BRBON

V 0.2

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.

The BRBON API was designed as a drop-in replacement for our implementation of JSON (called VJSON) but designed for speed, it follows the same array/dictionary approach as JSON. Like JSON though, data-exchangeability is an issue hence the endianness of the format must be specifiable as well.

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.

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.

A noticeable limitation is that the size of an Item cannot exceed 2GB (Int32.max). Though a future release is expected to support larger data structures. Since however BRBON is designed for speed, an Item should fit in memory for fastest access. Hence at the current state of hardware design it does not make much sense to use larger Items.

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 an 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.

Item

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

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

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.
- Contiguously stored items (dictionary, sequence) always start at 8 byte boundaries.
- 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)
- An offset to the parent item is present (in effect making the Item data structure traversable in both directions)

`Item-Header`: Identify the type of data contained in this item.

`Item-Length`: The number of bytes of the item (minimum 16, always a multiple of 8)

`Parent-Offset`: The offset of the parent relative to the start of the item. The parent offset may be zero in streamed data. If data is streamed, the receiver should fill in the parent offset if needed.

`Count/Value`: For types smaller than 32bits, this is the value of the type. For 64 bit types, this field is not used (zero). For variable length types (Binary, String, Dictionary, Sequence, Array) this field holds the number of items or the number of bytes that are contained 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.

Item Header

Type	Options	Flags	Name Length

```
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:

Use Value	Use Count	Hex code	Type description
0	0	0	Illegal value. When found the data is likely corrupt.
0	0	0x01	Int64
0	0	0x02	UInt64
0	0	0x03	Float64
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	0x45-0x5F	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

A byte with the value 0.

Flag area may be used during runtime. When stored or transferred 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. 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 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.

Parent-Offset

An UInt32 which when added to the pointer that points at the first byte of the root Item yields the pointer to the first byte of the parent Item.

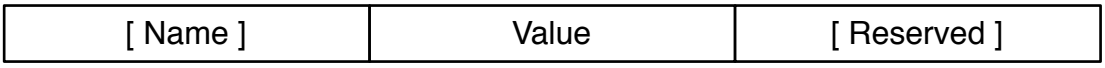
The least significant 2 bits are reserved and should not be used. If one of these two bits is set, it is likely an error in the data structure. It is recommended to raise an error and/or disregard the item.

Item-Length

This is an UInt32 value that specifies how many bytes there are in the Item. The bytes of this value are stored in the endianness of the block it is part of. (If no block is used, then the endianness of the machine it is on)

The least significant 3 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)



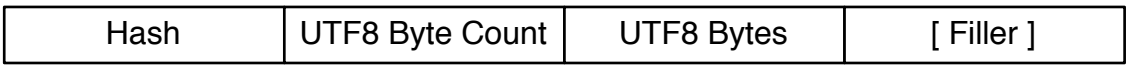

```
NVR =
      [Name, ]          (* Name-Length *)
      Value             (* Value-Length, depends on Type *)
      [, Reserved]      (* NVR-Length - Name-Length - Value-Length *)
```

The name field is optional.

The value field is not optional.

The reserved field is optional but is implicitly present when the length of the Value field is less than a multiple of 8 bytes. It is then used to fill out the NVR length to a multiple of 8 bytes.

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.

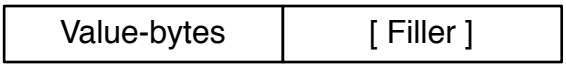
The UTF8 byte count is the number of bytes used by the UTF8 code that makes up the name.

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

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.

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 = [{, UTF8-Byte-Codes}]
```

The string is a sequence of bytes representing the UTF8 code of the string. The number of bytes is specified in the ‘count’ field of the Item.

There is no trailing zero.

Filler data may be present.

Sequence

Contains a sequence of other items.

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

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

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

Filler data may be present.

Array

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

```
Array = Element-Descriptor [{, Element }]
Element-Descriptor =
      Element-Header,  (* 4 bytes *)
      Element-Length   (* 4 bytes the number of bytes in the 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.

The elements in the array are coded as follows:

Type description	Single Element Coding
Null	1 Byte (always 0)
Bool	1 Byte (0 or 1 depending on value)
Int8	1 Byte
Int16	2 Bytes
Int32	4 Bytes
Int64	8 Bytes
UInt8	1 Byte
UInt16	2 Bytes
UInt32	4 Bytes
UInt64	8 Bytes
Float32	4 Bytes

Type description	Single Element Coding
Float64	8 Bytes
String	Count for the number of bytes used to encode the string in UTF8 as an UInt32 (4 bytes) Followed by the UTF8 coding bytes
Array	Count for the number of elements in the array followed by the array value field. (As described elsewhere in this section)
Dictionary	Count for the number of items in the dictionary followed by the items.
Sequence	Count for the number of items in the dictionary followed by the items.
Binary	Count for the number of bytes following as an UInt32 (4 bytes) Followed by the bytes.

Note 1: The element is always addressed through the index, even though the Array, Dictionary and sequence may have a name.

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 = [{,byte}]]
```

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

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

- 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