Redbin format

Table of Contents

1. Redbin codec . . . 2
2. Lexical conventions . . . 3
3. Encoding format . . . 3
4. Header . . . 4
5. Symbol table . . . 4
6. Record definitions . . . 5
    6.1. Special . . . 6
        6.1.1. Padding . . . 6
        6.1.2. Reference . . . 6
    6.2. Unsupported . . . 7
    6.3. Datatypes . . . 8
        6.3.1. `datatype!` . . . 8
        6.3.2. `unset!` . . . 8
        6.3.3. `none!` . . . 8
        6.3.4. `logic!` . . . 9
        6.3.5. `block!` . . . 9
        6.3.6. `paren!` . . . 9
        6.3.7. `string!` . . . 9
        6.3.8. `file!` . . . 10
        6.3.9. `url!` . . . 10
        6.3.10. `char!` . . . 10
        6.3.11. `integer!` . . . 10
        6.3.12. `float!` . . . 11
        6.3.13. `context!` . . . 11
        6.3.14. `word!` . . . 11
        6.3.15. `set-word!` . . . 12
        6.3.16. `lit-word!` . . . 12
        6.3.17. `get-word!` . . . 12
        6.3.18. `refinement!` . . . 13
        6.3.19. `issue!` . . . 13
        6.3.20. `native!` . . . 13
        6.3.21. `action!` . . . 13
        6.3.22. `op!` . . . 14
        6.3.23. `function!` . . . 14
        6.3.24. `path!` . . . 14
        6.3.25. `lit-path!` . . . 15

6.3.26. set-path! . . . 15
6.3.27. get-path! . . . 15
6.3.28. bitset! . . . 15
6.3.29. object! . . . 16
6.3.30. typeset! . . . 16
6.3.31. error! . . . 16
6.3.32. vector! . . . 17
6.3.33. pair! . . . 17
6.3.34. percent! . . . 17
6.3.35. tuple! . . . 18
6.3.36. map! . . . 18
6.3.37. binary! . . . 18
6.3.38. time! . . . 18
6.3.39. tag! . . . 18
6.3.40. email! . . . 19
6.3.41. date! . . . 19
6.3.42. money! . . . 19
6.3.43. ref! . . . 20
6.3.44. image! . . . 20

NOTE Specification version 2.

Redbin is a binary format that accurately represents Red values stored in memory while enabling fast loading (avoiding the parsing and validation stage of the text representation format). Redbin format is largely inspired by REBin; it can encode binding information for `any-word!` values, references to shared buffers for `series!` values, and can handle arbitrary cycles for `any-block!` values.

1. Redbin codec

Redbin format enables use-cases typically associated with data serialization, such as:

- Persistent state and image-based environments;
- Remote procedure calls;
- Sharing data and programs over the network with other systems;
- Detecting changes in time-varying data.

To make writing applications based on these ideas and methods possible, an interface to Redbin format is provided via the codec sub-system. Redbin codec can be accessed with `save` and `load` functions as described below.

Syntax

```
save/as <where> <value> 'redbin
save <file> <value>

load/as <data> 'redbin
load <file>

<data>  : Redbin-encoded binary! value
<value> : value of any supported datatype
<where> : saving destination for encoded data; file!, url!, string!, binary!, none!
<file>  : file! with .redbin extension
```

NOTE	Redbin codec cannot encode or decode unsupported values or, by extension, values that contain them.

WARNING	Attempt to `load` Redbin data with malformed payload will likely lead to unexpected results or a runtime crash.

2. Lexical conventions

Several semi-formal lexical conventions are used throughout this document to describe the Redbin encoding format:

- Numbers in parentheses indicate field's size;
- Field followed by an equal sign has a fixed content;
- Field followed by a name of a record type enclosed in square brackets indicates an encoded Redbin record of that type;
- Ellipsis stands for a generic Redbin value record of any datatype;
- Pipe symbol indicates a choice between alternatives;
- Multiplication sign indicates a repetition;
- Path notation is used to refer to record's header flags.

3. Encoding format

The *default* encoding format is optimized for decoding speed, while the *compact* format requires a smaller storage space (at the expense of much slower decoding).

NOTE	Specification of the compact encoding format is not yet defined.

