BRBON

V 0.4

A Binary Object Notation by Balancing Rock.

Introduction

BRBON is a binary storage format specification. It started out as a binary version for JSON but the requirement for speed has rendered some JSON aspects obsolete. Still, the JSON origins can be recognised in the object oriented approach.

The major design driver for BRBON is speed. Speed in access and speed in loading & storage. A secondary goal was predictable speed. BRBON was originally designed for use in a web server application and thus the access times should be constant and not fluctuate widely.

Strictly speaking BRBON only refers to the format specification, not an implementation. However a reference API is offered for free under the same name.

- Vectored access: The format includes vectors that can be used to quickly traverse the entire structure (in both directions).
- Named objects: Objects may be named for identification, a hash value is included for faster name recognition.
- Aligned to 8 byte boundaries: Most of the elements in the structure are aligned to 8 byte boundaries. This ensures alignment for al types.
- All standard types are supported: Bool, Int8, Int16, Int32, Int64, UInt8, UInt16, UInt32, UInt64, Float32, Float64, String, Binary, UUID
- Collection types: Array, Dictionary, Sequence, Table.
- Special type: Null.
- Composite types: RGBA (color), Font
- Array's are packed for optimal storage density.
- The maximum Item capacity is 2GB (Int32.max). (The unfinished Block specification will include the capability for larger structures)

At this point in time BRBON is considered incomplete. The 'Item' specification has been completed but the 'Block' specification is still open. The Item specification can be used for local storage needs and is useful on its own. The reference API has been prepared for the Block extension by taking the endianness of the data into account. Endianness will be part of the block specification.

Two alternatives for BRBON have been considered: BSON and BinSON. However both were found lacking with respect to the capabilities for high speed (memory mapped) access.

History

Date	Version	Changes
2018-02-24	V0.2	First Draft
2018-03-22	V0.3	Restyling, removed count from count/Value and renamed it small-value. No longer backwards compatible to V0.2
2018-03-30	V0.3.1	Added the UUID type
2018-05-03	V0.4	Added Color and Font General update of the entire document

Block

A Block is a container of Items. It consists of a Block-Header, followed by Items, followed by a Block-Footer.

The purpose of a block is:

- Provide intra-system information (a.o. endianness)
- Allow sizes > 2GB
- Synchronisation of BRBON data between systems
- Allow detection of data inconsistency
- Allow encryption

TBD.

Item

The purpose of an Item is to describe the data contained in its payload as well as to provide path-based access to the payload. An Item can contain an hierarchy or list of other Items within it.

```
Item =
      Header           (* 16 bytes *)
      [,Name]          (* N x 8 bytes *)
      [,Value]         (* Storage area, M x 8 bytes *)
      [,Filler]        (* Filler bytes *)
```

An Item consist of a mandatory Header, an optional Name, an optional Value and an optional Filler field.

The Header, Name and Value are also referred to as ‘header field’, ‘name field’ and ‘value field’.

The value field is optional because for the Null type the value field is unnecessary. And in addition when a fixed length type uses 32 bits or less, the value is included in the header.

The size of an Item is always a multiple of 8 bytes. If an Item can be expressed in less than a multiple of 8, some filler bytes are included to ensure the multiple of 8 size.

However it is also possible to include filler bytes on purpose, for example to reserve space for anticipated storage needs. Filler bytes have an undefined value, but it is recommended to zero them.

Filler bytes are implicit by definition of the size of an item. If the size is larger than necessary, the extra bytes are considered filler.

Header

The header allows deconstruction of the optional Name and Value fields.

```
Header =
      Item-Type,                 (* 1 byte *)
      Item-Options,              (* 1 byte, set to 0 if unused *)
      Item-Flags,                (* 1 byte *)
      Item-Name-Field-Byte-Count (* 1 byte, always a multiple of 8 *)
      Item-Byte-Count,           (* 4 bytes, always a multiple of 8 *)
      Item-Parent-Offset,        (* 4 bytes *)
      Item-Small-Value           (* 4 bytes *)
```

The item header contains all data needed to deconstruct the name field and value field.

