L’API LibRed

Table of Contents

1. Résumé

LibRed est une version spéciale de l'interpréteur et de la librairie d'exécution Red, adaptée à l'intégration dans un logiciel développé dans d'autres langages que Red. Afin de permettre au logiciel non-Red d'interagir avec Red, libRed propose une API bas-niveau dédiée (suivant les standards soit de C cdecl ou de Microsoft stdcall) qui est décrite dans ce document. Parmi les fonctionnalités supportées se trouvent:

- la capacité de définir/récupérer la valeur d'un word dans le contexte global ou un contexte local.
- des constructeurs abrégés pour les types de données Red les plus courants.
- des fonctions de conversion pour les types de données Red compatibles avec le langage hôte (principalement C).
- la manipulation de séries depuis le langage hôte.
- des callbacks permettant à Red d'appeler des fonctions du langage hôte.
- des fonctions de débogage orientées console.

Terminologie: le terme *hôte* est utilisé pour désigner le langage ou l'application hôte encapsulant libRed.

Des exemples d'utilisation de libRed peuvent être trouvés ici.

2. Construction de libRed

La construction de votre version locale de libRed est simple:

```
red build libRed
```

ou à partir de la console Rebol et des sources de Red:

```
rc "build libRed"
```

Ces lignes de commande construiront la version de libRed pour le langage C (en utilisant l'ABI `cdecl`). Si vous avez besoin de l'ABI `stdcall` (pour la compatibilité avec les applications Microsoft), vous devez utiliser:

```
red build libRed stdcall
```

3. Références aux valeurs

Des valeurs de Red peuvent être renvoyées par les appels de fonctions libRed. Elles sont représentées comme des références 32-bits *opaques*. Ces références ont une courte durée de vie, elles ne sont donc appropriées que pour un usage local restreint, comme passer cette référence à un autre appel de fonction libRed. Il est possible d'attribuer ces références à des variables de l'hôte, mais elles devraient être utilisées **immédiatement après**. Ces références utilisent une gestion de mémoire spécifique qui ne gardera les références en vie que pour les quelques 50 appels suivants de l'API. Par exemple:

```
long a, blk;

a = redSymbol("a");
redSet(a, redBlock(0));                    // la référence retournée est ici utilisée
immédiatement après

blk = redGet(a);
redPrint(blk);                             // usage sûr de la référence

for(i = 0; i < 100, i++) {
    // redAppend(blk, redNone());          // usage peu sûr de la référence!
    redAppend(redGet("a"), redNone());     // version sûre
}
```

4. L'API C

L'API C peut être utilisée avec des applications C/C++, mais aussi pour intégrer Red dans tout autre langage de programmation qui a une interface FFI compatible avec C.

4.1. Gestion de la librairie

Une *instance* de libRed doit être créée pour pouvoir utiliser toute fonction de l'API.

NOTE	Actuellement, une seule session libRed par processus est autorisée. Il est prévu d'étendre cela dans le futur pour permettre le support de multiples instances.

4.1.1. redOpen()

```
void redOpen(void)
```

Initialise une nouvelle session de la librairie d'exécution Red. Cette fonction *doit* être appelée avant l'appel de toute autre fonction de l'API. Il n'est pas gênant de l'appeler plusieurs fois dans le même process, une seule session sera ouverte de toute façon.

NOTE	Si une autre fonction est appelée avant `redOpen`, la valeur retournée par cette fonction sera `-2`, indiquant une tentative d'accès illégal.

4.1.2. redClose()

```
void redClose(void)
```

Met fin à la session en cours de la librairie d'exécution Red, en libérant toutes les ressources allouées.

4.2. Exécution de code Red

Le logiciel hôte peut exécuter du code Red directement, en utilisant un contrôle à différent niveaux, depuis le passage textuel de code Red devant être évalué, jusqu'à l'appel direct de toute fonction Red, et le passage d'arguments construits côté hôte.

4.2.1. redDo()

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

Evalue l'expression passée sous forme de chaîne et retourne la dernière valeur de Red.

Exemples

```
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)
```

Charge et évalue le script Red auquel *filename* fait référence et retourne la dernière valeur de Red. Le *filename* est formatté en utilisant les conventions indépendantes du système d'exploitation de Red (fondamentalement dans le style Unix).

Exemples

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

4.2.3. redDoBlock()

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

Evalue le bloc argument et retourne la dernière valeur de Red.

