

Coding Style Guide

Table of Contents

1. Introduction	1
2. Line of code length	1
3. Indentation	2
4. Block layouts	2
5. Naming conventions	4
6. Casing	6
7. Macros (Red/System)	6
8. Function definitions	7
9. Function calls	9
10. Comments	10
11. String syntax	11
12. New line usage	11

1. Introduction

Red is a homoiconic language where code is represented as data. A consequence of that feature is that the language is almost totally free-form, in order to cope with all possible ways to format data, as well as being flexible enough for DSL-specific formatting needs. The following content is *just one of many, many ways* to format Red code.

This document describes the *official coding style used in the Red source code*, so respecting this coding style is a prerequisite of every pull request submitted to [red/red](#) Github repository.

As Red/System is a dialect of Red, it shares the same syntax and coding style rules. Specific Red/System rules are marked as such.

The purpose of the following rules is to maximise readability by keeping an optimal number of lines of code visible on-screen while minimizing the need for comments.

2. Line of code length

There is no strictly-defined maximum number of columns for a single line of code, as it can vary according to the type of the font used (size, proportional vs fixed-width) or highlighting effects. It should be possible to read a full line of code (excluding comments) in an editor occupying at maximum, half of a 1080p monitor width. On the displays we use for Red codebase coding, it is about 100 columns. In the description below, *excessive size* or *too long* expressions will refer to lines of code that do not satisfy the aforementioned criteria.

3. Indentation

The Red codebase uses tabulations with size of 4 columns to indent the source code. This gives a good trade-off between too small values (like 2 columns) and too big values (like 8 columns). Using tabs also means that you can adjust it to your personal preference in your editor, while respecting the rule (just pay attention to right alignments using tabs then).

All the contributed Red files to `red/red` repo should contain the following field in their header:

```
Tabs: 4
```

Each time you go to a new line after opening a block or a parenthesis, you should indent by one tab.

Correct

```
func [  
    arg1  
    arg2  
    ...  
][  
    print arg1  
    ...  
]
```

Incorrect

```
func [  
arg1           ;-- missing indentation!  
arg2  
...  
][  
    print arg1   ;-- excessive indentation!  
    ...  
]
```

4. Block layouts

All the following rules apply to blocks `[]` as well as parenthesis `()`.

Empty blocks do not contain any whitespace:

```
a: []
```

Contiguous blocks do not require whitespace between the end of one and the start of another:

```
[][]  
[]()
```

```
[  
  hello  
][          ;-- no spacing required  
  world  
]
```

However, it is acceptable to use whitespaces in-between nested blocks:

```
array: [[] [] [] []]  
list:  [ [] [] [] [] ]  
  
either a = 1 ["hello"] ["world"]  
either a = 1 [ ["hello"] ] [ ["world"] ]
```

For expressions containing small blocks, they are usually opened and closed on the same line

```
b: either a = 1 [a + 1][3]
```

If the line is too long, the block should be wrapped over several lines with one level of indentation:

Correct

```
b: either a = 1 [  
  a + 1  
][  
  123 + length? mold a  
]
```

Incorrect

```
b: either a = 1  
  [a + 1][123 + length? mold a]
```

*That style is wrong because it breaks the ability to copy/paste code to the Red console (**either** will be evaluated before the block arguments are detected).*

If the first block is small enough and can fit on the same line, then only the subsequent blocks are wrapped over several lines:

```

print either a = 1 ["hello"][
  append mold a "this is a very long expression"
]

while [not tail? series][
  print series/1
  series: next series
]

```

5. Naming conventions

Variable names should be single-word **nouns**. Choose words that are short and capture the meaning as best as possible. Common words should be used first (*especially if they are already used in existing Red source code in the same context*). If needed, use a [synonyms dictionary](#) to find the best word for the usage. Single-letter or abbreviated words (unless the abbreviated word is in common usage) should be avoided as much as possible.

