LibRed API

目次

1. 概要

LibRedはRedのインタプリタとランタイムライブラリの特別なバージョンです。libRedはRed以外の言語でソフトウェアを開発する場合に適しています。RedではないソフトウェアがRedとやりとりすることを可能にするために、libRedはこのドキュメントに記載されている専用のローレベルAPIを（CのcdeclかMicrosoftのstadcallで）公開します。サポートされる機能は以下の通りです。

- グローバルまたはローカルなコンテキストのwordへの値の設定と取得
- 最も一般的なRedのデータ型のコンストラクタへのショートカット
- Cと互換のあるRedのデータ型に関する関数のコンバージョン
- ホスト言語からのseriesの操作
- Redがホスト言語の関数を呼び出し可能にするコールバック機能
- コンソール向けのデバッグ関数

用語： ホスト という用語はホスト言語またはlibRedを組み込んだアプリケーションのことを指します。

libRedを使用したサンプルは こちら で確認することができます。

2. libRedのビルド

自身のローカル環境にlibRedをビルドする方法はシンプルです。

```
red build libRed
```

または、RebolコンソールとRedのソースで

```
rc "build libRed"
```

これらのコマンドラインはC言語用（cdeclのABIを使用する）libRedをビルドします。 もし（Microsoftアプリケーションとの互換のために）stdcall ABIが必要の場合、次のようにします。

```
red build libRed stdcall
```

3. 値の参照

libRedの関数呼び出しにより、Redの値が返されます。値は 不透明な 32ビットの参照で表現されます。これらの参照は短命で、別のlibRed関数の呼び出しに渡すなど、ローカルな使用に留めるべきです。ホストの変数にこれらの参照をセットすることは可能ですが、

セットしたら直ちに使用すべきです。これらの参照は特別なメモリマネージャで管理され、約50回のAPI呼び出しの間しか生存しません。以下に例を記載します。

```
long a, blk;

a = redSymbol("a");
redSet(a, redBlock(0));                   // 返された参照は直後のこの箇所で使用されています。

blk = redGet(a);
redPrint(blk);                            // 安全な参照の使用

for(i = 0; i < 100, i++) {
    // redAppend(blk, redNone());         // アンセーフな参照の使用です!
    redAppend(redGet("a"), redNone());    // 安全なバージョン
}
```

4. C API

C APIはC/C++のアプリケーションで使用することができます。また、その他のC互換 FFI を持つすべての言語とRedを統合することができます。

4.1. ライブラリの管理

libRedの インスタンス は他の全てのAPIを使用するために必要です。

NOTE 現在のところ、1つのプロセス内で1つのlibRedセッションのみ許されます。複数インスタンスをサポートするように将来拡張される予定です。

4.1.1. redOpen()

```
void redOpen(void)
```

新しいRedのランタイムライブラリセッションを初期化します。この関数は他のすべての*API*関数を呼び出す前に呼び出しされる必要があります。

NOTE もしredOpenの実行前に他の関数が呼び出しされた場合、その関数の戻り値は「-2」になります。これは不適切なアクセスが試みられたことを示します。

4.1.2. redClose()

```
void redClose(void)
```

現在のRedのランタイムライブラリセッションを終了し、確保されたリソースを開放します。

4.2. Redのコードの実行

ホストソフトウェアはRedのコードをダイレクトに実行できます。これにはいくつかの異なるコントロールレベルがあり、Redのコードの文字列を評価させたり、ホスト側で生成した引数をRedの関数に直接渡して呼び出しするといった方法があります。

4.2.1. redDo()

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

文字列として渡されたRedの式を評価し、最後のRedの値を戻り値として返します。

例

```
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)
```

*filename*で指定されたRedのスクリプトをロード、評価し、最後の値を返します。*filename*はRedのOS非依存の規約（基本的にはUnixスタイルです）を使用してフォーマットされます。

例

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

4.2.3. redDoBlock()

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

引数のブロックを評価し、最後の値を返します。

例

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

4.2.4. redCall()

```
red_value redCall(red_word name, ..., red_integer 0)
```

name wordで指定されたRedの関数（any-

function!型）を実行します。任意の数の（Redの値の）引数を渡すことができ、関数の最後の値を返します。引数の終わりを示すため、「null」か「0」で 終わらなければなりません。

例

```
redCall(redWord("random"), redInteger(6));     // 1から6の間のランダムなinteger!の値を返します。
```

4.3. コールバック関数の登録

Redで発生したイベントにレスポンスしたり、Redの呼び出しを（たとえば「print」や「ask」を）リダイレクトしたりするには、Redの側からホストの関数へコールバックする必要があります。これは「redRoutine()」関数を使うことで実現できます。