The general layout of Redbin data is described below. Each definition links to a respective section in this document.

Redbin header

Holds information about the rest of the Redbin data.

Symbol table

Optional; if present, contains interned strings used by records of symbolic datatypes.

Payload

Stores Redbin records that encode Red values.

Data in these sections is stored in a little-endian format. All integer fields represent non-negative values, but since Red runtime interprets them as signed, they have an upper limit of 2^{31}-1.

4. Header

Redbin data starts with a header having the following format:

```
magic="REDBIN" (6), version=1|2 (1), flags (1), length (4), size (4)

length : number of root records to load.
size   : the size of records payload in bytes.
```

The layout of `flags` field is described in the table below.

Table 1. Redbin header flags.

Bits	Description
7-3	Reserved for future use.
2	If set, indicates that Redbin data contains a symbol table.
1	If set, indicates that data immediately following the `flags` field is compressed. The compression algorithm is implementation-dependent.
0	If set, indicates that the records section is encoded using the compact format.

The header is the only mandatory section in Redbin format encoding; both symbol table and payload can be omitted, provided that relevant flags and fields a properly specified.

5. Symbol table

The symbol table immediately follows the header data. It is optional and should only be used if `any-word!` values are present in the Redbin payload. The symbol table has two sections:

Offsets table

A list of offsets to a string representation of each symbol inside the strings buffer;

Strings buffer

Immediately follows offsets table; contains UTF-8 encoded, NUL-terminated strings concatenated to each other, with an optional 64-bit boundary padding at the end of each string.

The position of an offset in the table is its *index* (zero-based), which is used as a reference by

symbols in `context!` and `any-word!` records. The offsets in the table are offsets in bytes from the beginning of the strings buffers section to the referred string.

Table of offsets encoding is described below:

```
Default: length (4), size (4), offset (4) * length
Compact: TBD
```

`length` field contains the number of entries in the table. `size` field indicates the size of the strings buffer in bytes (including optional padding).

During the runtime booting process, these symbols are merged with Red's symbol table and the offsets are replaced by the symbol ID values from that table. Runtime codec omits this merging stage and instantiates symbols in-place for each relevant decoded record.

After the symbol table, Red values are stored as a sequence of records, with no special delimiters or end markers. The loaded values from the root level are stored in a `block!` series.

6. Record definitions

Each record in Redbin payload starts with a 32-bit `header` field defined as:

Table 2. Layout of Redbin record header.

Bits	Description	Relevant datatypes
31	`new-line` flag; if set, indicates the presence of a new-line flag in value slot.	All.
30	`no-values` flag; if set, indicates that `context!` record does not contain value records.	`context!`
29	`stack?` flag; if set, indicates that values of decoded `context!` are allocated on the data stack rather than on the heap memory.	`context!`
28	`self?` flag; if set, indicates that decoded `context!` is capable of self-referencing via `self` word.	`context!`
27-26	`kind` field; encodes `context!` type.	`context!`
25	`set?` flag; if set, indicates that `any-word!` record is followed by value record to which decoded `any-word!` needs to be set.	`any-word!`
24	`owner?` flag; if set, indicates that decoded `object!` owns one or more values.	`object!`

Bits	Description	Relevant datatypes
23	`native?` flag; if set, indicates that decoded `op!` value is derived from `native!`, else from `action!`.	`op!`
22	`body?` flag; if set, indicates that `op!` value is derived from either `function!` or `routine!` and has a body block.	`op!`
21	`complement?` flag; if set, indicates that decoded `bitset!` value is complemented.	`bitset!`
20	`sign` flag; if set, indicates that decoded `money!` value has a negative sign.	`money!`
19	`reference?` flag; if set, indicates that Redbin record contains a reference.	See Reference section.
18-16	Reserved for future use.	—
15-8	`unit` field; encodes element size (i.e. unit) in a series buffer.	`series!`
7-0	`type` field; encodes value type.	All.

Here follow individual descriptions of each type of record.

6.1. Special

Some types of Redbin records do not correspond to any Red value datatype and are described in this section.

6.1.1. Padding

```
Default: header (4)
Compact: N/A

header/type=0
```

This empty record is used to properly align 64-bit values.

