LibRed API

Table of Contents

1. Abstract

LibRed is a special version of the Red interpreter and runtime library, suitable for integrating into software built in languages other than Red. In order to allow the non-Red software interact with Red, libRed exposes a dedicated low level API (following either C cdecl or Microsoft stdcall standards) which is described in this document. Among the supported features are:

- Ability to set/get a word's value from global or a local context.
- Shortcut constructors for the most common Red datatypes.
- Conversion functions for Red datatypes compatible with the host language (mostly C).
- Series manipulation from the host language.
- Callbacks allowing Red to call host language functions.
- Console-oriented debugging functions.

Terminology: the term *host* is used to designate the host language or application embedding libRed.

Samples of libRed usage can be found here.

2. Building libRed

Building your local version of libRed is simple:

```
red build libRed
```

or from Rebol's console and Red sources:

```
rc "build libRed"
```

Those command-lines will build the libRed version for C language (using the `cdecl` ABI). If you need the `stdcall` ABI (for Microsoft apps compatibility), you need to use:

```
red build libRed stdcall
```

3. Value references

Red values can be returned by libRed function calls. They are represented as *opaque* 32-bit

references. Those references are short-lived, so they are suitable only for restricted local use, like passing that reference to another libRed function call. Setting such references to host variables is possible, but it should be used **immediately after**. Those references are using a specific memory manager that will only keep references alive for about the next 50 API calls. For example:

```
long a, blk;

a = redSymbol("a");
redSet(a, redBlock(0));                    // returned reference is used immediately
here

blk = redGet(a);
redPrint(blk);                             // safe reference usage

for(i = 0; i < 100, i++) {
    // redAppend(blk, redNone());          // unsafe reference usage!
    redAppend(redGet("a"), redNone());     // safe version
}
```

4. C API

The C API can be used with C/C++ applications, but also for integrating Red within any other programming language having a C-compatible FFI.

4.1. Managing the library

A libRed *instance* needs to be created in order to use any function from the API.

NOTE	Currently, only a single libRed session per process is allowed. It is intended that this will be extended in the future to allow multi-instances support.

4.1.1. redOpen()

```
void redOpen(void)
```

Initializes a new Red runtime library session. This function *must* be called before calling any other API function. It is safe to call it several times in the same process, only one session will be opened anyway.

NOTE	If another function is called before `redOpen`, the return value of that function will be `-2`, indicating an illegal access attempt.

4.1.2. redClose()

```
void redClose(void)
```

Terminates the current Red runtime library session, freeing all allocated resources.

4.2. Running Red code

The host software can run Red code directly, using different level of control, from providing Red code in text form to be evaluated, down to calling any Red function directly, and passing arguments constructed on the host side.

4.2.1. redDo()

```
red_value redDo(const char* source)
```

Evaluates the Red expression passed as string and returns the last Red value.

Examples

```
redDo("a: 123");

redDo("view [text {hello}]");

char *s = (char *) malloc(100);
const char *caption = "Hello";
redDo(sprintf(s, "view [text \"%s\"]", caption));
```

4.2.2. redDoFile()

```
red_value redDoFile(const char* filename)
```

Loads and evaluates the Red script referred by *filename* and returns the last Red value. The *filename* is formatted using Red OS-independent conventions (basically Unix-style).

Examples

```
redDoFile("hello.red");
redDoFile("/c/dev/red/demo.red");
```

4.2.3. redDoBlock()

```
red_value redDoBlock(red_block code)
```

Evaluates the argument block and returns the last Red value.

Example

```
redDoBlock(redBlock(redWord("print"), redInteger(42)));
```

4.2.4. redCall()

```
red_value redCall(red_word name, ...)
```

Invokes the Red function (of `any-function!` type) referenced by *name* word, passing to it any required arguments (as Red values). Returns the function's last Red value. The arguments list **must** terminate with a `null` or `0` value, as end marker.

Example

```
redCall(redWord("random"), redInteger(6), 0);     // returns a random integer! value
between 1 and 6
```

4.3. Registering a callback function

Responding to an event occuring in Red, or redirecting some Red calls to the host side (like redirecting `print` or `ask`) requires a way to call back a host function from Red side. This can be achieved using `redRoutine()` function.

4.3.1. redRoutine()

```
red_value redRoutine(red_word name, const char* spec, void* func_ptr)
```

Defines a new Red routine named *name*, with *spec* as specification block and *func-ptr* C function pointer as body. The C function **must** return a Red value (`redUnset()` can be used to signify that the return value is not used).

Example

```
#include "red.h"
#include <stdio.h>

red_integer add(red_integer a, red_integer b) {
    return redInteger(redCInt32(a) + redCInt32(b));
}

int main(void) {
    redRoutine(redWord("c-add"), "[a [integer!] b [integer!]]", (void*) &add);
    printf(redCInt32(redDo("c-add 2 3")));
    return 0;
}
```

4.4. Making Red values from C

Many functions from the libRed API require passing Red values (as *references*). The following functions are simple constructors for the most commonly used datatypes.