4.3.1. redRoutine()

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

name で指定された名前と、 *spec* をスペックブロック、 *func-ptr* で指定されたCの関数ポインタをボディとして、新しいRedのroutineを定義します。指定するCの関数は**Red**の値を返さなければなりません（戻り値が使われないことを示すために「redUnset()」を使うこともできます）。

例

```
#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. CからRedの値を作る

libRed APIの多くの関数は（ *references* として）Redの値を渡す必要があります。以下の関数は最もよく使われるデータ型のシンプルなコンストラクタです。

4.4.1. redSymbol()

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

（Cのstringとして与えられた）ロードされた *word* に関連づけられたシンボルIDを返します。このIDはwordの値のの代わりにシンボルIDを必要とする他のlibRed API関数に渡すことができます。

例

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

4.4.2. redUnset()

```
red_unset redUnset(void)
```

値 *unset!* を返します。

4.4.3. redNone()

```
red_none redNone(void)
```

値 *none!* を返します。

4.4.4. redLogic()

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

「logic!」型の値を返します。*logic* の値が「0」であれば「false」、それ以外であれば「true」が戻り値となります。

4.4.5. redDatatype()

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

type で指定されたIDに対応する「datatype!」の値を返します。IDは「RedType」列挙体で定義されています。

4.4.6. redInteger()

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

number で指定された値を元に「integer!」の値を返します。

4.4.7. redFloat()

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

number で指定された値をもとに「float!」の値を返します。

4.4.8. redPair()

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

2つのinteger値から「pair!」の値を返します。

4.4.9. redTuple()

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

3つのinteger値（通常はRGBカラーを表現するために使われます）を元に「tuple!」の値を返します。渡された引数は8ビット値に切り捨てられます。

4.4.10. redTuple4()

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

4つのinteger値（通常はRGBカラーを表現するために使われます）を元に「tuple!」の値を返します。渡された引数は8ビット値に切り捨てられます。

4.4.11. redBinary()

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

指定されたメモリバッファのポインターとbytesで指定されたバッファの長さから `binary!` の値を返します。受け取ったバッファは内部的にコピーされます。

4.4.12. redImage()

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

指定されたメモリバッファのポインターから `image!` の値を返します。イメージのサイズはピクセル単位の `width` と `height` で定義されます。受け取ったバッファは内部的にコピーされます。指定可能なバッファのフォーマットは以下です。

- `RED_IMAGE_FORMAT_RGB`: 24BPP（24-bit per pixel）のフォーマット
- `RED_IMAGE_FORMAT_ARGB`: 32BPP（32-bit per pixel）とアルファチャンネルのフォーマット

4.4.13. redString()

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

string のポインタから「string!」の値を返します。引数の文字列に対してデフォルトではUTF-8エンコーディングであることが期待されます。他のエンコーディングを定義する場合、「redSetEncoding()」関数を使うことで行えます。

4.4.14. redWord()

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

Cのstringから「word!」の値を返します。引数の文字列に対してデフォルトではUTF-8エンコーディングが期待されます。他のエンコーディングを定義する場合、「redSetEncoding()」関数を使うことで行えます。wordとしてロードできない文字列が渡された場合、値「error!」が返されます。

4.4.15. redBlock()

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

引数のリストから新しいblock!のseriesを返します。リストは終わりを示すため、「null」か「0」で 終わらなければなりません。

例

```
redBlock(0);                                     // 空のblockを生成
redBlock(redInteger(42), redWord("hi"), 0);   // [42 hi] というblockを生成
```

4.4.16. redPath()

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

引数のリストから新しいpath!のseriesを返します。リストは終わりを示すため、「null」か「0」で 終わらなければなりません。

例

```
redDo("a: [b 123]");
long res = redDo(redPath(redWord("a"), redWord("b"), 0);
printf("%l\n", redCInt32(res));    //「123」が出力されます。
```

4.4.17. redLoadPath()

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

Cの文字列として表現されたパスからpath!のseriesを返します。これにより、ここの要素を個別に生成することなく、簡単にパスを生成できます。

例

```
redDo(redLoadPath("a/b"));    // a/bというpath!の値を生成して評価します。
```

4.4.18. redMakeSeries()

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

type で指定した型の新しいseries!を、 *slots* で指定した数の要素が入るサイズで生成します。これはseriesを生成する包括的な関数です。typeは `RedType` 列挙体の値のいずれかである必要があります。

例