6.1.2. Reference

```
Default: header (4), length (4), offset (4) * length
Compact: TBD

header/type=255
```

Reference records are used to encode various relations between Red values, such as `any-word!`

bindings and shared `series!` buffers.

`length` field specifies the number of `offset` fields contained inside a reference record; each `offset` field specifies a zero-based offset to an already loaded Red value thru its parent, starting from the root block. A list of such offsets effectively forms a path to a referenced value.

Red value that is used as a parent to calculate offset into is called a *waypoint*; Red value to which the path is formed by a reference is called a *target*. Reference records are usually used by other value records to obtain datatype-specific parts that they share with the target. Red value record that contains a reference is called a *referral*. In all record definitions that follow, referral format is used to describe such a form of encoding, which is used only when `reference?` header flag of a respective value record is set.

Redbin records that can act as referrals are: `series!`, `map!`, `bitset!`, `any-word!`, `refinement!`, `object!`, `function!`.

Only a selected number of datatypes can be a waypoint or a target, and rules of offset calculation and referencing for each of them are described in the table below.

Table 3. Datatypes thru and to which reference paths can be formed.

Datatypes	Waypoint	Target
`any-block!`, `map!`	An offset from the series' head. `map!` is treated as a linear block.	Series buffer is reused.
`any-string!`, `binary!`, `bitset!`, `vector!`, `image!`	—	Series buffer is reused.
`action!`, `native!`	Offset from the head of the spec block.	Spec buffer is reused.
`object!`	An offset from the head of the values block.	Binding information is reused.
`any-word!`, `refinement!`	An offset into a context to which value is bound, which is represented as either `object!` or `function!` value.	Binding information is reused.
`function!`	Offset of value `0` selects a spec block, offset of value `1` selects a body block. Other offset values are forbidden.	Binding information is reused.
`op!`	Offset of value `0` selects a spec block. Other offset values are forbidden.	Binding information of `function!` value from which `op!` is derived is reused.

A referral can target its parent, in such case a cycle is formed.

6.2. Unsupported

Some Red value datatypes (listed below) are not supported by Redbin format.

Table 4. Red datatypes not supported by Redbin format.

Datatypes	Reason
`routine!`, `op!` derived from `routine!`	Contains a direct pointer to machine code.
`handle!`	Contains a reference to session-specific and OS-specific system resource.
`event!`	Contains a direct pointer to session-specific and OS-specific system resource.

A list of additional limitations follows below:

- Pre-compiled functions can be encoded, but on decoding start to behave as interpreted;
- Object's `self` keyword cannot be encoded in some cases.

6.3. Datatypes

This section describes the encoding of Redbin records that correspond to Red value datatypes.

6.3.1. `datatype!`

```
Default: header (4), value (4)
Compact: TBD

header/type=1
```

`value` field contains datatype ID represented as a 32-bit integer.

6.3.2. `unset!`

```
Default: header (4)
Compact: TBD

header/type=2
```

`unset!` is a singleton value and can be encoded as a `header` field with datatype ID.

6.3.3. `none!`

```
Default: header (4)
Compact: TBD

header/type=3
```

`none!` is a singleton value and can be encoded as a `header` field with datatype ID.

6.3.4. logic!

```
Default: header (4), value=0|1 (4)
Compact: TBD

header/type=4
```

value content of 0 encodes a false value. Non-zero value content encodes a true value.

6.3.5. block!

```
Default:  header (4), head (4), length (4), ... * length
Referral: header (4), head (4), buffer [reference]
Compact:  TBD

header/type=5
header/reference?=0|1
```

The head field indicates a zero-based offset of the index position from the block's head. The length field contains the number of values to be stored in the block. The block values' records then follow the length field.

6.3.6. paren!

```
Default:  header (4), head (4), length (4), ... * length
Referral: header (4), head (4), buffer [reference]
Compact:  TBD

header/type=6
header/reference?=0|1
```

Same encoding rules as block!.