Names made of multiple words are separated with a dash - character. Use a two-words name only when a fitting single-word cannot be found or would be too confusing with already used ones. Word names with more than two hyphenated parts should only be used in rare cases. Consistent usage of single-word names reduces the horizontal code size and improves readability. Avoid useless verbosity.

Correct

```

code: 123456
name: "John"
table: [2 6 8 4 3]
lost-items: []

unless tail? list [author: select list index]

```

Incorrect

```

code_for_article: 123456
Mytable: [2 6 8 4 3]
lostItems: []

unless tail? list-of-books [author-property: select list-of-books selected-index]

```

When naming a function, strive to use single-word *verbs*, in order to express an action, though two or three words names are often necessary. More than three words should be avoided as much as possible. A noun or an adjective followed by a question mark is also accepted. Often, it denotes that the return value is of **logic!** type, but this is not a strict rule, as it is handy to form single-word action names for retrieving a property (e.g. **length?**, **index?**). When forming function names with

two or more words, always start with the verb. If your words are named carefully, the code becomes almost self-documented, reducing the need for comments.

Correct

```
make:  func [...  
reduce: func [...  
allow: func [...  
crunch: func [...
```

Incorrect

```
length:  func [...  
future:  func [...  
position: func [...  
blue-fill: func [...      ;-- should be fill-blue
```

There is an exception for described naming conventions, which applies only to OS imports and third-party libraries not written in Red: to make API-specific functions and structure fields easy to recognize, their original names should be used. This helps to visually distinguish imported names from regular Red or Red/System code. For example:

```

tagMSG: alias struct! [
  hWnd [handle!]
  msg [integer!]
  wParam [integer!]
  lParam [integer!]
  time [integer!]
  x [integer!]
  y [integer!]
]

#import [
  "User32.dll" stdcall [
    CreateWindowEx: "CreateWindowExW" [
      dwExStyle [integer!]
      lpClassName [c-string!]
      lpWindowName [c-string!]
      dwStyle [integer!]
      x [integer!]
      y [integer!]
      nWidth [integer!]
      nHeight [integer!]
      hWndParent [handle!]
      hMenu [handle!]
      hInstance [handle!]
      lParam [int-ptr!]
      return: [handle!]
    ]
  ]
]

```

6. Casing

All variable and function names should be lowercase by default, unless there is a good reason for using uppercasing such as:

- name is an acronym e.g. GMT (Greenwich Mean Time)
- name is operating-system or (non-Red) third-party API-related

7. Macros (Red/System)

Apply the same naming conventions when choosing Red/System macros names. Names of macros are generally written in uppercase, to visually distinguish them from the rest of the code (unless the intention is to make the macros look like regular code, like pseudo-datatype definitions). When multiple words are used, they are separated by an underscore `_` character to emphasize macro's differences from regular code.

(TBD: extract all single-word names used in the Red codebase as examples)

8. Function definitions

The general rule is to keep the spec block on a single line. The body block can be on the same line or over several lines. As Red/System spec blocks tend to be longer, most functions spec blocks are wrapped over several lines, so, for sake of visual consistency, small spec blocks are often wrapped as well.

Correct

```
do-nothing: func [][]
increment: func [n [integer!]][n + 1]

increment: func [n [integer!]][
  n + 1
]

increment: func [
  n [integer!]
][
  n + 1
]
```

Incorrect

```
do-nothing: func [
][
]

do-nothing: func [

][

]

increment: func [
  n [integer!]
][n + 1]
```

When the spec block is too long, it should be wrapped over several lines. When wrapping the spec block, each type definition must be on the same line as its argument. The optional attributes block should be on its own line. Each refinement starts on a new line. If followed by a single argument, the argument can be on the same line or a new line with an indentation (just be consistent with other refinements in the same spec block). For **/local** refinement, if the local words are not followed by type annotation, they can be put on the same line.

When wrapping the spec block over several lines, it is recommended to align the datatype definitions for consecutive arguments, on the same column for easier reading. Such alignment is

preferably done using tabs (if you strictly follow these coding style rules) or else, using spaces.

