PROGRAMMING WITH RED/SYSTEM

François Jouen
Human and Artificial Cognitions Laboratory (CHArt)
and
Réanimation Pédiatrique Raymond Poincaré (R2P2)
Ecole Pratique des Hautes Etudes
Université Paris Sciences et Lettres

Introduction

I spent a lot of time with Red/System when I was developing the interface for OpenCV, and then the redCV library. You'll find the original code here: https://github.com/ldci/OpenCV3-red and https://github.com/ldci/redCV. These are just some notes about my tests with Red/S and I hope they will be useful to other members of the Red community. The sample code is free and without guarantees. This document will be improved as the different tests are carried out. You can find a reference document for Red/System here: https://static.red-lang.org/red-system-specs.html. This page in maintained by Nenad Rakocevic and updated when there are important modifications in the development of Red language. Large parts of the present document come from Nenad's documentation (thanks).

Red/System is also used to create Red language. You'll can find in *red-master/runtime/datatypes* directory a lot of Red/System code involved in designing Red datatypes. It's very interesting to study this code to understand the way Red works. Most of the methods implemented in the various datatypes can be easily integrated in routines for a very fast compilation and execution.

What is Red/System?

Red/System is a *dialect* (or Domain Specific Language/DSL) of the Red programming language. Its purpose is to provide:

- low-level system programming capabilities
- a tool to build Red's runtime low-level library
- a tool to bind Red to external C libraries
- a tool to link code and produce fast executables

Red/System also assumes the function of an *intermediate representation language* (IRL) for compiled Red code. Red/System compiles directly into machine code for all supported targets by Red. Most parts of Red language are written with Red/System DSL

Red/System can be considered as a **C-level language** with memory pointer support and a *very basic and limited* set of datatypes.

Red/System datatypes

All the following datatypes are **first-class citizens** of Red/System language.

The **byte!** datatype's purpose is to represent *unsigned integers* in the 0-255 range. Byte! is basic, but very useful. I'm generally using this datatype for matrices and RGB image channels processing.

You can find here (http://www.rebol.com/docs/core23/rebolcore-16.html#section-3) a more complete description of this datatype.

```
b: #"a"          ;--Character
b: #"A"          ;--Character
b: #"5"          ;--Character
b: #"^A"         ;--Character
b: #"^(1A)"      ;--Hexa form
b: #"^(back)"    ;--Red name form
```

Casting is possible, but limited. Trying to cast an integer value greater than 255 to a byte! will result in a data loss or data corruption, but casting a byte value to integer will succeed.

The **integer!** datatype represents *positive and negative numbers*. The memory size of an integer is *32 bits*, so the range of supported numbers is: -2147483648 to 2147483647. Contrarily to C and other languages, there are no variations around integer! datatype such as unsigned integer or long signed integer. Integer! datatype also supports a **hexadecimal** representation. Hexadecimal integer representation is mostly used to represent memory addresses or binary data for bitwise operations. Allowed range is: 00000000h to FFFFFFFFh. The *0x* prefix is often used in many languages to mark a literal hexadecimal value. It could have been used in Red/System too if the *<number>x<number>* literal form wasn't reserved in Red for the pair! datatype. As Red/System is a DSL, it has to use the same representation for hex values, so *<hexa>h* was chosen.

```
decimal form:             1234
decimal negative form:   -1234
hexadecimal form:         04D2h
```

The **float!** datatype represents an IEEE-754 **double precision** floating point number. Float! *memory size is 64-bit*. **Float64!** Is an alias to float! A maximum of 16 digits is accepted for literal float! values. If more digits are specified, they will be dropped. It is allowed to apply a *type casting* transformation on a float! value to convert it to a float32! value.

The f**loat32!** datatype represents an IEEE-754 **single precision** floating point number. Float32! memory size is *32-bit*. It is also allowed to apply a type casting on a float32! value to convert it to a float64! value. Type casting to integer! is also allowed mainly for bits manipulation purpose. The reason for having a single precision floating point type is for making the interfacing with libraries straightforward. For pure Red/System programs, float! should be the default choice.

Both Float! and Float32 support the standard and the scientific notation.

```
0.0
1.0
-12345.6789
3.14159265358979
-1E3
+1.23456E-265
```

With floats, the printed result is rounded to the closest integer value by default if it is less than an internal epsilon value. There is an option to disable that pretty printing for floats, so you'll get a more accurate output. Just use this instruction in your Red (not Red/S) code: system/options/float/pretty?: no

The **logic!** datatype represents Boolean values: **TRUE** and **FALSE**. Logic variables are initialized using a literal logic value or as the result of a conditional expression.

