BRBON

An Binary Object Notation by Balancing Rock.

Introduction

JSON is a convenient and very maintainable format for data storage and transfer. However the price paid for this convenience is speed. This is not always a problem, but JSON also invites developers to use it as a database replacement system. This works fine for small amounts of data and low intensity access. But as the database grows and is used more intensely a speedier approach will often become necessary.

BRBON was derived from JSON but designed for speed, it follows the same array/dictionary approach as JSON.

BRBON has been enhanced with vectors and 8-byte alignement to allow for maximum speed. There is no optimisation for space for two reasons: memory and storage space is cheap nowadays, and when transferring data there will usually be a compression mechanism available, possibly in hardware.

Strictly speaking BRBON refers to the format specification in this document, not to an implementation. Though an implementation with the same is provided. It should be noted though that the current implementation with this name does not implement a memory map based access mechanism but a simpler memory structure based approach.

The current specification is version 0.2 and is intended for storage and access only. It thus concerns itself only with the Item specification. In the future a block wrapper is envisioned that will contain the Items as defined in this document and that may be used to exchange data between applications.

The most glaring omission from this specification is a way to specify the endianness of the Items. This will in the future be included in the block wrapper. For now it is assumed that data is only stored on a single machine or transferred between machines with the same endianness.

The most glaring omission from the current implementation is multithreading support. The API should be accessed from within a single thread.

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

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 to provide intra-system information about access to the Items contained within it, as well as a mechanism that allows synchronisation of BRBON data between systems.

To be defined.

Item

The purpose of an Item is to describe the data contained in the payload as well as to provide path-based access to this data. An Item can contain a hierarchy of other Items within it. A path is a description of which item an API user wants to access. A path can consist of a mix of names and indexes.

Format

Item


```
Item =
      Item-Header, (* 4 bytes *)
      NVR-Length, (* 4 bytes - always a multiple of 8 *)
      NVR (* NVR-Length *)
```

The main driver for the overall layout is speed over size. This leads to the following two approaches:

- The Item size is always a multiple of 8 bytes.
- The next item in a sequence of items can be found by vectoring over the NVR part (which is the only part that can have a variable length)

NVR is an abbreviation for: Name, Value, Reserved.

Item Header


```
Item-Header =
      Type, (* 1 byte *)
      Options, (* 1 byte, must be 0 *)
      Flags, (* 1 byte, must be 0 *)
      Name-Length (* 1 byte *)
```

Type

The type is a UInt8 uniquely identifying the content of the Value field.

The following types are defined:

Hex code	Type description
0x00	Null
0x01	Bool
0x02	Int8
0x03	Int16
0x04	Int32
0x05	Int64
0x06	UInt8

Hex code	Type description
0x07	UInt16
0x08	UInt32
0x09	UInt64
0x0A	Float32
0x0B	Float64
0x0C	String
0x0D	Array
0x0E	Dictionary
0x0F	Sequence
0x10	Binary
0x11-0x7F	Reserved (do not use)
0x80-0xFF	Available for user definitions (free for all)

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

A byte with the value 0.

Flag area used during runtime. When stored they should always be zero.

Name Length

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

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

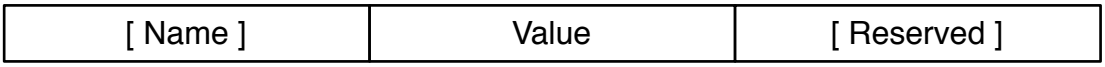
The least significant 2 bits are reserved and should not be used. If one of these two bits is set, it is likely an error in the data structure. It is recommended to raise an error and/or disregard the item.

NVR Length

This is an UInt32 value that specifies how many bytes the NVR (Name Value reserved) field uses. The bytes of this value ares stored in the endianness the block it is part of. (If no block is used, then the endianness of the machine it is on)

The least significant 2 bits are reserved and should not be used. If one of these two bits is set, it is likely an error in the data structure. It is recommended to raise an error and/or disregard the item.

Name Value Reserved (NVR)




```
NVR =
      [Name, ] (* Name-Length *)
      Value (* Value-Length, depends on Type, always a multiple of 8 *)
      [, Reserved] (* NVR-Length - Name-Length - Value-Length *)
```

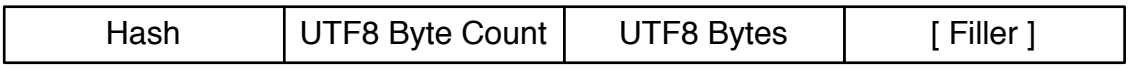
The name field is optional.