The type field allows interpretation of the value field. For the Null type, there is no value field.

The item options are intended for future use, they should be set to zero. Options are persisted and should be evaluated when reading the item or the item’s name/value.

The item flags are intended for API run-time use. An API that uses these bits must set this field to zero when reading or receiving an Item. Among the possible uses are o.a. record locking.

The Item name field byte count contains the number of bytes allocated to the name field. When the name field is not present, it must be set to zero. The maximum length of a name field is therefore 255 bytes. However since the name field must also be a multiple of 8 bytes, the actual limit is 248 bytes. The lower three bits of this field must always be zero.

The item byte count contains the number of bytes of the entire item. If not all bytes are used, then the presence of the filler field is implicit. The item byte count is always a multiple of 8, the lower three bits must always be zero. The endianness of this number is given by the endianness of the Block that contains this item. If no block is present, the machine endianness is assumed.

The item parent offset is the offset from the first byte of a root item to the first byte of the parent item of this item. If this item is not contained in a parent item, it is a root item, and the value of this field is zero. The endianness of this number is given by the endianness of the Block that contains this item. If no block is present, the machine endianness is assumed.

Note: It is not possible to distinguish between a root item and a first level child of a root item by looking only at this value. This is not a problem because in a hierarchy the data will be located in a buffer and the address of the root item will be known to the API. The API can then compare the start address of the item with the start address of the buffer to identify a root.

In addition: this value is only of use when it is necessary to traverse an hierarchy up. In parsers this will generally be unnecessary. In memory managers the API will be address aware and can recalculate & compare this value when reading or receiving Items to ensure validity.

The Item Small Value is a 32 bit field that is used to store fixed length values of maximal 32 bits. The endianness is given by the endianness of the Block that contains this item. If no block is present, the machine endianness is assumed.

Item-Type

The item-type is a byte field specifying the type of payload in either the small-value field or the value-field.

The following types are defined:

Hex code	Type description	Location
0	Illegal value.	n.a.
0x01	Null	n.a.
0x02	Bool	Small-Value
0x03	Int8	Small-Value
0x04	Int16	Small-Value
0x05	Int32	Small-Value
0x06	Int64	Value-Field
0x07	UInt8	Small-Value
0x08	UInt16	Small-Value
0x09	UInt32	Small-Value
0x0A	UInt64	Value-Field
0x0B	Float32	Small-Value
0x0C	Float64	Value-Field
0x0D	String	Value-Field
0x0E	CRC String	Value-Field
0x0F	Binary	Value-Field
0x10	CRC Binary	Value-Field

Hex code	Type description	Location
0x11	Array	Value-Field
0x12	Dictionary	Value-Field
0x13	Sequence	Value-Field
0x14	Table	Value-Field
0x15	UUID	Value-Field
0x16	RGBA	Value-Field
0x17	Font	Value-Field
0x18-0x7F	Reserved (do not use)	n.a.
0x80-0xFF	Available for user definitions (free for all).	n.a.

For more information on the types and how they are coded, see the Types section.

Name

```
Name =
      Name-CRC,              (* 2 bytes, CRC16, ARC, rev poly 0x8005 *)
      Name-UTF8-Byte-Count, (* 1 byte, the number of used UTF8 bytes *)
      Name-UTF8-Byte-Code    (* UTF8-Byte-Code bytes *)
      [, Name-Filler]        (* Filler, optional *)
```

`Name-CRC`: The hash value is a CRC16 using the reverse polynomial 0x8005 (= 0xA001 reversed) and 0 as the start seed.

`Name-UTF8-Byte-Count`: This is the number of bytes used by the UTF8 byte code that constitutes the name.

`Name-UTF8-Byte-Code`: The UTF-8 byte code that can be converted into a valid string.

