BRBON

V 0.2

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 is 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.
- Collection types: Array, Dictionary, Sequence, Table.
- Special type: Null.
- 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

2018-02-24 V0.2 First draft of 0.2

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 of other Items within it.

Item-Header	Item-Byte-Count	Parent-Offset	Count/Value	NVR

```
Item = Item-Descriptor [, NVR]
Item-Descriptor =
      Item-Header,     (* 4 bytes *)
      Item-Byte-Count, (* 4 bytes, value is always a multiple of 8 *)
      Parent-Offset,   (* 4 bytes *)
      Count/Value      (* 4 bytes, value or number of items contained *)
```

`Item-Header`: Identifies the type of data contained in this item with options, run-time flags and length of an optional name field.

`Item-Byte-Count`: The number of bytes of the item (minimum of 16 bytes, always a multiple of 8, maximum of Int32.max bytes ~ 2GB)

`Parent-Offset`: The offset of the parent relative to the first item byte in a block. This value can be used by an API to traverse an item hierarchy upward. Note that this value can be recalculated at any time.

`Count/Value`: For types equal or smaller than 32bits, this is the value in the item. The value itself is aligned to the first byte in this field, i.e. an 8 bit value occupies the first byte, a 16 bit value the first two bytes etc. Unused bytes contain random data. For 64 bit types, this field is not used (may contain random data). For variable length types (Binary, String, Dictionary, Sequence, Array, Table) this field holds the number of items or the number of bytes that are contained in the value in the NVR field.

`NVR:` is an abbreviation for: Name, Value, Reserved. Though ‘reserved’ could also have been named ‘filler’. The NVR field is optional. Its optionality is given by the the value of the name length field and/or the type of value contained in the item.

Item Header

Type	Options	Flags	Name-Field-Byte-Count

```
Item-Header =
      Type,                 (* 1 byte *)
      Options,              (* 1 byte, set to 0 if unused *)
      Flags,                (* 1 byte *)
      Name-Field-Byte-Count (* 1 byte *)
```

Type

The type is a byte field uniquely identifying the content of the Value (in the NVR field) field.

The following types are defined:

Use Value	Use Count	Hex code	Type description
0	0	0	Illegal value. When found the data is likely corrupt.

Use Value	Use Count	Hex code	Type description
0	0	0x01	Int64
0	0	0x02	UInt64
0	0	0x03	Float64
0	0	0x04	Brbon String
0	0	0x03-0x1F	Reserved (do not use)
0	0	0x20-0x3F	Available for user definitions (free for all). The Count/Value field should not be used.
0	1	0x40	String
0	1	0x41	Array
0	1	0x42	Dictionary
0	1	0x43	Sequence
0	1	0x44	Binary
0	1	0x45	Table
0	1	0x46-0x5F	Reserved (do not use)
0	1	0x60-0x7F	Available for user definitions (free for all). The Count/Value field should be used as Count.
1	0	0x80	Null (Used as a placeholder value)
1	0	0x81	Bool
1	0	0x82	Int8
1	0	0x83	Int16
1	0	0x84	Int32
1	0	0x85	UInt8
1	0	0x86	UInt16
1	0	0x87	UInt32
1	0	0x88	Float32
1	0	0x89-0x9F	Reserved (do not use)
1	0	0xA0-0xBF	Available for user definitions (free for all). The Count/Value field should be used as Value.
1	1	0xC0-0xFF	Reserved (do not use)

For a discussion of the layout of each of these types, see the value field below.

Options

A field of 8 bits witch is currently unused and must be zero. If there is a non-zero bit, it must be assumed that the data of this item cannot be interpreted.

Bits 7 - 0	Unspecified, Reserved, Must be zero. If a non-zero bit is found, synchronisation is lost.

Flags

Bits 7 - 0	Unspecified, can be used at run-time by the API.

The flag area is for used during runtime. When stored or transferred the bits are undefined.