The value field is not optional, but its length may be zero. This can be the case for items containing a ‘null’. Note that a null may have a name and/or a reserved field.

The value field will always contain N times 8 bytes, but N may be zero.

The reserved field is optional and is implicitly present when the length of the NVR field is bigger than the name field and value field combined. The reserved field must be N times 8 bytes and N may be zero.

Name




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

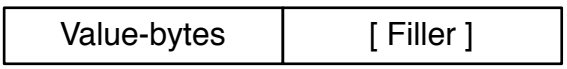
The hash value is a CRC16 using the reverse polynomial 0x8005 (= 0xA001 reversed) and 0 as the start seed.

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

UTF8 bytes is a sequence of UTF8 code units. Note that this string need not be zero terminated.

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

```
Value =
      Value-Bytes (* Number of bytes depend on type of value *)
      [, Filler] (* Filler-Length = 1..7 bytes, all zero *)
```

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

The filler is a number of bytes that ensure that the number of bytes in the value field is a multiple of 8. The content of the bytes must be 0.

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

Null

1 byte value indicating the type of the null item. Followed by 7 filler bytes. The type field is encoded in the same pattern as used for the Item-Header type field.

Bool

An 8 bit (1 byte) value that represents either true (!= 0x00) or false (== 0x00).

To improve synchronisation it is recommended to use either 0x01 or 0xFF for ‘true’.

There will be 7 filler bytes after the bool-byte.

UInt8 & Int8

A 1 byte value with 7 filler bytes.

UInt16 & Int16

A 2 byte value in the endianness of the block. (Machine endianness if there is no block)

Followed by 6 filler bytes.

UInt32 & Int32 & Float32

A 4 byte value in the endianness of the block. (Machine endianness if there is no block)

Followed by 4 filler bytes.

UInt64 & Int64 & Float64

A 8 byte value in the endianness of the block. (Machine endianness if there is no block)

There are no filler bytes.

String

```
String =
      Count (* 4 bytes *)
      [{, UTF8-Byte-Codes}]
```

The `Count` is an UInt32 number in the endianness of the block (machine endianness if there is no block) that specifies the number of bytes in the UTF8 code. May be zero.

Filler data may be present.

Sequence

Contains a sequence of other items.

```
Sequence =
      Count,           (* 4 bytes *)
      Parent-Offset    (* 4 bytes *)
      [{,Item}]]
```

The `Count` is an UInt32 number in the endianness of the block (machine endianness if there is no block) that specifies the number of Items that follow. May be zero.

The is `Parent-Offset` the offset from the start of the top level item to the parent of this item. For the top level element it is zero.

Filler data may be present.

Array

Contains a sequence of value-byte fields. All of the same type and length.

```
Array =
      Count,           (* 4 bytes *)
      Parent-Offset    (* 4 bytes *)
      Element-Spec,    (* 4 bytes is equal to the Item Header *)
      Element-Size     (* 4 bytes *)
      [{, Value-Bytes }]


Element-Spec & Element-Size == Item-Header & NVR-Length
```

The `Count` is an UInt32 for the number of elements in the array. May be zero.

The is `Parent-Offset` the offset from the start of the top level item to the parent of this item. For the top level element it is zero.

The `Element Spec` and Length can be viewed as the Item-Header and NVR-Length of the individual elements in the array.

In fact, the `Element-Spec`, size and the Value-Bytes of the first element form together a fully functional Item.

The `Value-Bytes` contain the stored data including a possible name. There are no filer bytes for elements. But there may be reserved bytes if the Element Byte Count is larger than necessary for the stored data.

Filler data may be present (after the array)

Dictionary

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

Binary

```
Binary =
      Count, (* 4 bytes *)
      Zero (*4 bytes, all zero, to even out to an 8 byte boundary)
```

```
[{,byte}]]
```

An UInt32 indicating how many bytes follow.

Filler data may be present.

Reserved:

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

It is recommended to zero reserved bytes for improved synchronisation performance.

Notes

- Each Item has 4 bytes header and 4 bytes payload length. (payload is the data that is stored)
- A payload has a minimum size of 8 bytes and is increased in 8 bytes units as necessary. Smaller types will be padded with filler bytes.
- 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
- Item types 126 .. 255 are user defined, item types below that number should be implemented by all BRBON API's.

— — — —

END OF DOCUMENT