6.3.7. string!

```
Default:  header (4), head (4), length (4), data (unit * length), padding (1-3)
Referral: header (4), head (4), buffer [reference]
Compact:  TBD

header/type=7
header/unit=1|2|4
header/reference?=0|1
```

The head field has the same meaning as in other series records. The unit field indicates the encoding format of the string, only values of 1, 2, and 4 are valid. The length field contains the number of

codepoints to be stored in the string, up to 16777215 codepoints (2^{24} - 1) are supported. The string is encoded in either UCS-1, UCS-2 or UCS-4 format, depending on the maximum width of contained codepoints. No NUL-terminating character is present in `data`, nor accounted for in the `length` field. An optional tail padding of 1 to 3 NUL bytes can be present to align the end of the `string!` record with a 32-bit boundary.

6.3.8. `file!`

```
Default:  header (4), head (4), length (4), data (unit * length), padding (1-3)
Referral: header (4), head (4), buffer [reference]
Compact:  TBD

header/type=8
header/unit=1|2|4
header/reference?=0|1
```

Same encoding rules as `string!`.

6.3.9. `url!`

```
Default:  header (4), head (4), length (4), data (unit * length), padding (1-3)
Referral: header (4), head (4), buffer [reference]
Compact:  TBD

header/type=9
header/unit=1|2|4
header/reference?=0|1
```

Same encoding rules as `string!`.

6.3.10. `char!`

```
Default: header (4), value (4)
Compact: TBD

header/type=10
```

`value` field contains a UCS-4 codepoint stored as a 32-bit integer.

6.3.11. `integer!`

```
Default: header (4), value (4)
Compact: TBD

header/type=11
```

`value` field contains a signed 32-bit integer that encoded Red value represents.

6.3.12. `float!`

```
Default: padding [padding], header (4), value (8)
Compact: TBD

header/type=12
```

An optional `padding` field is added to properly align the `value` field at a 64-bit boundary. `value` field itself contains a 64-bit IEEE 754 floating-point numeral.

6.3.13. `context!`

```
Default: header (4), length (4), symbol (4) * length, ... * length
Compact: TBD

header/type=14
header/kind=0|1|2
header/no-values=0|1
header/stack?=0|1
header/self?=0|1
```

Contexts are Red values used internally by some datatypes like `function!`, `object!` and derivative types. A context record contains two consecutive lists, the first one is a list of word entries in the context represented as `symbol` references, the second one is the associated value records for each of the symbols in the first list.

`kind` field in record's header encodes context's type: `0` for global context, `1` for the context of a function, and `2` for the context of an object. The global context is never encoded explicitly, which means that only values of `1` and `2` are used. `length` field indicates the number of entries in the context.

If `no-values` flag is set, it means that there are no value records following the symbols (empty context). If `stack?` flag is set, then the values are allocated on the stack instead of the heap memory. The `self?` flag is used to indicate that the context can handle a self-referencing word (`self` in objects).

6.3.14. `word!`

```
Default:  header (4), symbol (4), index (4), ...|context [object!|function!]
Referral: header (4), symbol (4), index (4), context [reference]
Compact:  TBD

header/type=15
header/set?=0|1
header/reference?=0|1
```

`symbol` field is an index in Redbin symbol table. `index` is word's index in the context to which it is bound. If `set?` flag is set, then word is bound to a global context and `index` field is followed by a value record to which word needs to be set; otherwise `index` field is followed by either `object!` or `function!` record that contains context to which word needs to be bound.

NOTE	In the current implementation, enabled `set?` flag indicates that word is bound to a global context, but value record is omitted.

6.3.15. `set-word!`

```
Default:  header (4), symbol (4), index (4), ...|context [object!|function!]
Referral: header (4), symbol (4), index (4), context [reference]
Compact:  TBD

header/type=16
header/set?=0|1
header/reference?=0|1
```

Same encoding rules as `word!`.

6.3.16. `lit-word!`

```
Default:  header (4), symbol (4), index (4), ...|context [object!|function!]
Referral: header (4), symbol (4), index (4), context [reference]
Compact:  TBD

header/type=17
header/set?=0|1
header/reference?=0|1
```

Same encoding rules as `word!`.

6.3.17. `get-word!`

```
Default:  header (4), symbol (4), index (4), ...|context [object!|function!]
Referral: header (4), symbol (4), index (4), context [reference]
Compact:  TBD

header/type=18
header/set?=0|1
header/reference?=0|1
```

Same encoding rules as `word!`.