```
redMakeSeries(RED_TYPE_PAREN, 2);  // paren! seriesを生成します。

long path = redMakeSeries(RED_TYPE_SET_PATH, 2); // set-path!を生成します。
redAppend(path, redWord("a"));
redAppend(path, redInteger(2));    // pathが `a/2:` に更新されます。
```

4.5. Cの値をRedから作成する

Redの値をホストの側へ変換することは可能ですが、C言語のいくつかのデータ型に限られます。

4.5.1. redCInt32()

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

Redのinteger!の値から32ビット符号付き整数を返します。

4.5.2. redCDouble()

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

Redのfloat!の値からCの倍精度浮動小数点数の値を返します。

4.5.3. redCString()

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

Redのstring!の値からUTF-8の文字列バッファポインタを返します。他のエンコーディングは `redSetEncoding` 関数で指定できます。

4.5.4. redTypeOf()

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

Redの値の型IDを返します。型IDは `RedType` 列挙体で定義されています。データ型 セクションを参照してください。

4.6. Redのactionの呼び出し

`redCall`
を使用することですべてのRedの関数を呼び出しすることができます。ただし、最もよく使われるactionには利便性とパフォーマンスのためいくつかのショートカットが提供されています。

4.6.1. redAppend()

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

value を *series* に追加し、先頭位置をポイントするseriesを返します。

4.6.2. redChange()

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

series の値を *value* で変更し、変更した箇所以降のseriesを返します。

4.6.3. redClear()

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

series の値を現在のインデックスから末尾まで削除し、新しい末尾位置のseriesを返します。

4.6.4. redCopy()

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

非スカラー型の値をコピーします。

4.6.5. redFind()

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

value が見つかった位置の *series* を返すか、NONEを返します。

4.6.6. redIndex()

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

先頭から見た_series_ のインデックスまたはコンテキスト中のwordのインデックスを返します。

4.6.7. redLength()

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

現在位置から末尾までの *series* の中の値の数を返します。

4.6.8. redMake()

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

spec と *proto* の型から、新しい値を作成して返します。

4.6.9. redMold()

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

値をRedのソースフォーマットで文字列化して返します。

4.6.10. redPick()

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

series の指定したインデックスの *value* を返します。

4.6.11. redPoke()

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

series の指定したインデックスの *value* を置き換え、新しい値を返します。

4.6.12. redPut()

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

series の指定したキーに対応する値を置き換え、新しい値を返します。

4.6.13. redRemove()

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

series の現在のインデックスの値を削除し、削除後のseriesを返します。

4.6.14. redSelect()

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

series の中の *value* を探し、見つかればその次の値、見つからなければNONEを返します。

4.6.15. redSkip()

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

現在のインデックスから指定の数だけ移動した *series* を返します。

4.6.16. redTo()

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

spec の値を *proto* で指定されたデータ型に変換します。

4.7. Redのwordへのアクセス

Redのwordのセットやredのwordからの値の取得はホストとRedのランタイム環境との間で値をやり取りする最も直接的な方法です。

4.7.1. redSet()

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

id シンボルで指定されたwordに *value* の値をセットします。これによって作成されたwordはグローバルコンテキストに作成されます。 *value* はこの関数によって返される値になります。

4.7.2. redGet()

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

id シンボルで指定されたwordの値を返します。これによって作成されたwordはグローバルコンテキストに作成されます。

4.8. Redのパスへのアクセス

パスはRedのデータにアクセスできる非常にフレキシブルな方法です。そのため、libRedにおいて専用のアクセス方法があります。とりわけ、それはオブジェクトコンテキスト内のwordへのアクセスを可能にします。

4.8.1. redSetPath()

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

path に *value* をセットし、その *value* を返します。

4.8.2. redGetPath()

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

path で参照される *value* を返します。

4.9. Red のオブジェクトフィールドへのアクセス

オブジェクトのフィールドへの複数のセット／ゲットアクセスが必要な場合、パスを構築するよりもオブジェクトの値を直接使用する方がシンプルで木の間いいことがあります。以下の２つの関数はそのようなアクセスを可能にするために用意されています。

NOTE これらのアクセサは `object!` 型だけでなく、関連付けされた他の配列型にも機能します。従って `map!` の値に対して使うことも可能です。

4.9.1. redSetField()

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

object の *field* に *value* をセットし、その *value* を返します。 *field* 引数は `redSymbol()` によって作られたシンボルIDです。

4.9.2. redGetField()

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

object の *field* に保持されている *value* を返します。 *field* 引数は `redSymbol()` によって作られたシンボルIDです。

4.10. デバッギング

いくつかの便利なデバッグ機能も提供されています。ほとんどは出力のためにシステムシェルウィンドウを必要としますが、ログウィンドウを強制的に開いたり、ファイルに出力をリダイレクトすることもできます。

4.10.1. redPrint()

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

標準出力か、もし開かれている場合はデバッグコンソールに *value* を出力します。

4.10.2. redProbe()

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

標準出力化、もし開かれている場合はデバッグコンソールに *value* をprobeした値を出力します。 *value* はこの関数呼び出しの戻り値になります。

4.10.3. redHasError()