Name Length

Contains an UInt8 value indicating the number of bytes in the name field. This value is/must always be a multiple of 8. Its maximum value is 248.

When the name field is not present, this value is zero.

The least significant 3 bits are reserved and should be zero. If one of these bits is set, it is likely an error in the data structure. It is recommended to raise an error and/or disregard the item.

Item Byte Count

This is an UInt32 value that specifies how many bytes this item occupies.

The range is 16 … Int32.max and must be a multiple of 8.

The least significant 3 bits are reserved and should be zero. If one of these bits is set, it is likely an error in the data structure. It is recommended to raise an error and/or disregard the item.

Parent-Offset

An UInt32 which when added to the address of the first byte of the root item yields the address of the first byte of the item which this item is contained in.

Range 0 … Int32.max and will be a multiple of 8 because the size of all items is a multiple of 8.

The value 0 is a special case, it may point to self if self is the root item or it may point to the root item if self is contained in the root item. To differentiate between them, compare the address of the ‘parent’ with the address of ‘self’, if it is equal, the item is the root item.

The least significant 3 bits are reserved and should be 0. If one of these bits is set, it is likely an error in the data structure. It is recommended to raise an error and/or disregard the item.

Count/Value

4 bytes that are used as either a value or a count for the number of items/elements/bytes in the value field.

When used as a value the value is aligned with the first byte:

	1st Byte	2nd byte	3rd byte	4th byte
8 Bit value	X	-	-	-
16 Bit value	X	X	-	-
32 Bit value	X	X	X	X

The endianness of the value will be as specified for the entire block.

When used as a count field, the 4 bytes are used a a single UInt32. The usage of the count field depends on the type of the value field:

Item Type	Usage
String	The number of UTF8 code bytes
Binary	The number of bytes
Array	The number of elements
Sequence	The number of items
Dictionary	The number of items
Table	The number of rows

Name Value Reserved (NVR)

Name	Value	Reserved

```
NVR =
      [Name]       (* Name-Length *)
      [,Value]     (* Value-Length, depends on Type *)
      [,Reserved] (* NVR-Length - Name-Length - Value-Length *)
```

`Name`: An optional name. Only present if the Name-Field-Byte-Count in the header is non-zero.

`Value`: An optional value. Present for 64bit types and variable length types.

`Reserved`: Implicit if the item byte count is larger than necessary for the name & value.

Name Field

Hash	Count	UTF8-code	Filler

```
Name-Field =
      Hash,       (* 2 bytes = CRC16, ARC, reverse polynomial 0x8005 *)
      Count,      (* 1 byte, the number of used UTF8 bytes *)
      UTF8-Code   (* UTF8-Byte-Count *)
      [, Filler] (* Filler-Length = 1..7 bytes, all zero *)
```

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

`Count`: The count is the number of bytes used by the UTF8 code that makes up the name.

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

`Filler`: The filler is a number of bytes that ensure that the number of bytes in the name field is a multiple of 8. The content of the filler bytes must be 0. This limits the total name field to a maximum of 248 bytes.

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

Value

The value bytes are the bytes that represent the value of the type stored in this item.

The layout/special considerations for the value content depending on the type are as follows:

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.

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 count/value field.

When stored in an array the value will be stored in a single byte in the value field.

In an array the values are ‘packed’, i.e. the value field of the bool itself is *not* expanded to an 8 byte boundary.

Endianness is irrelevant.

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 count/value field.

When stored in an array the value will be stored in a single byte in the value field.

In an array the values are ‘packed’, i.e. the value field of the Int8 itself is *not* expanded to an 8 byte boundary.

Endianness is irrelevant.

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 count/value field.

When stored in an array the value will be stored in 2 consecutive bytes in the value field.

In an array the values are ‘packed’, i.e. the value field of the Int16 itself is *not* expanded to an 8 byte boundary.

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 count/value field.