6.3.18. `refinement!`

```
Default:  header (4), symbol (4), index (4), ...|context [object!|function!]
Referral: header (4), symbol (4), index (4), context [reference]
Compact:  TBD

header/type=19
header/set?=0|1
header/reference?=0|1
```

Same encoding rules as `word!`.

6.3.19. `issue!`

```
Default: header (4), symbol (4)
Compact: TBD

header/type=20
```

`symbol` field is an index in Redbin symbol table.

6.3.20. `native!`

```
Default: header (4), ID (4), spec [block!]
Compact: TBD

header/type=21
```

`ID` field is an offset into the internal `natives/table` jump table, followed by a `block!` record encoding native's spec.

6.3.21. `action!`

```
Default: header (4), ID (4), spec [block!]
Compact: TBD

header/type=22
```

ID field is an offset into the internal actions/table jump table, followed by a block! record encoding action's spec.

6.3.22. op!

```
Default: header (4), parent [function!]|spec [block!], ID (4)
Compact: TBD

header/type=23
header/body?=0|1
neader/native?=0|1
```

If body? flag is set, it indicates that op! is derived from a function!; if body? flag is not set, then op! is derived from either action! or native! — the choice between the two is indicated by native? flag.

If body? flag is set, then header field is followed by a function! record that encodes op! value's parent. Otherwise, it is followed by a block! record encoding op! value's spec, and then by an ID of either action! or native! value.

6.3.23. function!

```
Default:  header (4), spec-size (4), body-size (4), context [context!], spec [block!],
body [block!]
Referral: header (4), context [reference]
Compact:  TBD

header/type=24
header/reference?=0|1
```

spec-size and body-size specify sizes of spec and body blocks, respectively, and are used for pre-allocation by the decoder.

The target of the reference is either function!, op!, or any-word!; function! value (loaded value, parent of op!, or context of any-word!) is copied over verbatim, which means that referral shares with it not only binding information, but also spec and body blocks.

6.3.24. path!

```
Default:  header (4), head (4), length (4), ... * length
Referral: header (4), head (4), buffer [reference]
Compact:  TBD

header/type=25
header/reference?=0|1
```

Same encoding rules as `block!`.

6.3.25. `lit-path!`

```
Default:  header (4), head (4), length (4), ... * length
Referral: header (4), head (4), buffer [reference]
Compact:  TBD

header/type=26
header/reference?=0|1
```

Same encoding rules as `block!`.

6.3.26. `set-path!`

```
Default:  header (4), head (4), length (4), ... * length
Referral: header (4), head (4), buffer [reference]
Compact:  TBD

header/type=27
header/reference?=0|1
```

Same encoding rules as `block!`.

6.3.27. `get-path!`

```
Default:  header (4), head (4), length (4), ... * length
Referral: header (4), head (4), buffer [reference]
Compact:  TBD

header/type=28
header/reference?=0|1
```

Same encoding rules as `block!`.

6.3.28. `bitset!`

```
Default:  header (4), length (4), data (length), padding (1-3)
Referral: header (4), buffer [reference]
Compact:  TBD

header/type=30
header/complement?=0|1
```

If `complement?` flag is set, it indicates that bitset is complemented. The `length` field encodes the number of bytes stored. `data` is a memory dump of `bitset!` series buffer, byte order is preserved. `data` field needs to be padded with enough NUL bytes to keep the next record aligned at a 32-bit boundary.

6.3.29. `object!`

```
Default:  header (4), class (4), on-set (4), arity (4), context [context!]
Referral: header (4), context [reference]
Compact:  TBD

header/type=32
header/owner?=0|1
header/reference?=0|1
```

`class` field stores object's class ID. `on-set` field is a pair of 16-bit integers, each of which encodes an offset to `on-change*` and `on-deep-change*` function in object's values block. `arity` field has the same format as `on-set`, but encodes arities of the respective functions. These two fields are optional and are encoded only if `owner?` flag is set in record's header.

6.3.30. `typeset!`

```
Default: header (4), array1 (4), array2 (4), array3 (4)
Compact: TBD

header/type=33
```

`array1`, `array2`, and `array3` fields form a bitset where an index of each `1` bit indicates a datatype ID contained inside a typeset.