A **c-string!** value is a sequence of non-null bytes terminated by a null byte. A c-string variable holds the memory address of the first byte of the c-string, so it can be viewed as *an implicit pointer* to a variable of the byte! datatype. A c-string having a null character as first byte is an empty c-string. You don't have to add a null byte to literal c-strings. It is added automatically during compilation. It is possible to retrieve the number of bytes (excluding the null end marker) in a c-string at runtime using the **length?** function. Applied to c-string!, the **size?** function will return the number of bytes in the c-string, including the ending null byte. Since c-string! datatype is a pointer, t is possible to apply some simple math operations like addition and subtraction. C-string address would be increased or decreased by the integer argument.

```
s: "Hello"
print [s lf]     ;"Hello" is printed
s: s + 1         ;--now s points to address + 1
print [s lf]     ;--"ello" would be printed
s: s + 1         ;--now s points to address  + 1
print [s lf]     ;--"llo" would be printed
```

Since c-string is an implicit pointer, when created, the returned value is the content of the c-string and not the address of the c-string. If you want to know the memory address of the c-string, use a pointer: p: as byte-ptr! of c-string. Naturally, It is possible to access individual bytes in a c-string using path notation with an index.

The **struct! datatype** is more or less equivalent to *C struct type*. It is a record of one or several values, each value having its own datatype. A struct variable holds the *memory address* of a struct value. Struct! values definitions tend to be quite long, so in some cases, when struct! definitions are required to be inserted in other struct! definitions or in functions specification block, it is possible to use an **alias** name to reference a struct! definition through the source code. It also allows the self-referencing case to be quite simply solved. An alias is not a value, it doesn't take any space in memory, it can be seen as a **virtual datatype**. So, by convention, alias names should end with an exclamation mark, in order to distinguish them more easily from variables in the source code. Aliased names live in their own namespace, so they cannot interfere with variable names. Returned value when creating a struct! with **declare** Is the address of the initialized structure.

IMPORTANT: Struct! values members are *padded in memory* in order to preserve optimal alignment for each OS target (for example, it is aligned to 4 bytes for IA32 target). Red/System *Size?* function will return the size of the struct! value in memory including the padding bytes.

Pointer datatype's purpose is to hold the memory address of another value. Pointers support both math operations (addition and subtraction) and path notation access. A pointer value is defined by the pointed value address and datatype. As c-string! and struct! are implicit pointers, the reference document underlines that the only pointed datatypes allowed are integer!, float!, float32! and byte! (logic! pointer is not necessary). A pointer can also point to a one-dimensional array of values literally specified which is considered as **literal array**. This is also possible for **binary arrays**. Binary arrays are a type of literal array made of bytes only. But, Red/S pointers can also point to other variables such as structures. This really interesting when a C library contains a lot of nested structures.

```
Red/System [
	Title:	"Pointers"
	Author:	"ldci"
]
;--3 Red programming rules: Readability, Reliability and Reusability.
print ["Let's play with structures and pointers" lf]
print ["Create structure and pointer" lf]
;--First we create a structure as an alias (no memory allocation)
union: alias struct! [
   f       [ float!]
   i       [integer!]
   str     [c-string!]
]
;--then a pointer to the structure
p-struct!: declare struct! [ptr [union]] ;--memory allocation
;--now we have to allocate the structure
un: declare union
un/f: pi
un/i: 255
un/str: "Hello Red world"
;--and to create a pointer to the structure
p-struct!/ptr: un
;--verify structure initialization
print ["Structure address: " un lf]
print ["Pointer value:     " p-struct! lf]
print ["Nice, we are pointing to the structure!" lf]
print [p-struct!/ptr/f " " p-struct!/ptr/i " " p-struct!/ptr/str lf]
print ["Now, let's modify the pointer values" lf]
p-struct!/ptr/f: 0.0
p-struct!/ptr/i: 0
p-struct!/ptr/str: "Bye bye crual world!"
print ["Pointer values are changed" lf]
print [p-struct!/ptr/f " " p-struct!/ptr/i " " p-struct!/ptr/str  lf]
print ["And the structure too! " lf]
print [un/f " " un/i " " un/str lf]

The result is :
Let's play with structures and pointers
Create structure and pointer
Structure address: 000030D0
Pointer value:     000030C4
Nice, we are pointing to the structure!
3.141592653589793 255 Hello Red world
Now, let's modify the pointer values
Pointer values are changed
0.0 0 Bye bye crual world!
And the structure too!
0.0 0 Bye bye crual world!
```

In Red/System, byte, integer and float pointers are defined:

```
p: declare pointer! [integer!]       ;-- int *p;
p: declare pointer! [byte!]          ;-- char *p;
p: declare pointer! [float!]         ;-- double *p;
```

But you can add new forms of pointer as I had to do for OpenCV binding:

```
#define double-byte-ptr!    [struct! [ptr [byte-ptr!]]]    ; equivalent to C's byte **
#define double-int-ptr!     [struct! [ptr [int-ptr!]]]     ; equivalent to C's int **
#define double-float-ptr!   [struct! [ptr [float-ptr!]]]   ; equivalent to C's double **
#define p-buffer!           [struct! [buffer [c-string!]]] ; equivalent to C's char **
```