```
red_value redHasError(void)
```

直前のAPI呼び出しでエラーが発生していれば error! の値を、エラーが起きていなければ `null` を返します。

4.10.4. redFormError()

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

エラーが起きていれば フォーマットされたエラーを含むUTF-8の文字列ポインタを、エラーが起きていなければ `null` を返します。

4.10.5. redOpenLogWindow()

```
int redOpenLogWindow(void)
```

ログウィンドウを開き、全てのRedのprint出力をそのウィンドウにリダイレクトします。この機能はホストアプリケーションがデフォルトのprint出力先であるシステムシェルから起動されていない場合に役に立ちます。すでにログウィンドウが開いている場合、この関数を複数回読んでも、何も起こりません。戻り値は成功すれば「1」、失敗した場合は「0」です。

> **NOTE** Windowsプラットフォームでのみ有効です。

4.10.6. redCloseLogWindow()

```
int redCloseLogWindow(void)
```

ログウィンドウを閉じます。ログウィンドウがすでに閉じている状況でこの関数を呼んでも何も起こりません。戻り値は成功すれば「1」、失敗した場合は「0」です。

> **NOTE** Windowsプラットフォームでのみ有効です。

4.10.7. redOpenLogFile()

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

name で指定されたファイルにRedのprint関数の出力をリダイレクトします。 *name* にはOSに応じたパス形式で、相対パス、仮想パスを指定することができます。

4.10.8. redCloseLogFile()

```
void redCloseLogFile(void)
```

`redOpenLogFile()` [illegible]

> **NOTE**
>
> [illegible]MS Office[illegible]

4.11. [illegible]

libRed API[illegible]Red[illegible] `redTypeOf`[illegible] `redMakeSeries()`[illegible]`redDataType()`[illegible]Red[illegible]`RedType`[illegible]

```
RED_TYPE_<DATATYPE>
```

[illegible] [illegible] [illegible]

5. Visual Basic API

VB[illegible]VBA[illegible]MS Office [illegible]Visual Basic API[illegible]C API[illegible]

- `redBlock()`, `redPath()`, `redCall()` [illegible]Red[illegible]C[illegible] `null` [illegible] `0` [illegible]
- `redBlockVB()`, `redPathVB()`, `redCallVB()` [illegible]VB[illegible]

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

5.1. [illegible]

VB[illegible]VBA[illegible]libRed[illegible]`stdcall` [illegible]ABI[illegible]libRed[illegible]

```
red build libRed stdcall
```

[illegible] `libRed.bas` [illegible]

5.2. redLogic()

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

VBの `boolean` の値から Redの `logic!` の値を生成して返します。

5.3. redBlockVB()

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

引数のリストから新しいblock!を生成して返します。引数の数は可変であり、VisualBasicの値だけで構成されている必要があります。

例

```
redProbe redBlockVB()               ' 空のブロックを生成します。
redProbe redBlockVB(42, "hello")    ' [42 "hello"] というブロックを生成します。
```

5.4. redPathVB()

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

引数のリストから新しいpath!のseriesを返します。引数の数は可変であり、VisualBasicの値だけで構成されている必要があります。

例

```
redDo("a: [b 123]")
res = redDo(redPathVB("a", "b"))
Debug.print redCInt32(res))         ' 123を出力します。
```

5.5. redCallVB()

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

第一引数として渡された文字列で参照される `any-function!` 型のRedの関数に、任意の数の引数を渡して実行します。引数の数は可変であり、VisualBasicの値である必要があります。

例

```
redCallVB("random", 6);             ' 1から6までのランダムなinteger!の値を返します。
```

5.6. コールバック関数の登録

Redの側から呼び出されることができるVisualBasicの関数を作る時は、C APIの場合と同様、 `redRoutine()`

を呼び出します。その関数の最後の引数は関数のポインタです。VBではそのようなポインタは *module* で定義された関数の場合のみ取得でき、 *UserForm* では取得できません。

以下はExcelの「Red Console」デモで使用されているコールバックです。

```
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
```