Exemple

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

4.2.4. redCall()

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

Invoque la fonction Red (de type `any-function!`) à laquelle le mot *name* fait référence, en lui passant les arguments éventuellement requis (comme valeurs Red). Renvoie la dernière valeur de la fonction Red. La liste d'arguments **doit** se terminer par une valeur `null` ou `0`, comme marqueur de fin.

Exemple

```
redCall(redWord("random"), redInteger(6), 0);     // renvoie une valeur integer!
aléatoire entre 1 et 6
```

4.3. Enregistrement d'une fonction de callback

La réponse à un événement se produisant dans Red, ou la redirection vers l'hôte de certains appels de Red (comme la redirection de `print` ou `ask`) requiert un moyen de rappeler une fonction de l'hôte depuis Red. Cela peut être accompli en utilisant la fonction `redRoutine()`.

4.3.1. redRoutine()

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

Définit une nouvelle routine Red nommée *name*, avec le bloc de spécifications *spec* et le pointeur de fonction C *func-ptr* comme corps. La fonction C **doit** renvoyer une valeur Red (`redUnset()` peut être employé pour indiquer que la valeur de renvoi est inutilisée).

Exemple

```
#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. Fabrication de valeurs Red depuis C

De nombreuses fonctions de l'API libRed requièrent le passage de valeurs Red (comme *références*). Les fonctions suivantes sont des constructeurs simples pour les types de données les plus couramment utilisés.

4.4.1. redSymbol()

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

Renvoie un ID de symbole associé avec le *word* entré (donné sous forme de chaîne C). Cet ID peut alors être passé à d'autres fonctions de l'API libRed qui requièrent un ID de symbole au lieu d'une valeur de word.

Exemple

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

4.4.2. redUnset()

```
red_unset redUnset(void)
```

Renvoie une valeur `unset!`.

4.4.3. redNone()

```
red_none redNone(void)
```

Renvoie une valeur `none!`.

4.4.4. redLogic()

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

Renvoie une valeur `logic!`. Une valeur *logic* de `0` donne la valeur `false`, toute autre valeur donne la valeur `true`.

4.4.5. redDatatype()

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

Renvoie une valeur `datatype!` correspondant à l'ID *type* qui est une valeur de l'énumération `RedType`.

4.4.6. redInteger()

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

Renvoie une valeur `integer!` à partir de *number*.

4.4.7. redFloat()

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

Renvoie une valeur `float!` à partir de *number*.

4.4.8. redPair()

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

Renvoie une valeur `pair!` à partir de deux valeurs entières.

4.4.9. redTuple()

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

Renvoie une valeur `tuple!` à partir de trois valeurs entières (habituellement pour représenter des couleurs RGB). Les arguments fournis seront tronqués à des valeurs sur 8 bits.

4.4.10. redTuple4()

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

Renvoie une valeur `tuple!` à partir de quatre valeurs entières (habituellement pour représenter des couleurs RGBA). Les arguments fournis seront tronqués à des valeurs sur 8 bits.

4.4.11. redBinary()

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

Renvoie une valeur `binary!` à partir d'un pointeur de mémoire `buffer` et la longueur du buffer en octets. Le buffer d'entrée sera copié en interne.

4.4.12. redImage()

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

Renvoie une valeur `image!` à partir d'un pointeur de mémoire `buffer`. La taille de l'image est définie en pixels par `width` et `height`. Le buffer d'entrée sera copié en interne. Les formats de buffer acceptés sont:

- `RED_IMAGE_FORMAT_RGB`: 24-bit par pixel.
- `RED_IMAGE_FORMAT_ARGB`: 32-bit pa pixel, le canal alpha en tête.

4.4.13. redString()

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

Renvoie une valeur `string!` à partir du pointeur *string*. L'encodage attendu par défaut pour la chaîne argument est UTF-8. D'autres encodages peuvent être spécifiés en utilisant la fonction `redSetEncoding()`.

4.4.14. redWord()

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

Renvoie une valeur `word!` à partir d'une chaîne C. L'encodage attendu par défaut pour la chaîne argument est UTF-8. D'autres encodages peuvent être spécifiés en utilisant la fonction `redSetEncoding()`. Les chaînes qui ne peuvent pas être interprétées comme des mots renverront une valeur `error!`.

4.4.15. redBlock()

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

Renvoie une nouvelle série de type `block!` construite à partir de la liste d'arguments. La liste **doit** se terminer par une valeur `null` ou `0`, comme marqueur de fin.

Exemples