There is no specific support in Red/System for C-like **void** pointers. The official way is to use a byte-ptr! type to represent C void* pointers. A special **null** value is available to use for pointer! and other pointer-like types and pseudo-type function! Null does not have a specific type, but can be used to replace any other pointer-like value. So, null cannot be used as initializing value for a global variable or a local variable that does not have an explicit type specification. Null is a **first class value**, so it can be assigned to a variable, passed as argument to a function or returned by a function. It is not possible to explicitly cast null to a given type, only implicit type casting automatically done by the compiler is allowed.
Dereferencing a pointer is defined as the operation allowing access to the pointed value. In Red/System, it is achieved by adding *a /value refinement* to the pointer variable.

```
p: as int-ptr!
p/value: 1234      ;-- write 1234
print  p/value     ;--get pointed value
```

Basically, **binary arrays** are similar to c-strings and can be considered as pointers, and thus support both math operations (addition and subtraction) and path notation access. This datatype is really useful when you need to process large part of memory made of bytes such as images, videos or sounds files.

```
Red/System [
	Title:   "Binary Arrays"
	Author:  "ldci"
]

binArray: #{48656C6C6F20576F726C64}
print ["String address: " binArray lf]
n: size? binArray
print ["String Size:    " n lf]                          ;--size? for pointer
print ["String Length:  " length? as c-string! binArray lf]  ;--length? for string
print ["Integer Representation:  "]
i: 1
while [i <= n][
	print [as integer! binArray/i]
	i: i + 1
]
print lf

print ["Char Representation:     "]
i: 1
while [i <= n][
	print [binArray/i " "]
	i: i + 1
]
print lf

print ["C-String Representation: "]
until [
    print binArray/1
    binArray: binArray + 1
    binArray/1 = null-byte
]
print  lf
```

The result:

```
String address: 0000307C
String Size:    11
String Length:  11
Integer Representation:  721011081081113287111114108100
Char Representation:     H e l l o  W o r l d
C-String Representation: Hello World
```

A pointer can also point to **literal array** i.e. a one-dimensional array of values literally specified. The array is *statically allocated* and can be accessed using pointer path notation or pointer arithmetic. The size of a literal array can be retrieved using *size?* function.
Let’s begin with a simple sample with. A block of integers:

```
blk: [0 1 2 3 4 5 6 7 8 9]
n: size? blk
print ["Size: " n lf]
i: 1
while [i <= n][
        print [blk/i lf]
        i: i + 1
]
>>
Block
Size: 10
0
1
2
3
4
5
6
7
8
9
```

This allows works with blocks of float! and blocks of chars:

```
s: [#"h" #"e" #"l" #"l" #"o"]                          ;--s is an implicit pointer
print ["Block of Chars [Bytes]" lf]
print ["Block address:    " s lf]
n: size? s
print ["Size: " n " Content:  "]
i: 1
while [i <= n][
        print [s/i]
        i: i + 1
]
>>
Block of Chars [Bytes]
Block address:    0000313C
Size: 5 Content:  hello
```

Mixing the different allowed types in the same array is permitted (enumerations can also be used). In such case, the size of each array's slot is either 32-bit, or 64-bit if a float64! value is present.
A first sample with 32-bit slots. In this case, path notation can be used to get values, but you have to cast the value with the correct datatype. If not casted, you’ll get *a Runtime Error 1: access violation* message.

```
print ["Mixed block 1" lf]
c: [456 "hello" "Red" "world" #"e" true];--all values are 32-bit
print size? c
print lf
print [as integer! c/1 lf]
print [as c-string! c/2 lf]
print [as c-string! c/3 lf]
print [as byte! c/5 lf]
print [as logic! c/6 lf]
>>
Mixed block 1
6
456
hello
Red
e
true
```

With float! values, this a little bit complicated, but supported.

```
mblk: [1.05 "Hello" 123 "World!" 3.14]
```

You have to create a float-pointer to access 64-bit float values. For the 32-bit values, casting is necessary, and you have to increment the data index according to the presence of 64-bit values.

```
pf: as float-ptr! mblk              ;--OK we have float values inside the block
probe pf/1                          ;--float value index 1 64-bit
probe as c-string! mblk/3           ;--add 2 to array index
probe mblk/5                        ;--add 2 to array index
probe as c-string! mblk/7           ;--add 2 to array index
probe pf/5                          ;--float value index 5 64-bit
```

About casting

In Red/System casting is achieved using the *as* keyword followed by the target type and the value to cast. *As*-target type (e.g. as-integer 3.14) is also supported, but the first notation (e.g. as integer! 3.14) is better for me to avoid confusion with Red language casting with *to*- prefix (e.g. to-integer 3.14). In Red language, as- is reserved for some specific castings such as-color, as-ipv4, as-money, as-pair, and as-rgba. In Red/System, type casting is possible between values of compatible types. You'll find in https://static.red-lang.org/red-system-specs.html , a matrix about compatible casting.