4.4.1. redSymbol()

```
long redSymbol(const char* word)
```

Returns a symbol ID associated with the loaded *word* (provided as a C string). This ID can then be passed to other libRed API functions requiring a symbol ID instead of a word value.

Example

```
long a = redSymbol("a");
redSet(a, redInteger(42));
printf("%l\n", redGet(a));
```

4.4.2. redUnset()

```
red_unset redUnset(void)
```

Returns an `unset!` value.

4.4.3. redNone()

```
red_none redNone(void)
```

Returns a `none!` value.

4.4.4. redLogic()

```
red_logic redLogic(long logic)
```

Returns a `logic!` value. A *logic* value of `0` gives a `false` value, all other values give a `true`.

4.4.5. redDatatype()

```
red_datatype redDatatype(long type)
```

Returns a `datatype!` value corresponding to the *type* ID, which is a value from `RedType` enumeration.

4.4.6. redInteger()

```
red_integer redInteger(long number)
```

Returns an `integer!` value from *number*.

4.4.7. redFloat()

```
red_float redFloat(double number)
```

Returns a `float!` value from *number*.

4.4.8. redPair()

```
red_pair redPair(long x, long y)
```

Returns a `pair!` value from two integer values.

4.4.9. redTuple()

```
red_tuple redTuple(long r, long g, long b)
```

Returns a `tuple!` value from three integer values (usually for representing RGB colors). Passed arguments will be truncated to 8-bit values.

4.4.10. redTuple4()

```
red_tuple redTuple4(long r, long g, long b, long a)
```

Returns a `tuple!` value from four integer values (usually for representing RGBA colors). Passed arguments will be truncated to 8-bit values.

4.4.11. redBinary()

```
red_binary redBinary(const char* buffer, long bytes)
```

Returns a `binary!` value from a memory `buffer` pointer and the buffer's length in bytes. The input buffer will be copied internally.

4.4.12. redImage()

```
red_image redImage(long width, long height, const void* buffer, long format)
```

Returns an `image!` value from a memory `buffer` pointer. Image's size is defined in pixels by `width` and `height`. The input buffer will be copied internally. Accepted buffer formats are:

- `RED_IMAGE_FORMAT_RGB`: 24-bit per pixel.
- `RED_IMAGE_FORMAT_ARGB`: 32-bit per pixel, alpha channel leading.

4.4.13. redString()

```
red_string redString(const char* string)
```

Returns a `string!` value from *string* pointer. Default expected encoding for the argument string is UTF-8. Other encodings can be defined using the `redSetEncoding()` function.

4.4.14. redWord()

```
red_word redWord(const char* word)
```

Returns a `word!` value from a C string. Default expected encoding for the argument string is UTF-8. Other encodings can be defined using the `redSetEncoding()` function. Strings which cannot be loaded as words will return an `error!` value.

4.4.15. redBlock()

```
red_block redBlock(red_value v,...)
```

Returns a new `block!` series built from the arguments list. The list **must** terminate with a `null` or `0` value, as end marker.

Examples

```
redBlock(0);                                    // Creates an empty block
redBlock(redInteger(42), redWord("hi"), 0);   // Creates [42 hi] block
```

4.4.16. redPath()

```
red_path redPath(red_value v, ...)
```

Returns a new `path!` series built from the arguments list. The list **must** terminate with a `null` or `0` value, as end marker.

Example

```
redDo("a: [b 123]");
long res = redGetPath(redPath(redWord("a"), redWord("b"), 0));
printf("%l\n", redCInt32(res));    // will output 123
```

4.4.17. redLoadPath()

```
red_path redLoadPath(const char* path)
```

Returns a `path!` series built from a path expressed as a C string. This provides a quick way to build paths without constructing individually each element.

Example

```
redGetPath(redLoadPath("a/b"));    // Creates and evaluates the a/b path! value.
```

4.4.18. redMakeSeries()

```
red_value redMakeSeries(unsigned long type, unsigned long slots)
```

Returns a new `series!` of type *type* and enough size to store *slots* elements. This is a generic series constructor function. The type needs to be one of the `RedType` enumeration values.

Examples

```
redMakeSeries(RED_TYPE_PAREN, 2);  // Creates a paren! series

long path = redMakeSeries(RED_TYPE_SET_PATH, 2); // Creates a set-path!
redAppend(path, redWord("a"));
redAppend(path, redInteger(2));    // Now path is `a/2:`
```

4.5. Making C values from Red

Converting Red values to *host* side is possible, though, restricted by the limited number of types in C language.