6.3.31. `error!`

```
Default: header (4), code (4), ... * 6
Compact: TBD

header/type=34
```

`code` field encodes error's identifier and is followed by 6 value records for error's fields: `arg1`, `arg2`,

arg3, near, where, stack.

6.3.32. vector!

```
Default:  header (4), head (4), length (4), type (4), data (unit * length), padding
(1-3)
Referral: header (4), head (4), buffer [reference]
Compact:  TBD

header/type=35
header/unit=1|2|4|8
```

type field contains datatype ID of vector's element. unit field indicates the size of the vector element's type size in bytes. Only the following combinations of type and unit values are supported:

Table 5. Combinations of vector! fields.

Type	Unit
char!, integer!	1, 2, 4
float!	4, 8
percent!	8

The data field holds a list of values. If unit is equal to 1 or 2, data needs to be padded with NUL bytes up to a 32-bit boundary.

6.3.33. pair!

```
Default: header (4), x (4), y (4)
Compact: TBD

header/type=37
```

x and y fields encode the respective pair's elements as 32-bit integers.

6.3.34. percent!

```
Default: padding [padding], header (4), value (8)
Compact: TBD

header/type=38
```

Same encoding rules as float!.

6.3.35. tuple!

```
Default: header (4), array1 (4), array2 (4), array3 (4)
Compact: TBD

header/type=39
header/unit=3-12
```

`unit` field encodes tuple's size in bytes; only values from `3` to `12` are allowed. `array1`, `array2`, and `array3` fields together form a memory dump of tuple's slot payload.

6.3.36. map!

```
Default:  header (4), length (4), ... * length
Referral: header (4), buffer [reference]
Compact:  TBD

header/type=40
header/reference?=0|1
```

The `length` field contains the number of elements (both keys and values) encoded.

6.3.37. binary!

```
Default:  header (4), head (4), length (4), data (length)
Referral: header (4), head (4), buffer [reference]
Compact:  TBD

header/type=41
header/reference?=0|1
```

`data` field contains memory dump of binary's series buffer, byte order is preserved.

6.3.38. time!

```
Default: padding [padding], header (4), value (8)
Compact: TBD

header/type=43
```

Same encoding rules as `float!`.

6.3.39. tag!

```
Default:  header (4), head (4), length (4), data (unit * length), padding (1-3)
Referral: header (4), head (4), buffer [reference]
Compact:  TBD

header/type=44
header/unit=1|2|4
header/reference?=0|1
```

Same encoding rules as `string!`.

6.3.40. `email!`

```
Default:  header (4), head (4), length (4), data (unit * length), padding (1-3)
Referral: header (4), head (4), buffer [reference]
Compact:  TBD

header/type=45
header/unit=1|2|4
header/reference?=0|1
```

Same encoding rules as `string!`.

6.3.41. `date!`

```
Default: header (4), date (4), time (8)
Compact: TBD

header/type=47
```

`date` field contains date value packed into a 32-bit integer. The following format is used (field sizes are in bits):

```
year (15), time? (1), month (4), day (5), timezone (7)
```

`year` and `timezone` sub-fields contain signed values. `time` field stores time value as a 64-bit float.

6.3.42. `money!`

```
Default: header (4), currency (1), amount (11)
Compact: TBD

header/type=49
header/sign=0|1
```

<code>amount</code> field is a sequence of nibbles representing the base (17) and subunit (5) of money value, byte order is preserved. If <code>sign</code> flag is set, the amount is interpreted as negative. <code>currency</code> field is an integer value representing currency ID (0 for generic money, ≤ 255 for existing currency code).

6.3.43. `ref!`

```
Default:  header (4), head (4), length (4), data (unit * length), padding (1-3)
Referral: header (4), head (4), buffer [reference]
Compact:  TBD

header/type=50
header/unit=1|2|4
header/reference?=0|1
```

Same encoding rules as `string!`.

6.3.44. `image!`

```
Default:  header (4), head (4), length (4), rgba (4 * width * height)
Referral: header (4), head (4), buffer [reference]
Compact:  TBD

header/type=51
header/reference?=0|1
```

`length` field is a pair of 16-bit integers encoding width and height of an image. `rgba` field contains RGBA content of an image (4 bytes per pixel) with preserved byte order.