About size? Function

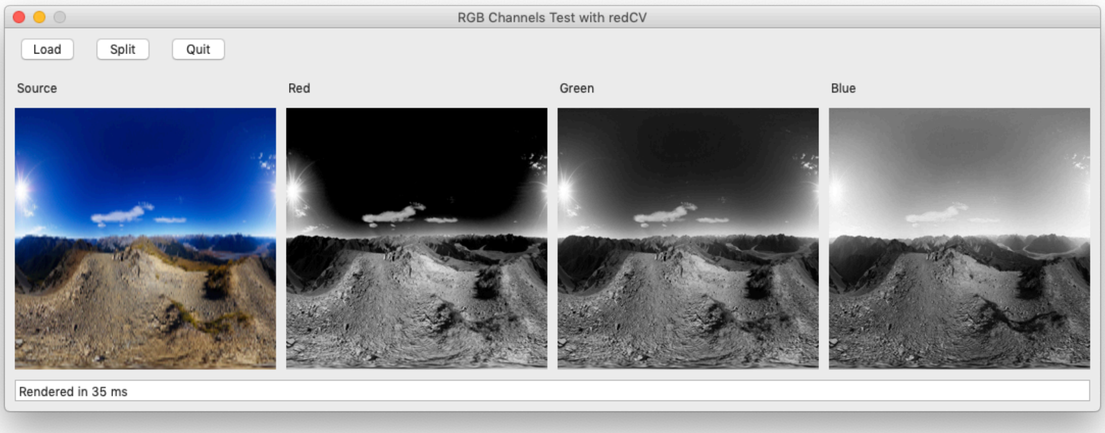
This function returns the memory storage size in bytes required by a value of given type. When passed a c-string literal value, it will return the number of bytes in the c-string, including the ending null byte. Knowing the size of values type is very important when values must be padded in memory, mainly when interfacing with C libraries.

```
aStruct!: alias struct! [
   f      [float!]
   i      [integer!]
   str    [c-string!]
]
rcvRect!:  alias struct! [
            x                                  [integer!]
            y                                  [integer!]
            width                              [integer!]
            height                             [integer!]
            weight                             [float!]
]

print ["size? tests (bytes)" lf]
print ["Byte!:      " size? byte! lf]
print ["Integer!:  " size? integer! lf]
print ["Float32!:  " size? float32! lf]
print ["Float!:      " size? float! lf]
print ["Logic!:      " size? logic! lf]
print ["Byte-ptr!:     " size? byte-ptr! lf]
print ["Int-ptr!:       " size? int-ptr! lf]
print ["Float32-ptr!:   " size? float32-ptr! lf]
print ["Float-ptr!:      " size? float-ptr! lf]
print ["Structure 1:   " size? aStruct! lf]
print ["Structure 2:   " size? rcvRect! lf]
>>
size? tests (bytes)
Byte!:              1
Integer!:           4
Float32!:           4
Float!:             8
Logic!:             4
Byte-ptr!:          4
Int-ptr!:           4
Float32-ptr!:       4
Float-ptr!:         4
Structure 1:        16
Structure 2:        24
```

Using Red/System with Red

Red/system can be embedded inside Red applications for fast code support. Whenever Red application needs to run faster, we can rewrite slow red code functions in Red/System functions to achieve huge performance gains. For example, in Red, splitting a 2048x1024 image into 3 RGB channels requires more than 3 seconds. In Red/System, only 35 ms!



Now, we need a way to call into Red/System from Red. This can be easily done with the specific Red *routine,* which defines a function with a given Red specifications and Red/System body. This is an example from Ivo Balbaert's book: *Learn Red – Fundamentals of Red.* First, we have to define a Red/S function for calculating factorials:

```
fact: func [
n [integer!]
     return: [integer!]
  ][
     if n = 0 [ return 1]
     n * fact n - 1
]
```

Now we have to call our Red/System fact function from a Red script:

```
#system-global [#include %fact.reds]
red-fact: routine [
             n          [integer!]
            return:   [integer!]
][
   fact n  ; function from fact.reds
]
print red-fact 9 ;== 362880
```

In the first line, the Red/System script is included as a usual Red script, but it must be prefixed by *#system-global*, a compiler directive. This line also executes the Red/System script. Then we write a routine which calls the Red/S function. This approach is functional and efficient, but I prefer to directly write the Red/S code as routines and not linking a Red/S function to a Red routine. This is the way I mostly adopted for redCV. A simple #include is sufficient and routines can be accessed everywhere in the application.

```
rcvCopyImage: routine [
"Copy source image to destination image"
   src      [image!]
   dst      [image!]
   /local
      pixS [int-ptr!]
      pixD [int-ptr!]
      handleS handleD i n [integer!]
][
   handleS: 0
   handleD: 0
   pixS: image/acquire-buffer src :handleS
   pixD: image/acquire-buffer dst :handleD
   n: IMAGE_WIDTH(src/size) * IMAGE_HEIGHT(src/size)
   i: 0
   while [i < n] [
            pixD/value: pixS/value
             pixS: pixS + 1
            pixD: pixD + 1
            i: i + 1
   ]
   image/release-buffer src handleS no
   image/release-buffer dst handleD yes
]
;--now we can call the routine in any Red function or Red code
rcvCloneImage: function [
"Returns a copy of source image"
            src         [image!]
][
            dst: make image! src/size  ;--Red code
            rcvCopyImage src dst       ;--Red/S code
            dst
]
```

This approach is very useful for end-user: no needs to know how Red/S code is implemented, just use routines as classical Red functions.