Correct

```
make-world: func [  
    earth    [word!]  
    wind     [bitset!]  
    fire     [binary!]  
    water    [string!]  
    /with  
        thunder [url!]  
    /only  
    /into  
        space   [block! none!]  
    /local  
    plants animals men women computers robots  
][  
    ...  
]
```

Incorrect

```
make-world: func [  
    [throw] earth [word!]      ;-- attributes block not on its own line  
    wind    [bitset!]  
    fire [binary!]            ;-- unaligned type definition  
    water   [string!]  
    /with  
        thunder [url!]  
    /only  
    /into space [block! none!] ;-- inconsistent with /with formatting  
    /local  
        plants animals      ;-- breaking line too early  
        men women computers robots  
][  
    ...  
]
```

The main docstring (describing the function) should be on its own line if the spec block is wrapped. The argument and refinement docstrings should be on the same line as the item they are describing. Docstrings start with a capital letter and do not require an ending dot (it's added automatically when printed on screen by `help` function).

Correct


```

increment: func ["Add 1 to the argument value" n][n + 1]

make-world: func [
  "Build a new World"
  earth  [word!]    "1st element"
  wind   [bitset!]  "2nd element"
  fire    [binary!]  "3rd element"
  water   [string!]
  /with    "Additional element"
    thunder [url!]
  /only     "Not implemented yet"
  /into      "Provides a container"
    space [unset!]  "The container"
  /local
    plants animals men women computers robots
][
  ...
]

```

Incorrect

```

make-world: func ["Build a new World" ;-- should be on a newline
  earth  [word!]    "1st element"
  wind   [bitset!]  "2nd element" ;-- excessive indentation
  fire    [binary!]
  "3rd element"      ;-- should be on same line as `fire`
  water   [string!]
  /with    "Additional element"
    thunder [url!]
  /only "Not implemented yet"    ;-- should be aligned with other docstrings
  /into
    "Provides a container"      ;-- should follow the refinement
    space [unset!]  "The container"
  /local
    plants animals men women computers robots
][
  ...
]

```

9. Function calls

Arguments are following the function call on the same line. If the line becomes too long, arguments can be wrapped over several lines (one argument per line) with an indentation.

Correct

```
foo arg1 arg2 arg3 arg4 arg5
```

```
process-many  
  argument1  
  argument2  
  argument3  
  argument4  
  argument5
```

Incorrect

```
foo arg1 arg2 arg3  
  arg4 arg5
```

```
foo  
  arg1 arg2 arg3  
  arg4 arg5
```

```
process-many  
  argument1  
    argument2  
      argument3  
        argument4  
          argument5
```

For long expressions with many nested parts, spotting the bounds of each expression can sometimes be difficult. Using parenthesis for grouping a nested call with its arguments is acceptable (but not mandatory).

```
head insert (copy/part [1 2 3 4] 2) (length? mold (2 + index? find "Hello" #"o"))
```

```
head insert  
  copy/part [1 2 3 4] 2  
  length? mold (2 + index? find "Hello" #"o")
```

10. Comments

In Red codebase:

- comments are written using the `;` prefix (stronger visual clue)
- single-line comments start at column 57 (works best on average, else column 53)
- multi-line comments are done using several single-line prefixes rather than `comment {…}` constructions.

The general rule is to put comments on the same line as the beginning of the corresponding code

instead of on a new line in order to save significant vertical space. Though, if the comment is used for separating chunks of code, then putting it on a new line is fine.

11. String syntax

Use `""` for single-line strings. The `{}` form is reserved for multi-line strings. Respecting this rule ensures:

- a more consistent source representation before and after LOADing code
- better convey of meaning

One exception to the rule is when a single-line string includes the `"` character itself. In this case, it is preferred to use the `{}` form rather than escaping the quote `^"` as it is more readable.

12. New line usage

TBD