4.5.1. redCInt32()

```
long redCInt32(red_integer number)
```

Returns a 32-bit signed integer from a Red `integer!` value.

4.5.2. redCDouble()

```
double redCDouble(red_float number)
```

Returns a C double floating point value from a Red `float!` value.

4.5.3. redCString()

```
const char* redCString(red_string string)
```

Returns a UTF-8 string buffer pointer from a Red `string!` value. Other encodings can be defined using the `redSetEncoding()` function.

4.5.4. redTypeOf()

```
long redTypeOf(red_value value)
```

Returns the type ID of a Red value. The type ID values are defined in the `RedType` enumeration. See Datatypes section.

4.6. Calling a Red action

It is possible to call any Red function using `redCall`, though, for the most common actions, some shortcuts are provided for convenience and better performances.

4.6.1. redAppend()

```
red_value redAppend(red_series series, red_value value)
```

Appends a *value* to a *series* and returns the series at head.

4.6.2. redChange()

```
red_value redChange(red_series series, red_value value)
```

Changes a *value* in *series* and returns the series after the changed part.

4.6.3. redClear()

```
red_value redClear(red_series series)
```

Removes *series* values from current index to tail and returns series at new tail.

4.6.4. redCopy()

```
red_value redCopy(red_value value)
```

Returns a copy of a non-scalar value.

4.6.5. redFind()

```
red_value redFind(red_series series, red_value value)
```

Returns the *series* where a *value* is found, or `none`.

4.6.6. redIndex()

```
red_value redIndex(red_series series)
```

Returns the current index of *series* relative to the head, or of word in a context.

4.6.7. redLength()

```
red_value redLength(red_series series)
```

Returns the number of values in the *series*, from the current index to the tail.

4.6.8. redMake()

```
red_value redMake(red_value proto, red_value spec)
```

Returns a new value made from a *spec* for that *proto*'s type.

4.6.9. redMold()

```
red_value redMold(red_value value)
```

Returns a source format string representation of a value.

4.6.10. redPick()

```
red_value redPick(red_series series, red_value value)
```

Returns the *series* at a given index *value*.

4.6.11. redPoke()

```
red_value redPoke(red_series series, red_value index, red_value value)
```

Replaces the *series* at a given *index* with the *value*, and returns the new value.

4.6.12. redPut()

```
red_value redPut(red_series series, red_value index, red_value value)
```

Replaces the value following a key in a *series* or `map!` value, and returns the new value.

4.6.13. redRemove()

```
red_value redRemove(red_series series)
```

Removes a value at current *series* index and returns series after removal.

4.6.14. redSelect()

```
red_value redSelect(red_series series, red_value value)
```

Find a *value* in a *series* and return the next value, or `none`.

4.6.15. redSkip()

```
red_value redSkip(red_series series, red_integer offset)
```

Returns the *series* relative to the current index.

4.6.16. redTo()

```
red_value redTo(red_value proto, red_value spec)
```

Converts *spec* value to a datatype specified by *proto*.

4.7. Accessing a Red word

Setting a Red word or getting the value of a Red word is the most direct way to pass values between the *host* and Red runtime environment.

4.7.1. redSet()

```
red_value redSet(long id, red_value value)
```

Sets a word defined from *id* symbol to *value*. The word created from the symbol is bound to global context. *value* is returned by this function.

4.7.2. redGet()

```
red_value redGet(long id)
```

Returns the value of a word defined from *id* symbol. The word created from the symbol is bound to global context.

4.8. Accessing a Red path

Paths are very flexible way to access data in Red, so they have their dedicated accessor functions in libRed. Notably, they allow access to words in object contexts.

4.8.1. redSetPath()

```
red_value redSetPath(red_path path, red_value value)
```

Sets a *path* to a *value* and returns that *value*.

4.8.2. redGetPath()

```
red_value redGetPath(red_path path)
```

Returns the *value* referenced by the *path*.

4.9. Accessing a Red object field

When multiple setting/getting accesses are needed on an object's fields, using the object value directly instead of building paths is simpler and preferable. The following two functions are tailored for such access.

NOTE	These accessors can work with any other associated array types, not just `object!`. So passing a `map!` is allowed too.

4.9.1. redSetField()

```
red_value redSetField(red_value object, long field, red_value value)
```

Sets an *object*'s *field* to a *value* and returns that *value*. The *field* argument is a symbol ID created using `redSymbol()`.

4.9.2. redGetField()

```
red_value redGetField(red_value obj, long field)
```

Returns the *value* stored in the *object*'s *field*. The *field* argument is a symbol ID created using `redSymbol()`.

4.10. Debugging

Some handy debugging functions are also provided. Most of them require a system shell window for the output, though, it is possible to force the opening of a log window, or redirect the output to a file.

4.10.1. redPrint()

```
void redPrint(red_value value)
```

Prints the *value* on the standard output, or in the debug console if opened.