Using pointers with Red

Routines are very efficient, but limited in specification block only to Red datatypes. In many C libraries, functions require to use a pointer, which is unsupported by Red language. But pointer is just a memory address expressed as an integer! datatype. So, we can use integer to communicate between Red/S. functions and Red routines. This sample comes from my OpenCV binding for Red.

```
loadImg: routine [
        name                [string!]
        return:             [integer!]
        /local
        fName               [c-string!]
        &cvImage            [integer!]
        cvImage             [int-ptr!]
][
        &cvImage: 0                                ;--pointer address initialization
        fName: as c-string! string/rs-head name    ;--from Red string! to Red/S c-string!
        cvImage: as int-ptr! cvLoadImage fName CV_LOAD_IMAGE_COLOR;--get image as a pointer
        if cvImage <> null [
                cvFlip cvImage cvImage -1          ;--Red/S function using a pointer as parameter
                &cvImage: as integer! cvImage      ;--Pointer address  as an integer
        ]
        &cvImage ;--returns the pointer address
]
```

This kind of approach is very efficient when you have a large amount of binary data to process such as image or sound files. In these cases, we can use a byte-ptr!

```
;--Thanks to Qtxie for the optimization!
getBinaryValue: routine [
        dataAddress         [integer!] ;--pointer address
        dataSize            [integer!] ;--binary data size
        return:             [binary!]
] [
        as red-binary! stack/set-last as red-value! binary/load as byte-ptr! dataAddress dataSize
]
```

Undocumented *#get* compiler directive

By choice, Red and Red/S namespaces are separated in order to avoid border-effects. The global variables defined in Red are not directly accessible inside the routines. However, global variables can be passed as parameters to routines. Sometimes, it's crucial to access to Red variables inside a Red/S function or a Red routine. This can be done with the #get compiler directive which uses red words (#get 'word).

```
Red [
	Title:   "Strings"
	Author:  "ldci"
]
;--the way to share global Red words with Red/System code
s: "Hello Red"				;--string variable
b: #{48656C6C6F20526564}		;--same string as binary

;--Routines for accessing Red words
getString: routine [
	return: [string!]
][as red-string! #get 's]
getBinary: routine [
	return: [binary!]
][as red-binary! #get 'b]

;--Red code
print ["Test String: " getString]
append s " is amazing"
print ["Test String: " getString]
print ["Test Binary: " getBinary]

>>
Test String:   Hello Red
Test String:  Hello Red is amazing
Test Binary:  #{48656C6C6F20526564}
```

The treatment of Red/System variables is different. The values defined by *#define* are only accessible in the *#system space* but not in the routines. This is because #define is not an assignment, but a compiler substitution macro that is reserved for the system space. On the other hand, the values in *#enum* are accessible without problem in routines because the enumeration elements are assigned and can be imported into the routines that interface with Red/System. When I was writing Haar cascade classifiers for facial recognition in redCV, I had to share vectors between Red and Red/System. This is the way I adopted. I first created a set of variables accessible by Red/S with the *#system-global* compiler directive.

```
#system-global [
	#include %structures/hStructs.reds		; --Red/System structures
	cascade:		declare rcvCascade!		;--classifier cascade
	equRect:		declare rcvRect!		;--rectangle
	tr:			declare rcvRect!		;--rectangle
	ssum:			declare rcvIntImage!		;--summed-area table
	sqsum:			declare rcvIntImage!		;--square summed-area table
	stsum:			declare rcvIntImage!		;--tilted summed-area table
	winSize0:		declare rcvSize!		;--window size of the training set
	winSize:		declare rcvSize!		;--size of the image scaled up or down
	sarray:			declare int-ptr!		;--stagesArray
```

```
        rarray:             declare int-ptr!            ;--rectanglesArray
        srarray:            declare int-ptr!            ;--scaledRectanglesArray
        oarray:             declare int-ptr!            ;--tiltedArray
        warray:             declare float-ptr!          ;--weightsArray
        a1array:            declare float-ptr!          ;--alpha1Array left values
        a2array:            declare float-ptr!          ;--alpha2Array right values
        n1array:            declare float-ptr!          ;--nodes left value
        n2array:            declare float-ptr!          ;--nodes right values
        l1array:            declare int-ptr!            ;--nodes has left nodes?
        l2array:            declare int-ptr!            ;--nodes has right nodes ?
        tarray:             declare float-ptr!          ;--treeThreshArray
        starray:            declare float-ptr!          ;--stagesThreshArray
        unSigned:           pow 2.0 32.0                ;--for c-like uint conversion
]
```