`Name-Filler`: The filler is a number of bytes that ensure that the number of bytes in the name field is a multiple of 8. This limits the total name field to a maximum of 248 bytes.

Note: Subtracting the 3 bytes used by CRC and UTF8 byte count gives a maximum size of 245 bytes for the name characters.

Rationale: The CRC allows for faster finding of items by name. If the CRC does not match, the UTF8 byte code does not need to be compared. Names longer than 245 characters seem unlikely, they would make for unreadable code.

Value

The layout of the value field is given in the section Types.

Filler

Filler data must be present if the value field is not a multiple of 8 bytes. It then rounds of the item byte count to a multiple of 8 bytes. More often though the filler may be present because a designer wants to ensure that some area is available for later expansion without having to shift the entire data field following this item.

Types

Null

The null is a special type that has no associated value. It is intended to be used as a placeholder that can be changed into any other type.

It can contain filler data.

A null cannot be stored in an array or table.

Bool

The value field is a single byte containing either zero (false) or non-zero (true).

When stored in an item the value will be stored in the first byte of the small-value field.

When stored in an array or table the value will be stored in a single byte, the first byte in an element or field.

Int8

A single byte, range -128 … +127. (0x80 … 0x7F)

When stored in an item the value will be stored in the first byte of the small-value field.

When stored in an array or table the value will be stored in a single byte, the first byte in an element or field.

Int16

A 2-byte value, range -32,768 … +32,767 (0x8000 … 0x7FFF)

When stored in an item the value will be stored in the first two bytes of the small-value field.

When stored in an array or table the value will be stored in the first 2 consecutive bytes in an element or field.

Endianness is specified by the block endianness or when the block is absent the machine endianness.

Int32

A 4-byte value, range -2,147,483,648 … +2,147,483,647 (0x8000_0000 … 0x7FFF_FFFF)

When stored in an item the value will be stored in the small-value field.

When stored in an array or table the value will be stored in the first 4 consecutive bytes in an element or field.

Endianness is specified by the block endianness or when the block is absent the machine endianness.

Int64

An 8-byte value, range -9,223,372,036,854,775,808 … +9,223,372,036,854,775,807 (0x8000_0000_0000_0000 … 0x7FFF_FFFF_FFFF_FFFF)

When stored in an item the value will be stored in the first eight consecutive bytes in the value field.

When stored in an array or table the value will be stored in the first eight consecutive bytes in the value field.

The small-value field is not used.

Endianness is specified by the block endianness or when the block is absent the machine endianness.

UInt8

A single byte, range 0… +255. (0x00 … 0xFF)

When stored in an item the value will be stored in the first byte of the small-value field.

When stored in an array or table the value will be stored in a single byte, the first byte in an element or field.

UInt16

A 2-byte value, range 0 … +65,535 (0x0000 … 0xFFFF)

When stored in an item the value will be stored in the first two bytes of the small-value field.

When stored in an array or table the value will be stored in the first 2 consecutive bytes in an element or field.

Endianness is specified by the block endianness or when the block is absent the machine endianness.

UInt32

A 4-byte value, range 0 … +4,294,967,295 (0x0000_0000 … 0x8FFF_FFFF)

When stored in an item the value will be stored in the small-value field.

When stored in an array or table the value will be stored in the first 4 consecutive bytes in an element or field.

Endianness is specified by the block endianness or when the block is absent the machine endianness.

UInt64

An 8-byte value, range 0 … +18,446,744,073,709,551,615 (0x8000_0000_0000_0000 … 0x7FFF_FFFF_FFFF_FFFF)

When stored in an item the value will be stored in the first eight consecutive bytes in the value field.

When stored in an array or table the value will be stored in the first eight consecutive bytes in the value field.

The small-value field is not used.

Endianness is specified by the block endianness or when the block is absent the machine endianness.