```
redBlock(0);                                    // Crée un bloc vide
redBlock(redInteger(42), redWord("hi"), 0);   // Crée le bloc [42 hi]
```

4.4.16. redPath()

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

Renvoie une nouvelle série `path!` construite à partir de la liste d'arguments. La liste **doit** se terminer par une valeur `null` ou `0`, comme marqueur de fin.

Exemple

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

4.4.17. redLoadPath()

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

Renvoie une série `path!` construite à partir d'un chemin exprimé sous forme de chaîne de caractères C. Cela offre un moyen rapide de construire des chemins sans construire chaque élément individuellement.

Exemple

```
redGetPath(redLoadPath("a/b"));    // Crée et évalue la valeur de path! a/b.
```

4.4.18. redMakeSeries()

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

Renvoie une nouvelle `series!` de type *type* et de taille suffisante pour contenir *slots* éléments. Il s'agit d'une fonction générique de construction de séries. Le type doit être l'une des valeurs de l'énumération `RedType`.

Exemples

```
redMakeSeries(RED_TYPE_PAREN, 2);  // Crée une série de type paren!

long path = redMakeSeries(RED_TYPE_SET_PATH, 2); // Crée un set-path!
redAppend(path, redWord("a"));
redAppend(path, redInteger(2));    // Maintenant le chemin est `a/2:`
```

4.5. Fabrication de valeurs C depuis Red

La conversion de valeurs de Red vers le côté *hôte* est possible, elle est cependant restreinte par le nombre limité de types du langage C.

4.5.1. redCInt32()

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

Renvoie un entier signé sur 32 bits à partir d'une valeur `integer!` en Red.

4.5.2. redCDouble()

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

Renvoie une valeur en virgule flottante double précision à partir d'une valeur `float!` en Red.

4.5.3. redCString()

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

Renvoie un pointeur vers un buffer de chaîne de caractères UTF-8 à partir d'une valeur `string!` en Red. D'autres encodages peuvent être définis en utilisant la fonction `redSetEncoding()`.

4.5.4. redTypeOf()

```
long redTypeOf(red_value valeur)
```

Renvoie l'ID de type d'une valeur de Red. Les valeurs d'ID de type sont définies dans l'énumération `RedType`. Voir la section Types de Données.

4.6. Appel d'une action Red

Il est possible d'appeler n'importe quelle fonction Red en utilisant `redCall`, cependant pour les actions les plus communes, certains raccourcis sont fournis pour la facilité d'usage et de meilleures performances.

4.6.1. redAppend()

```
red_value redAppend(red_series série, red_value valeur)
```

Ajoute une *valeur* à une *série* et renvoie la série au début.

4.6.2. redChange()

```
red_value redChange(red_series série, red_value valeur)
```

Change une *valeur* dans une *série* et renvoie la série après la partie changée.

4.6.3. redClear()

```
red_value redClear(red_series série)
```

Supprime les valeurs de la *série* de la position courante jusqu'à la fin et retourne la série à la nouvelle fin.

4.6.4. redCopy()

```
red_value redCopy(red_value valeur)
```

Renvoie une copie d'une valeur non-scalaire.

4.6.5. redFind()

```
red_value redFind(red_series série, red_value valeur)
```

Renvoie la *série* à la position où la *valeur* est trouvée, sinon `none`.

4.6.6. redIndex()

```
red_value redIndex(red_series série)
```

Renvoie la position courante de la *série* par rapport au début, ou d'un mot dans un contexte.

4.6.7. redLength()

```
red_value redLength(red_series série)
```

Renvoie le nombre de valeurs dans la *série*, de la position courante jusqu'à la fin.

4.6.8. redMake()

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

Renvoie une nouvelle valeur fabriquée à partir des *spec* pour le type de ce *proto*.

4.6.9. redMold()

```
red_value redMold(red_value valeur)
```

Renvoie une chaîne au format source représentant cette valeur.

4.6.10. redPick()

```
red_value redPick(red_series série, red_value valeur)
```

Renvoie la *valeur* de la *série* à la position donnée.

4.6.11. redPoke()