Then, I created a set of variables for Red language.

```
stagesArray:                    make vector! []             ;--nStages
rectanglesArray:                make vector! []             ;--totalNodes * 12
scaledRectanglesArray:          make vector! []             ;--totalNodes * 12
tiltedArray:                    make vector! []             ;--totalNodes
logic1Array:                    make vector! []             ;--totalNodes
logic2Array:                    make vector! []             ;--totalNodes
weightsArray:                   make vector! [float! 64 0] ;--totalNodes * 3
alpha1Array:                    make vector! [float! 64 0] ;--totalNodes
alpha2Array:                    make vector! [float! 64 0] ;--totalNodes
node1Array:                     make vector! [float! 64 0] ;--totalNodes
node2Array:                     make vector! [float! 64 0] ;--totalNodes
leftArray:                      make vector! [float! 64 0] ;--totalNodes
rightArray:                     make vector! [float! 64 0] ;--totalNodes
treeThreshArray:                make vector! [float! 64 0] ;--totalNodes
stagesThreshArray:              make vector! [float! 64 0] ;--nStages
```

Finally I wrote a routine that links the Red/S and Red variables with #get directive. This way, Red/S variables can be called from Red/S or from Red with a pointer inside routines.

```
rcvCreateArrayPointers: routine [
"Create integer or float pointers that give access to Red arrays by routines"
][
        sarray:  as int-ptr! vector/rs-head as red-vector!   #get 'stagesArray
        rarray:  as int-ptr! vector/rs-head as red-vector!   #get 'rectanglesArray
        srarray: as int-ptr! vector/rs-head as red-vector!   #get 'scaledRectanglesArray
        oarray:  as int-ptr! vector/rs-head as red-vector!   #get 'tiltedArray
        l1array: as int-ptr! vector/rs-head as red-vector!   #get 'logic1Array
        l2array: as int-ptr! vector/rs-head as red-vector!   #get 'logic2Array
        warray:  as float-ptr! vector/rs-head as red-vector! #get 'weightsArray
        a1array: as float-ptr! vector/rs-head as red-vector! #get 'alpha1Array
        a2array: as float-ptr! vector/rs-head as red-vector! #get 'alpha2Array
        n1array: as float-ptr! vector/rs-head as red-vector! #get 'node1Array
        n2array: as float-ptr! vector/rs-head as red-vector! #get 'node2Array
        tarray:  as float-ptr! vector/rs-head as red-vector! #get 'treeThreshArray
        starray: as float-ptr! vector/rs-head as red-vector! #get 'stagesThreshArray
]
```

This may sound complicated, but it illustrates the power and versatility of Red language. In this application, Red vectors are populated by Red code, and then pointers to vectors are used for fast calculation inside different routines for facial recognition.

Red/System and objects
In summer 2020 with Toomas Vooglaid, we wrote a matrix library for Red. The way we adopted is an object-oriented approach, and thus matrix is a generic Red object which allows to create and process matrices. Matrix object contains a lot of methods for mathematics on matrices. Matrix constructor takes following arguments:

```
mType: matrix type as integer [1: char, 2: integer, 3: float]
mBits:   bit-size as integer [8|16|32 for char and integer, 32|64 for float!]
mSize:  matrix size as pair with COLSxROWS (e.g 3x3)
mData: matrix values as block transformed into vector for fast computation
```

Code is written in Red and is rather fast enough for small matrices. However, for large matrices such as RGBA images, Red/System is necessary. In matrix library the way we adopted was first to create a series of Red/S functions.

```
;--How to access matrix object with routines
;--Thanks to Qingtian Xie:)

#system [
	mat: context [
		;--pure Red/S  code
		#enum matrix-fields! [
			MAT_OBJ_TYPE  ;-- 0
			MAT_OBJ_BITS  ;-- 1
			MAT_OBJ_ROWS;-- 2
			MAT_OBJ_COLS  ;-- 3
			MAT_OBJ_DATA ;-- 4
		]
		get-type: func [
			mObj	[red-object!]
			return: [integer!]
			/local
			val			[red-value!]
			int			[red-integer!]
		][
			val: object/get-values mObj
			int: as red-integer! val + MAT_OBJ_TYPE
			int/value
		]

		get-bits: func [
			mObj	[red-object!]
			return: [integer!]
			/local
			val			[red-value!]
			int			[red-integer!]
		][
			val: object/get-values mObj
			int: as red-integer! val + MAT_OBJ_BITS
			int/value
		]
		get-unit: func [
			mObj	[red-object!]
			return: [integer!]
			/local
			vec			[red-vector!]
			s			[series!]
		][
			vec: mat/get-data mObj
			s: GET_BUFFER(vec)
			GET_UNIT(s)
		]

		get-rows: func [
			mObj	[red-object!]
			return: [integer!]
			/local
			val			[red-value!]
			int			[red-integer!]
		][
			val: object/get-values mObj
			int: as red-integer! val + MAT_OBJ_ROWS
			int/value
		]

		get-cols: func [
			mObj	[red-object!]
			return: [integer!]
			/local
			val			[red-value!]
			int			[red-integer!]
		][
			val: object/get-values mObj
			int: as red-integer! val + MAT_OBJ_COLS
			int/value
		]

		get-data: func [
			mObj	[red-object!]
			return: [red-vector!]
			/local
				val [red-value!]
		][
			val: object/get-values mObj
			as red-vector! val + MAT_OBJ_DATA
		]
	];--end of context
];--end of Red/System
```