Float32

A 4-byte value. Accurate to about 6 decimals, range approx 1.1e-38 to 3.4e38

When stored in an item the value will be stored in the small-value field.

When stored in an array or table the value will be stored in the first 4 consecutive bytes in an element or field.

Endianness is specified by the block endianness or when the block is absent the machine endianness.

Float64

An 8-byte value, range apron 2.2e-308 to 1.7e+308

When stored in an item the value will be stored in the first eight consecutive bytes in the value field.

When stored in an array or table the value will be stored in the first eight consecutive bytes in the value field.

The small-value field is not used.

Endianness is specified by the block endianness or when the block is absent the machine endianness.

String

```
String = byte-count [{, UTF8-Byte-Code}]
```

The string is a sequence of bytes representing the UTF8 code of the string. The UT8 byte code is preceded by a count of the number of bytes in the byte code.

The layout is as follows:

Offset	Content	Units	Bytes
0	Byte Count	UInt32	4
4	UTF8 Byte Code	Bytes	Byte Count
4 + Byte Count	Filler	Bytes	As needed

This layout is either stored in the Value-Field of an item, the element of an array or the field in a table.

Endianness is specified by the block endianness or when the block is absent the machine endianness.

CRC String

```
CRC-String = CRC32, Byte-Count [{, UTF8-Byte-Code}]
```

The string is a sequence of bytes representing the UTF8 code of the string. The UT8 byte code is preceded by the CRC32 of the bytes in the UTF8 byte code and a count of the number of bytes in the byte code.

The CRC32 uses a seed value of 0.

The layout is as follows:

Offset	Content	Units	Bytes
0	CRC32	UInt32	4
4	Byte Count	UInt32	4
8	UTF8 Byte Code	Bytes	Byte Count
8 + Byte Count	Filler	Bytes	As needed

This layout is either stored in the Value-Field of an item, the element of an array or the field in a table.

Endianness is specified by the block endianness or when the block is absent the machine endianness.

Rationale: This storage structure makes it possible to always use the first 8 bytes of the crc-string for a quick check to see if the following string is **not** equal to a search string. Only when the first 8 bytes are the same, the remaining bytes of the string need to be verified.

Binary

```
Binary = byte-count [{, Byte}]
```

A binary is a sequence of bytes. The byte sequence is preceded by a count of the number of bytes.

The layout is as follows:

Offset	Content	Units	Bytes
0	Byte Count	UInt32	4

Offset	Content	Units	Bytes
4	Binary	Bytes	Byte Count
4 + Byte Count	Filler	Bytes	As needed

This layout is either stored in the Value-Field of an item, the element of an array or the field in a table.

Endianness is specified by the block endianness or when the block is absent the machine endianness.

CRC Binary

```
CRC-Binary = CRC32, Byte-Count [{, Byte}]
```

A binary is a sequence of bytes. The byte sequence is preceded by a CRC32 over the bytes in the binary and a count of the number of bytes.

The CRC32 uses a seed value of 0.

The layout is as follows:

Offset	Content	Units	Bytes
0	CRC32	UInt32	4
4	Byte Count	UInt32	4
8	Binary	Bytes	Byte Count
8 + Byte Count	Filler	Bytes	As needed

This layout is either stored in the Value-Field of an item, the element of an array or the field in a table.

Endianness is specified by the block endianness or when the block is absent the machine endianness.

Rationale: This storage structure makes it possible to always use the first 8 bytes of the crc-binary for a quick check to see if the following binary is **not** equal to a search binary. Only when the first 8 bytes are the same, the remaining bytes of the binary need to be verified.