4.10.2. redProbe()

```
red_value redProbe(red_value value)
```

Probes the *value* on the standard output, or in the debug console if opened. The *value* is returned from this function call.

4.10.3. redHasError()

```
red_value redHasError(void)
```

Returns an `error!` value if an error has occured in previous API call, or `null` if there no error occured.

4.10.4. redFormError()

```
const char* redFormError(void)
```

Returns a UTF-8 string pointer containing a formatted error if an error has occured, or `null` if there no error occured.

4.10.5. redOpenLogWindow()

```
int redOpenLogWindow(void)
```

Opens the log window and redirects all the Red printing output to that window. This feature is useful if the host application is not run from the system shell, which is used by default for the printing output. Calling this function several times will have no effect if the log window is already opened. Returns `1` on success, `0` on failure.

NOTE	Only for Windows platforms.

4.10.6. redCloseLogWindow()

```
int redCloseLogWindow(void)
```

Closes the log window. Calling this function when the log window is already closed will have no effect. Returns `1` on success, `0` on failure.

NOTE	Only for Windows platforms.

4.10.7. redOpenLogFile()

```
void redOpenLogFile(const string *name)
```

Redirects the output from Red printing functions to the file specified by *name*. A relative or absolute path can be provided in *name* using OS-specific file path format.

4.10.8. redCloseLogFile()

```
void redCloseLogFile(void)
```

Closes the log file opened with `redOpenLogFile()`.

NOTE	Currently, the log file **must** be closed on exit, otherwise a lock is kept on it and this can even cause freezing or crashes in some hosts (like MS Office applications).

4.11. Datatypes definition

Some functions from libRed API can refer to Red datatypes: `redTypeOf()`, `redMakeSeries()` and `redDatatype()`. Red datatypes are represented on the host side, as an enumeration (`RedType`), where types are names using the following scheme:

```
RED_TYPE_<DATATYPE>
```

The exhaustive list can be found here.

5. Visual Basic API

The Visual Basic API can be used both for VB and VBA (from MS Office applications). It is essentially the same as the C API, so only differences will be described in the sections below. The differences are mostly in the variadic functions, which are split into two flavors:

- `redBlock()`, `redPath()`, `redCall()` only accept Red values, and do not require a terminal `null` or `0` value, like the C version.
- `redBlockVB()`, `redPathVB()`, `redCallVB()` only accept VB values, which are automatically converted according to the following table:

VisualBasic	Red
`vbInteger`	`integer!`
`vbLong`	`integer!`
`vbSingle`	`float!`
`vbDouble`	`float!`
`vbString`	`string!`

5.1. Setting up

In order to use libRed with VB/VBA, you need a version of the libRed binary that is compiled for `stdcall` ABI. If you need to recompile such version:

```
red build libRed stdcall
```

You will also need to import the `libRed.bas` module file in your project.

5.2. redLogic()

```
Function redLogic(bool As Boolean) As Long
```

Returns a Red `logic!` value constructed from a VB `boolean` value.

5.3. redBlockVB()

```
Function redBlockVB(ParamArray args() As Variant) As Long
```

Returns a new `block!` series built from the arguments list. The arguments number is variable and composed of VisualBasic values only.

Examples

```
redProbe redBlockVB()              ' Creates an empty block
redProbe redBlockVB(42, "hello")   ' Creates the [42 "hello"] block
```

5.4. redPathVB()

```
Function redPathVB(ParamArray args() As Variant) As Long
```

Returns a new `path!` series built from the arguments list. The arguments number is variable and composed of VisualBasic values only.

Examples

```
redDo("a: [b 123]")
res = redGetPath(redPathVB("a", "b"))
Debug.print redCInt32(res)        ' will output 123
```

5.5. redCallVB()

```
Function redCallVB(ParamArray args() As Variant) As Long
```

Invokes the Red function (of `any-function!` type) referenced by the string passed (first argument), passing it eventually some arguments (as VisualBasic values). Returns the function's last value. The

arguments number is variable and composed of VisualBasic values only.

Example

```
redCallVB("random", 6);            ' returns a random integer! value between 1 and 6
```

5.6. Registering a callback function

Creating a VisualBasic function that can be called from Red side is done like in C API, using the `redRoutine()` call. The last argument for that function is a function pointer. In VB, such pointer can be acquired only for a function defined in a *module*, but not in a *UserForm*.

This is the callback used by the Excel "Red Console" demo:

```
Sub RegisterConsoleCB()
    redRoutine redWord("print"), "[msg [string!]]", AddressOf onConsolePrint
End Sub

Function onConsolePrint(ByVal msg As Long) As Long
    If redTypeOf(msg) <> red_unset Then Sheet2.AppendOutput redCString(msg)
    onConsolePrint = redUnset
End Function
```