Then, we wrote some Red routines calling the pure Red/System code.

```
;--now some routines and functions
getMatType: routine [
	mObj	[object!]
	return:	[integer!]
][
	mat/get-type mObj
]

getMatBits: routine [
		mObj	[object!]
		return:	[integer!]
][
		mat/get-bits mObj
]

getMatOrder: routine [
		mObj	[object!]
		return:	[pair!]
		/local
		rows	[integer!]
		cols	[integer!]
		order	[red-pair!]
][
		order: pair/make-at stack/push* 0 0
		rows: mat/get-rows mObj
		cols: mat/get-cols mObj
		order/x: cols
		order/y: rows
		as red-pair! stack/set-last as cell! order
]

getMatDataLength: routine [
		mObj	[object!]
		return:	[integer!]
		/local
			vec  [red-vector!]
][
		vec: mat/get-data mObj
		vector/rs-length? vec
]

getMatData: routine [
		mObj	[object!]
		return:	[vector!]
		/local
			vec	[red-vector!]
][
		vec: mat/get-data mObj
		as red-vector! stack/set-last as cell! vec
]

getMatUnit: routine [
"Returns matrice unit"
		mObj	[object!]
		return: [integer!]
		/local
		s			[series!]
		vec	[red-vector!]
] [
		vec: mat/get-data mObj
		s: GET_BUFFER(vec)
		GET_UNIT(s)
]
```

```
getMatTypeAsString: function [
		mObj		[object!]
		return:	[string!]
][
		switch/default getMatType mObj [
				1 [return "Char!"]
				2 [return "Integer!"]
				3 [return "Float!"]
		][return "Unknown"]
]
```

Then, it's really easy to write a nice red code to process matrices with Red/System.

```
#!/usr/local/bin/red
Red [
]
#include %../matrix-obj.red
#include %../routines-obj.red

bc: [#"^A" #"^B" #"^C" #"^D" #"^E" #"^F" #"^G" #"^H" #"^@"] ;--char
m1: matrix/create 1 8  3x3 bc
print ["Matrix Type:            " getMatType m1 getMatTypeAsString m1]
print ["Matrix Bit Size:        " getMatBits m1]
print ["Matrix Order:           " getMatOrder m1]
print ["Matrix Data Length: " getMatDataLength m1]
print ["Matrix data:             " getMatData m1]
print ["Matrix unit:             " getMatUnit m1 lf]

m2: matrix/create 2 32 3x3 [1 2 3 4 5 6 7 8 9]                                   ;--integer
print ["Matrix Type:             " getMatType m2 getMatTypeAsString m2]
print ["Matrix Bit Size:         " getMatBits m2]
print ["Matrix Order:            " getMatOrder m2]
print ["Matrix Data Length: " getMatDataLength m2]
print ["Matrix data:             " getMatData m2]
print ["Matrix unit:             " getMatUnit m2 lf]

bf: [1.0 2.0 3.0 4.0 5.0 6.0 7.0 8.0 9.0 10.0 11.0 12.0];-float
m3: matrix/create 3 64 3x4 bf
print ["Matrix Type:             " getMatType m3 getMatTypeAsString m3]
print ["Matrix Bit Size:         " getMatBits m3]
print ["Matrix Order:            " getMatOrder m3]
print ["Matrix Data Length: " getMatDataLength m3]
print ["Matrix data:             " getMatData m3]
print ["Matrix unit:             " getMatUnit m3 lf]
>>
Matrix Type:                1 Char!
Matrix Bit Size:            8
Matrix Order:               3x3
Matrix Data Length:         9
Matrix data:                ^A ^B ^C ^D ^E ^F ^G ^H ^@
Matrix unit:                1

Matrix Type:                2 Integer!
Matrix Bit Size:            32
Matrix Order:              3x3
Matrix Data Length:        9
Matrix data:               1 2 3 4 5 6 7 8 9
Matrix unit:               4

Matrix Type:               3 Float!
Matrix Bit Size:           64
Matrix Order:              3x4
Matrix Data Length:        12
Matrix data:               1.0 2.0 3.0 4.0 5.0 6.0 7.0 8.0 9.0 10.0 11.0 12.0
Matrix unit:               8
```