Array

Contains a sequence of elements. All of the same byte count (length) and of the same type. Addressing of the elements is always through their index.

```
Array =
      Reserved,              (* 4 Bytes, must be zero *)
      Element-Type,          (* 1 Byte, see Item Type *)
      Reserved,              (* 3 Bytes, must be zero *)
      Element-Count,         (* The number of elements *)
      Element-Byte-Count     (* The number of bytes in an element *)
      [{, Element }]
```

The layout is as follows:

Offset	Content	Units	Bytes
0	Reserved	UInt32	4
4	Element-Type	Item-Type	1

Offset	Content	Units	Bytes
5	Zero	Bytes	3
8	Element-Count	UInt32	4
12	Element-Byte-Count	UInt32	4
16	Array	Element	Element-Count x Element-Byte-Count
16 + Element-Count x Element-Byte-Count	Filler	Bytes	As needed

When an array is contained in an array or table, it is included as an item.

Discussion: The array type is intended as a space saving device. (As well as a storage pendant for the array type in programming languages). Storing other containers in an array is possible, at the cost of a minor overhead since containers are always stored in total (i.e. as complete items). A mix of containers could easily be supported but can also lead to misunderstandings. Hence at the moment it has been excluded from the specification. When an array contains another container type all those containers must also be of the same type.

Sequence

Contains a sequence of other items.

```
Sequence = Count [{,Item}]]
```

A sequence does not place any restrictions on the items it contains. (Unlike a dictionary)

The layout is as follows:

Offset	Content	Units	Bytes
0	Reserved	UInt32	4
4	Count	UInt32	4
8	Items	Item	As needed
As needed	Filler	Bytes	As needed

When a sequence is contained in an array or table, is is included as an item.

Dictionary

See Sequence, but each item in the dictionary is guaranteed to have a unique name.

Table

A table is conceptually an array with the same type of dictionary for each element. The key's of the dictionary fields are only present once.

```
Table =
      RowCount,              (* 4 bytes, UInt32 *)
      ColumnCount,           (* 4 bytes, UInt32 *)
      RowsOffset,            (* 4 bytes, UInt32 *)
      RowByteCount           (* 4 bytes, UInt32 *)
      [{, ColumnDescriptor }](* 16 bytes * number of columns *)
```

```
      [{, ColumnName }]     (* N bytes, 8 byte boundary, * num of cols *)
      [{, Row }]            (* N bytes, 8 byte boundary, * num of cols *)

ColumnDescriptor =
      ColumnNameCrc16,      (* 2 bytes *)
      ColumnNameByteCount,  (* 1 bytes *)
      ColumnFieldType,      (* 1 bytes *)
      ColumnNameUtf8Offset, (* 4 bytes *)
      ColumnFieldOffset,    (* 4 bytes *)
      ColumnFieldByteCount  (* 4 bytes *)

ColumnName = UTF8-Byte-Count {, UTF8-Byte-Code } (* Max 248 bytes *)
Row = {, ColumnValue } (* 8 byte boundary aligned *)
```

`RowCount`: The number of rows in this table (UInt32)

`ColumnCount`: The number of columns in this table (UInt32)

`RowsOffset`: The offset from the start of the value field to the first byte of value of the first column in the first row.

RowByteCount: The number of bytes in a row, is equal to the value of all columnValueByteCounts added together.

`ColumnDescriptor`: There is a column descriptor for each column, containing descriptions of the name and value for that column. Each column descriptor is 16 bytes, and all column descriptors are arranged sequentially.

`ColumnNameCrc16`: The CRC16 value of the UTF8-byte code that is the name of the column. Using the reverse polynomial 0x8005 (= 0xA001 reversed) and 0 as the start seed.

`ColumnNameByteCount`: The number of bytes for the UTF8-byte sequence of the column name. (Not the number of used bytes, this is part of the ColumnName field)

`ColumnValueType`: The type of item stored in this column.

`ColumnNameUtf8Offset`: The offset from the start of the value area to the sequence of UTF8 byte codes that specify the name of the column.

`ColumnValueOffset`: The offset from the start of the row where this column 's value is located.

`ColumnValueByteCount`: The number of bytes in the column's value.

The ColumnDescriptor and ColumnName are located in two different area's. The descriptor contains an offset into ColumnNames field. For each descriptor there must be one column name and vice versa.