```
red_value redPoke(red_series série, red_value index, red_value valeur)
```

Remplace par *valeur* la valeur de la *série* à la position donnée, et renvoie la nouvelle valeur.

4.6.12. redPut()

```
red_value redPut(red_series série, red_value index, red_value valeur)
```

Remplace la valeur suivant une clé dans une *série* ou une `map!`, et renvoie la nouvelle valeur.

4.6.13. redRemove()

```
red_value redRemove(red_series série)
```

Supprime une valeur à la position courante de la *série* et renvoie la série après suppression.

4.6.14. redSelect()

```
red_value redSelect(red_series série, red_value valeur)
```

Cherche une *valeur* dans une *série* et retourne la valeur suivante, ou `none`.

4.6.15. redSkip()

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

Renvoie la *série* à une position décalée de *offset* relativement à la position courante.

4.6.16. redTo()

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

Convertit la valeur *spec* à un type de données spécifié par *proto*.

4.7. Accès à un mot Red

La définition ou la lecture de la valeur d'un mot (word) Red est la manière la plus directe de passer des valeurs entre l' *hôte* et l'environnement d'exécution de Red.

4.7.1. redSet()

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

Attribue la *valeur* au mot Red défini par le symbole *id* . Le mot créé à partir du symbole est lié au contexte global. Cette fonction renvoie la *valeur* .

4.7.2. redGet()

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

Renvoie la *valeur* d'un mot Red défini par le symbole *id* . Le mot créé à partir du symbole est lié au contexte global.

4.8. Accès à un chemin Red

Les chemins sont une manière très flexible d'accéder aux données en Red, et ils ont donc leurs fonctions accesseurs dédiées en libRed. En particulier, ils permettent l'accès à des mots dans des contextes d'objets.

4.8.1. redSetPath()

```
red_value redSetPath(red_path chemin, red_value valeur)
```

Donne une *valeur* à un *chemin* et renvoie cette *valeur*.

4.8.2. redGetPath()

```
red_value redGetPath(red_path chemin)
```

Renvoie la *valeur* référencée par le *chemin*.

4.9. Accès à un champ d'un objet Red

Lorsqu'on a besoin de multiples accès en écriture/lecture aux champs d'un objet, il est plus simple et préférable d'utiliser directement la valeur de l'objet plutôt que de construire des chemins. Les deux fonctions suivantes sont conçues pour de tels accès.

NOTE	Ces accesseurs peuvent fonctionner avec tous les autres types de tableaux associatifs, pas seulement `object!`. Ainsi on peut aussi leur passer une `map!`.

4.9.1. redSetField()

```
red_value redSetField(red_value objet, long champ, red_value valeur)
```

Donne une *valeur* au *champ* de l'*objet* et renvoie cette *valeur*. L'argument *champ* est un ID de symbole créé en utilisant `redSymbol()`.

4.9.2. redGetField()

```
red_value redGetField(red_value objet, long champ)
```

Renvoie la *valeur* stockée dans le *champ* de l'*objet*. L'argument *champ* est un ID de symbole créé en utilisant `redSymbol()`.

4.10. Débogage

Des fonctions de débogage pratiques sont aussi disponibles. La plupart d'entre elles requiert une fenêtre de terminal système pour la sortie, cependant il est possible de forcer l'ouverture d'une fenêtre de log, ou de rediriger la sortie vers un fichier.

4.10.1. redPrint()

```
void redPrint(red_value valeur)
```

Affiche la *valeur* sur la sortie standard, ou sur la console de débogage si elle est ouverte.

4.10.2. redProbe()

```
red_value redProbe(red_value valeur)
```

Affiche la description (probe) de la *valeur* sur la sortie standard, ou sur la console de débogage si elle est ouverte. Cet appel de fonction retourne la *valeur*.

4.10.3. redHasError()

```
red_value redHasError(void)
```

Renvoie une valeur `error!` si une erreur s'est produite dans le dernier appel à l'API, ou `null` si aucune erreur ne s'est produite.

4.10.4. redFormError()

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

Renvoie un pointeur de chaîne de caractères UTF-8 contenant une erreur formatée si une erreur s'est produite, ou `null` si aucune erreur ne s'est produite.

4.10.5. redOpenLogWindow()

