`	Load image as tile sheet
`::tscolor`	`rrggbb 'ts --`	Set tile color (tints white color)
`::tsdraw`	`n 'ts x y --`	Draw tile on screen

Word	Stack Effect	Description
`::tsdraws`	`n 'ts x y w h --`	Draw tile on screen with size

Sprite Sheets

Sprites are parts of an image, drawn from the center of defined size.

Word	Stack Effect	Description
`::ssload`	`w h file -- ss`	Load sprite sheet
`::ssprite`	`x y n ss --`	Draw sprite N at centered position
`::sspriter`	`x y ang n ss --`	Draw sprite N centered with rotation
`::sspritez`	`x y zoom n ss --`	Draw sprite N centered with scale
`::sspriterz`	`x y ang zoom n ss --`	Draw sprite N centered with rotation and scale

Complete Example: Simple Game

```
^r3/lib/sdl2gfx.r3
^r3/lib/rand.r3

#sprites
#x 320.0 #y 240.0
#vx 0.0 #vy 0.0

:player
    x int. y int. 2.0 0 sprites sspritez
    vx 'x +! vy 'y +! ;

:game-loop
    0 SDLcls                    | Clear screen
    player                      | Draw and update player
    SDLredraw                   | Update display

    SDLkey
    >esc< =? ( exit )            | Exit on Escape
    <le> =? ( -2.0 'vx ! )       | Left arrow
    <ri> =? ( 2.0 'vx ! )        | Right arrow
    <up> =? ( -2.0 'vy ! )       | Up arrow
    <dn> =? ( 2.0 'vy ! )        | Down arrow
    >le< =? ( 0.0 'vx ! )        | Stop on key release
    >ri< =? ( 0.0 'vx ! )
    >up< =? ( 0.0 'vy ! )
    >dn< =? ( 0.0 'vy ! )
    drop ;

:main
    "R3forth Game Demo" 640 480 SDLinit
    time msec rerand
    16 16 "player.png" ssload 'sprites !
    'game-loop SDLshow
    SDLquit ;

: main ;
```

Appendix 1 - Base Dictionary

Essential words in the R3 base dictionary:

Control Flow

Word	Stack Effect	Description
`;`	`--`	End word execution
`(`	`--`	Begin code block
`)`	`--`	End code block
`[`	`-- vec`	Begin anonymous definition
`]`	`vec --`	End anonymous definition
`EX`	`vec --`	Execute word by address

Stack Conditionals

Word	Stack Effect	Description
`0?`	`a -- a`	True if TOS = 0
`1?`	`a -- a`	True if TOS ≠ 0
`+?`	`a -- a`	True if TOS ≥ 0
`-?`	`a -- a`	True if TOS < 0
`<?`	`a b -- a`	True if a < b (removes TOS)
`>?`	`a b -- a`	True if a > b (removes TOS)
`=?`	`a b -- a`	True if a = b (removes TOS)
`>=?`	`a b -- a`	True if a ≥ b (removes TOS)
`<=?`	`a b -- a`	True if a ≤ b (removes TOS)
`<>?`	`a b -- a`	True if a ≠ b (removes TOS)
`AND?`	`a b -- c`	True if a AND b (removes TOS)
`NAND?`	`a b -- c`	True if a NAND b (removes TOS)
`IN?`	`a b c -- a`	True if a≤b≤c (removes TOS and NOS)

Stack Operations

Word	Stack Effect	Description
`DUP`	`a -- a a`	Duplicate TOS
`DROP`	`a --`	Remove TOS
`OVER`	`a b -- a b a`	Duplicate NOS
`PICK2`	`a b c -- a b c a`	Duplicate 3rd element
`PICK3`	`a b c d -- a b c d a`	Duplicate 4th element

Word	Stack Effect	Description
`PICK4`	`a b c d e -- a b c d e a`	Duplicate 5th element
`SWAP`	`a b -- b a`	Exchange TOS and NOS
`NIP`	`a b -- b`	Remove NOS
`ROT`	`a b c -- b c a`	Rotate 3 elements
`-ROT`	`a b c -- c a b`	Rotate 3 elements
`2DUP`	`a b -- a b a b`	Duplicate 2 values
`2DROP`	`a b --`	Remove 2 elements
`3DROP`	`a b c --`	Remove 3 elements
`4DROP`	`a b c d --`	Remove 4 elements
`2OVER`	`a b c d -- a b c d a b`	Duplicate 2 elements from 3rd position
`2SWAP`	`a b c d -- c d a b`	Exchange 4 elements

Return Stack

Word	Stack Effect	Description
`>R`	`a -- rstack: -- a`	Push to return stack
`R>`	`-- a rstack: a --`	Pop from return stack
`R@`	`-- a rstack: a -- a`	Read top of return stack

Arithmetic Operations

Word	Stack Effect	Description
`+`	`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`	a shift left b
`>>`	`a b -- c`	a shift right b
`>>>`	`a b -- c`	a shift right b unsigned
`MOD`	`a b -- c`	c = a mod b
`/MOD`	`a b -- c d`	c = a/b, d = a mod b
`*/`	`a b c -- d`	d = a*b/c without bit loss
`*>>`	`a b c -- d`	d = (a*b)>>c without bit loss
`<</`	`a b c -- d`	d = (a<<c)/b without bit loss
`NEG`	`a -- b`	b = -a

Word	Stack Effect	Description
ABS	a -- b	b = |a|
SQRT	a -- b	b = square root(a)
CLZ	a -- b	b = count leading zeros of a