Boxing

Another way to modify Red global values is to use the *box* function which can be found for most of scalar Red datatypes. All Red types not automatically converted are returned, on the R/S side, as a structure, such as *int: as red-integer!* The definitions are in the file red-master/runtime/datatypes/structures.reds. With structures, you can access values for reading or writing.

```
i: 100                     ;--integer!
f: 0.0                     ;--float!

getIntBoxing: routine [/local int][
        int: as red-integer! #get 'i
        integer/box int/value + 250
]

getFloatBoxing: routine [/local fl][
        fl: as red-float! #get 'f
        float/box fl/value + pi / 2.0
]

print ["Test Boxing: " getIntBoxing]
print ["Test Boxing: " getFloatBoxing]

>>
Test Boxing:  350
Test Boxing:  1.570796326794897
```

#call compiler directive

This is another compiler directive which is interesting, since it allows to integrate Red code inside Red/S code. Red/System can call back a Red-level function, passing arguments, using the #call compiler directive. This is great, because it allows you to substitute functions that don't exist in the Red/System dialect, like here the random function, not present in the earliest versions of Red. Red function can be call within #system block or within a Red routine.

```
Red [
        Title:  "Call"
        Author:  "ldci"
]
;--How to use R/S #call with Red

print ["Red Version : " system/version]
print ["Compiled    : " system/build/config/OS system/build/config/OS-version]
print "Random Tests"
random/seed now/time/precise            ;--generate Random/seed
rand: function [n [integer!]][random n]     ;--returns an integer
;--first solution with #system
#system [
  v: 1000                               ;--Red/S word to be used with routines
  #call [rand v]
  int: as red-integer! stack/arguments
  print ["System Call :  " int/value lf]
]
;--Second solution with Red routine
alea: routine [][
        #call [rand v]                  ;--Use Red/S variable
        int: as red-integer! stack/arguments
        int/value
]

v: 1000                                 ;--this a Red word, not the Red/S word
;--Classical Red Function
print ["Red Function: " rand v]
print ["Red Function: " rand v]
print ["Red Function: " rand v]

;--Red Routine which calls Red Function
print ["Red Routine : " alea]
print ["Red Routine : " alea]
print ["Red Routine : " alea]
>>
Red Version :  0.6.4
Compiled    :  macOS 10.14
Random Tests
System Call :  472
Red Function:  460
Red Function:  84
Red Function:  302
Red Routine :  252
Red Routine :  399
Red Routine :  259
```

Attention: The arguments should be passed as pointers to Red values of the expected type, except for integer! and logic! datatypes, that can be passed inlined, without requiring boxing. When required, for pointer arguments, a simple type casting is allowed. Once the Red function returns, the normal execution of Red/System code continues. The Red return value cannot be directly retrieved currently, but it should be accessible from Red arguments stack. The Red function must have all arguments typed, else a compilation error will occur.

Subroutines
In August 2020, the Red Team introduced a very useful tool for Red/S programming: the *subroutines*. They act as the *GOSUB* directive from Basic language. They are defined as a separate block of code inside a function's body and are called like regular functions (but without any arguments). So they are much lighter and faster than real functions and require just one slot of stack space to store the return address. To define a subroutine, you need to declare a local variable with the subroutine! datatype, then set that variable to a block of code. You can then invoke the subroutine by calling its name from anywhere in the function body (but after the subroutine own definition). I really appreciate this approach and I'm intensively using it in redCV.

```
rcvRandImage: routine [
        dst                 [image!]
        /local
        pixel1 pixel2       [subroutine!]
  pixD                      [int-ptr!]
  handleD i n               [integer!]
  r g b int                 [integer!]
][
        handleD: 0
        r: g: b: 0
        pixD: image/acquire-buffer dst :handleD
         n: IMAGE_WIDTH(dst/size) * IMAGE_HEIGHT(dst/size)
         ;--subroutines
         pixel1: [r: _random/rand and FFh g: _random/rand and FFh b: _random/rand and FFh]
        pixel2: [(255 << 24) OR (r << 16) OR (g  << 8) OR b]

        i: 0
        while [i < n] [
                pixel1
                pixD/value: pixel2
                pixD: pixD + 1
                i: i + 1
        ]
        image/release-buffer dst handleD yes
]
```

Conclusion
Red/System also provides support for low-level CPU register operations (interruptions, I/O, stack, privileged mode) and for an inlined assembler. Red/S gives you complete control over memory allocation. I have to explore these compiler directives. Red is still under development, but Red/System is sufficiently mature for developing real professional applications (https://github.com/ldci/Face). It's really easy to link Red and Red/S code. Whatever you use

Red or Red/S for your application, you have to respect the 3 Red programming rules: Readability, Reliability and Reusability. Enjoy.