```
int redOpenLogWindow(void)
```

Ouvre la fenêtre de log et redirige toutes les sorties d'affichage de Red vers cette fenêtre. Cette fonctionnalité est utile si l'application hôte n'est pas lancée depuis le terminal système, qui est utilisé par défaut pour la sortie d'affichage. Appeler cette fonction plusieurs fois sera sans effet si la fenêtre de log est déjà ouverte. Elle renvoie `1` en cas de succès, `0` en cas d'échec.

NOTE Uniquement pour les plateformes Windows.

4.10.6. redCloseLogWindow()

```
int redCloseLogWindow(void)
```

Ferme la fenêtre de log. Appeler cette fonction lorsque la fenêtre de log est déjà fermée sera sans effet. Renvoie `1` en cas de succès, `0` en cas d'échec.

NOTE	Uniquement pour les plateformes Windows.

4.10.7. redOpenLogFile()

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

Redirige la sortie des fonctions d'affichage de Red vers le fichier spécifié par *nom*. Un chemin relatif ou absolu peut être passé dans *nom* en utilisant le format de chemin d'accès aux fichiers spécifique au système d'exploitation.

4.10.8. redCloseLogFile()

```
void redCloseLogFile(void)
```

Ferme le fichier de log ouvert avec `redOpenLogFile()`.

NOTE	Actuellement, le fichier de log **doit** être fermé à la fin de l'exécution, sans quoi un verrou reste posé dessus et cela peut geler voire faire planter certains hôtes (comme les applications MS Office).

4.11. Définitions de types de données

Certaines fonctions de l'API libRed peuvent se référer à des types de données Red: `redTypeOf()`, `redMakeSeries()` et `redDatatype()`. Les types de données Red sont représentés côté hôte, par une énumération (`RedType`), où les types sont des noms qui suivent le format suivant:

```
RED_TYPE_<DATATYPE>
```

On peut trouver la liste exhaustive ici.

5. API Visual Basic

L'API Visual Basic peut être utilisée aussi bien pour le VB que pour le VBA (des applications MS Office). Elle est essentiellement identique à l'API C, donc seules les différences seront décrites dans les sections ci-dessous. Les différences sont principalement dans les fonctions variadiques, qui sont de deux types:

- `redBlock()`, `redPath()`, `redCall()` n'acceptent que des valeurs Red, et ne requièrent pas une valeur finale `null` ou `0`, comme la version C.
- `redBlockVB()`, `redPathVB()`, `redCallVB()` n'acceptent que des valeurs VB, qui sont

automatiquement converties suivant la table suivante:

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

5.1. Mise en place

Afin d'utiliser libRed avec VB/VBA, vous avez besoin d'une version de l'exécutable libRed qui est compilée pour l'ABI `stdcall`. Si vous devez recompiler une telle version:

```
red build libRed stdcall
```

Vous aurez aussi besoin d'importer le fichier du module `libRed.bas` dans votre projet.

5.2. redLogic()

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

Renvoie une valeur Red `logic!` construite à partir d'une valeur VB `boolean`.

5.3. redBlockVB()

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

Renvoie une nouvelle série `block!` construite à partir de la liste d'arguments. Le nombre d'arguments est variable et composé uniquement de valeurs VisualBasic.

Exemples

```
redProbe redBlockVB()              ' Crée un bloc vide
redProbe redBlockVB(42, "hello")   ' Crrée le bloc [42 "hello"]
```

5.4. redPathVB()

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

Renvoie une nouvelle série `path!` construite à partir de la liste d'arguments. Le nombre d'arguments est variable et composé uniquement de valeurs VisualBasic.

Exemples

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

5.5. redCallVB()

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

Invoque la fonction Red (de type `any-function!`) référencée par la chaîne de caractères passée (en premier argument), lui passant éventuellement des arguments (en tant que valeurs VisualBasic). Renvoie la dernière valeur de la fonction. Le nombre d'arguments est variable et composé uniquement de valeurs VisualBasic.

Exemple

```
redCallVB("random", 6);           renvoie une valeur aléatoire d' integer! entre 1 et
6
```

5.6. Enregistrement d'une fonction de callback

La création d'une fonction VisualBasic qui peut être appelée depuis le côté Red se fait comme dans l'API C, en utilisant l'appel `redRoutine()`. Le dernier argument de cette fonction est un pointeur de fonction. En VB, un tel pointeur ne peut être acquis que pour une fonction définie dans un *module*, mais pas dans une *UserForm*.

Voici la fonction de callback utilisée dans la démo Excel "Red Console":

```
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
```