Column names must be unique inside a table. Column names have a maximum length of 245 UTF8 byte code bytes.

Note: An API should probably set the initial content of a column field when a row is added to all zero to prevent/detect illegal content.

The layout of the table value field is as follows:

Offset	Content	Units	Bytes
0	Row Count	UInt32	4
4	Column Count	UInt32	4
8	Rows Offset	UInt32	4

Offset	Content	Units	Bytes
12	Row Byte Count	UInt32	4
16	Column Descriptors	Column Descriptor	16 x Column Count
16 + 16 x Column Count	Column Names	Column Name	As needed
Rows Offset	Rows (Fields)	Rows	Row Count x Row Byte Count
Rows Offset + Row Count x Row Byte Count	Filler	Bytes	As needed

The layout of the Column Descriptors is as follows:

Offset	Content	Units	Bytes
0	Column Name CRC16	UInt16	2
2	Column Name Field Byte Count	UInt8	1
3	Column Field Type	Item Type	1
4	Column Name UTF8 Offset	UInt32	4
8	Column Field Offset	UInt32	4
12	Column Field Byte Count	UInt32	4

The layout of the Column Names is as follows:

Offset	Content	Units	Bytes
0	Column Name Byte Count	UInt8	1
4	Column Name UTF8 Byte Code	Bytes	Column Name Byte Count

UUID

An array of 16 bytes.

```
UUID = Bytes     (* 16 bytes, UInt8 *)
```

When stored in an item the value will be stored in the first 16 consecutive bytes in the value field.

When stored in an array or table the value will be stored in the first 16 consecutive bytes in the value field.

The small-value field is not used.

There is no endianness associated wit this value.

Color

Four (4) UInt8 values that together make up a color specification. Stored in the sequence R(ed), G(reen), B(lue), A(lpha).

```
Color =
      Red,       (* 1 bytes, UInt8 *)
```

```
    Green,      (* 1 bytes, UInt8 *)
    Blue,       (* 1 bytes, UInt8 *)
    Alpha       (* 1 bytes, UInt8 *)
```

When stored in an item the values will be stored in the small value field.

When stored in an array or table the value will be stored in the first 4 consecutive bytes in the value field.

The small-value field is not used.

The layout is as follows:

Offset	Content	Units	Bytes
0	Red	UInt8	1
4	Green	UInt8	1
8	Blue	UInt8	1
12	Alpha	UInt8	1

This layout is either stored in the Small-Value-Field of an item, the element of an array or the field in a table.

Endianness is irrelevant.

Rationale: As color information is frequently used it is more memory efficient to store it in its own type instead of a composite type.

Font

A size specification followed by a font family name and a font name describing the font and the typeface.

```
Font =
    Size,           (* 4 bytes, Float32 *)
    Family-Size,    (* 1 byte, UInt8 *)
    Name-Size,      (* 1 byte, UInt8 *)
    Family,         (* Family-Size bytes, String, UTF8 encoded *)
    Name            (* Name-Size bytes, String, UTF8 encoded *)
```

When stored in an item it will be stored in the value field.

When stored in an array or table it will be stored in the first consecutive bytes of the value field.

The small-value field is not used.

The value field layout is as follows:

Offset	Content	Units	Bytes
0	Size	Float32	4
4	Family Size	UInt8	1
5	Name Size	UInt8	1
6	Family	UTF8	Family Size
6 + Family Size	Name	UTF8	Name Size

This layout is either stored in the Value-Field of an item, the element of an array or the field in a table.

Endianness is specified by the block endianness or when the block is absent the machine endianness.

Rationale: As font information is frequently used it is more memory efficient to store it in its own type instead of a composite type.

Usage: It is recommended to fully specify the font using the `size` and `name` fields only. The `family` is a fall-back to be used when the `name` field does not resolve to an installed font.

— — — —

END OF DOCUMENT