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

In an array the values are ‘packed’, i.e. the value field of the Int32 itself is *not* expanded to an 8 byte boundary.

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 value field.

When stored in an array the value will be stored in 8 consecutive bytes in the value field.

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 count/value field.

When stored in an array the value will be stored in a single byte in the value field.

In an array the values are ‘packed’, i.e. the value field of the UInt8 itself is *not* expanded to an 8 byte boundary.

Endianness is irrelevant.

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 count/value field.

When stored in an array the value will be stored in 2 consecutive bytes in the value field.

In an array the values are ‘packed’, i.e. the value field of the UInt16 itself is *not* expanded to an 8 byte boundary.

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 count/value field.

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

In an array the values are ‘packed’, i.e. the value field of the UInt32 itself is *not* expanded to an 8 byte boundary.

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 value field.

When stored in an array the value will be stored in 8 consecutive bytes in the value field.

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 count/value field.

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

In an array the values are ‘packed’, i.e. the value field of the Float32 itself is *not* expanded to an 8 byte boundary.

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 value field.

When stored in an array the value will be stored in 8 consecutive bytes in the value field.

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.

When stored as an item, the `byte-count` is specified in the ‘count’ field of the Item as an UInt32.

When stored in an array, the first four bytes of each element will specify the `byte-count` of the string as an UInt32. The element length will always be a multiple of 8 (including the count)

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

There is no trailing zero specified.

Binary

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

A binary is a sequence of bytes.

When stored as an item, the `byte-count` is specified in the ‘count’ field of the Item as an UInt32.

When stored in an array, the first four bytes of each element will specify the `byte-count` of the binary as an UInt32. The element length will always be a multiple of 8 (including the count)

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

BrbonString

An BrbonString is a string that is preceded by a CRC16 (like the name). This allows for faster search operations.

```
BrbonString =
    Byte-Count,           (* UInt32, 4 bytes *)
    crc16,                (* UInt16, 2 bytes *)
    [{, UTF8-Byte-Codes}] (* At least two zero bytes *)
```

The Byte Count is the number of bytes in the UTF8 byte code.

The UTF8-byte-code is a sequence of bytes representing the UTF8 code of the string.

The first two UTF8 byte codes after the crc16 must always be set. If the string is empty the first two bytes must be zero. If the string is only 1 byte long, the second byte must be zero.

The count/value field is unused.

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

There is no trailing zero specified.

Note: This storage structure makes it possible to always use the first 8 bytes of the 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.

Sequence

Contains a sequence of other items.

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

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

The number of items contained is specified in the ‘count’ field of the item.

If a sequence is contained in an array, each element has the same layout as an item of type sequence.

Array

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

```
Array = Count, Element-Descriptor [{, Element }]
Element-Descriptor =
      Element-Header,  (* 4 bytes *)
      Element-Length   (* 4 bytes, byte count of each element *)
Element-Header =
      Element-Type,    (* 1 Byte, same as Item-Header type *)
      Zero             (* 3 Bytes, all zero *)
```

The number of elements in the array is specified in the count field of the item.

Note: An element is always addressed through the index, even though the Array, Dictionary and Sequence and Table elements may have a name.

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 *)
    ColumnValueType,      (* 1 bytes *)
    ColumnNameUtf8Offset, (* 4 bytes *)
    ColumnValueOffset,    (* 4 bytes *)
    ColumnValueByteCount  (* 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.

Reserved:

An optional field that exists when the specified length is larger than the actual length of the name + value fields.

The content of the reserved bytes is random.

Notes

- The minimum size for an Item is 16 bytes.
- A name has a minimum of 8 bytes. A name has 3 overhead bytes hence the name itself can be 5 bytes long (UTF8 code) to occupy only 8 bytes. Longer names will increase their allotted space in chunks of 8 bytes up to a maximum of 248 bytes.

- If an Item is named, the minimum size increases to 24 bytes

— — — —

END OF DOCUMENT