Logical Operations

Word	Stack Effect	Description
AND	a b -- c	c = a AND b
NAND	a b -- c	c = a NAND b
OR	a b -- c	c = a OR b
XOR	a b -- c	c = a XOR b
NOT	a -- b	b = NOT a

Memory Operations

Word	Stack Effect	Description
@	a -- [a]	Fetch qword from address
C@	a -- byte[a]	Fetch byte from address
W@	a -- word[a]	Fetch word from address
D@	a -- dword[a]	Fetch dword from address
@+	a -- b [a]	Fetch qword and increment by 8
C@+	a -- b byte[a]	Fetch byte and increment by 1
W@+	a -- b word[a]	Fetch word and increment by 2
D@+	a -- b dword[a]	Fetch dword and increment by 4
!	a b --	Store A at address B
C!	a b --	Store byte A at address B
W!	a b --	Store word A at address B
D!	a b --	Store dword A at address B
!+	a b -- c	Store A at B and increment by 8
C!+	a b -- c	Store byte A at B and increment by 1
W!+	a b -- c	Store word A at B and increment by 2
D!+	a b -- c	Store dword A at B and increment by 4
+!	a b --	Add A to value at address B
C+!	a b --	Add A to byte at address B
W+!	a b --	Add A to word at address B

Word	Stack Effect	Description
`D+!`	`a b --`	Add A to dword at address B

Register Operations

Word	Stack Effect	Description
`>A`	`a --`	Load register A
`>B`	`a --`	Load register B
`B>`	`-- a`	Push register B
`A>`	`-- a`	Push register A
`A+`	`a --`	Add to A
`B+`	`a --`	Add to B
`A@`	`-- a`	Fetch from A
`B@`	`-- a`	Fetch from B
`cA@`	`-- a`	Fetch byte from A
`cB@`	`-- a`	Fetch byte from B
`dA@`	`-- a`	Fetch dword from A
`dB@`	`-- a`	Fetch dword from B
`A!`	`a --`	Store in memory at A
`B!`	`a --`	Store in memory at B
`cA!`	`a --`	Store in memory at A
`cB!`	`a --`	Store in memory at B
`dA!`	`a --`	Store in memory at A
`dB!`	`a --`	Store in memory at B
`A@+`	`-- a`	Fetch from A and increment by 8
`B@+`	`-- a`	Fetch from B and increment by 8
`cA@+`	`-- a`	Fetch from A and increment by 8
`cB@+`	`-- a`	Fetch from B and increment by 8
`dA@+`	`-- a`	Fetch from A and increment by 8
`dB@+`	`-- a`	Fetch from B and increment by 8
`A!+`	`a --`	Store at A and increment by 8
`B!+`	`a --`	Store at B and increment by 8
`cA!+`	`a --`	Store at A and increment by 8
`cB!+`	`a --`	Store at B and increment by 8

Word	Stack Effect	Description
`dA!+`	`a --`	Store at A and increment by 8
`dB!+`	`a --`	Store at B and increment by 8

Memory Block Operations

Word	Stack Effect	Description
`MOVE`	`d s c --`	Copy S to D, C qwords
`MOVE>`	`d s c --`	Copy S to D, C qwords in reverse
`FILL`	`d v c --`	Fill D, C qwords with V
`CMOVE`	`d s c --`	Copy S to D, C bytes
`CMOVE>`	`d s c --`	Copy S to D, C bytes in reverse
`CFILL`	`d v c --`	Fill D, C bytes with V
`DMOVE`	`d s c --`	Copy S to D, C dwords
`DMOVE>`	`d s c --`	Copy S to D, C dwords in reverse
`DFILL`	`d v c --`	Fill D, C dwords with V
`MEM`	`-- a`	Start of free memory

System Interface

Word	Stack Effect	Description
`LOADLIB`	`"name" -- liba`	Load dynamic library
`GETPROC`	`liba "name" -- aa`	Get function address
`SYS0`	`aa -- r`	Call function with 0 parameters
`SYS1`	`a aa -- r`	Call function with 1 parameter
`SYS2`	`a b aa -- r`	Call function with 2 parameters
`SYS3`	`a b c aa -- r`	Call function with 3 parameters
`SYS4`	`a b c d aa -- r`	Call function with 4 parameters
`SYS5`	`a b c d e aa -- r`	Call function with 5 parameters
`SYS6`	`a b c d e f aa -- r`	Call function with 6 parameters
`SYS7`	`a b c d e f g aa -- r`	Call function with 7 parameters
`SYS8`	`a b c d e f g h aa -- r`	Call function with 8 parameters
`SYS9`	`a b c d e f g h i aa -- r`	Call function with 9 parameters
`SYS10`	`a b c d e f g h i j aa -- r`	Call function with 10 parameters

Conclusion

This manual provides comprehensive coverage of R3forth programming concepts and syntax. R3forth offers a unique approach to concatenative programming with its explicit prefix system, 64-bit architecture, and modern library integration.

Key takeaways:

- **Stack-based programming:** All operations work through the data stack
- **Prefix system:** 8 prefixes define word behavior explicitly
- **No traditional control structures:** Uses conditional words with code blocks
- **Memory flexibility:** Access memory in 8, 16, 32, or 64-bit sizes
- **Modern integration:** SDL2 support for graphics and multimedia
- **Factorization encouraged:** Language design promotes clean code separation

The language excels at system programming, game development, and applications requiring direct hardware access while maintaining the elegance and simplicity characteristic of the Forth family.

For more examples and advanced usage, explore the `/demo` directory in the R3 repository and experiment with the various library combinations.

Repository: https://github.com/phreda4/r3
Original Documentation: Pablo H. Reda
English Translation: 2025