BRBON

V 0.6-beta

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 recognized 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.

Strictly speaking BRBON only refers to the format specification, not an implementation. However a SWIFT reference API for v0.4 is offered for free under the same name. An ADA implementation for v0.6 is in the works.

Features:

- 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 all 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 4GB (UInt32.max). (The unfinished Block specification will include the capability for larger structures)

At this point in time BRBON specification is considered usable but incomplete. The ‘Item’ specification has been completed in v0.4 the ‘Block’ specification has been added in v0.6. V0.4 can be used for local storage needs and data/file exchange between platforms of same endianness.

Two alternatives for BRBON have been considered: BSON and BinSON. However both were found lacking with respect to the capabilities for high speed (vectored) 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
2021-02-19	V0.5	Restricted the use of name characters to ASCII in the range 0x20..0x7E (Including the names of columns in a table) Initial value of a CRC-32 is set to -1 instead of 0. Not backward compatible to V0.4
2021-03-07	V0.6-draft	Added rudimentary block specification (Block type 1)
2021-05-10	V0.6-beta	Block specification in beta stadium. Initial value of all CRC-32 are now -1 instead of 0. Not backward compatible to V0.5

Table of Contents

BRBON

All data in BRBON is stored as binary data. If the receiving system uses the same endianness as the sending system, data can be used directly without the need for parsing or conversion.

Typically a BRBON API will provide byte-swapping where needed.

In the format, data is wrapped in a small containers called “Items”. There are items available for all primitive data types. Examples: Integer_8, Float_32, String, Binary, Dictionary, Array etc.

Like in JSON, the top level item will often be a container type, like an Array or Dictionary.

Most items will have a name or an index to facilitate storage and retrieval at a “path”. The path is a sequence of names and indices denoting an item. The only exception to this rule is the top level item with can be unnamed and cannot have an index.

It depends on the API how the path elements are separated. In this document the slash (“/”) character is used.

Example: “Books/15/Title” for the 15th title in a books collection.

An item contains all support data to manage the data (possibly other items) inside it. As such an overhead is associated with each stored data item. The minimum size for an item is 16 bytes, including the stored value itself. The maximum size depends on the size of the data stored inside it, but the byte count specifier limits the total size of an item to 2^32 – 1 (about 4 GByte). To store data larger than this the next higher level object “Block” is used.

Note that the BRBON format itself always aligns blocks and items on 8 byte boundaries. Where necessary empty bytes are inserted. This guarantees that data access is always aligned which improves speed.

BRBON v0.4 was implemented as a SWIFT API and is fully usable for storage/transfer between same endianness machines.

BRBON v0.6 improves on v0.4 by adding the block specification. Backwards compatibility was broken to v0.5 by changing the seed value for all CRC-32 values (to allow for detection of missing leading zero’s).

The block specification adds the possibility for BRBON APIs to manage multiple blocks (containing each one or more top level items). Hence the path information is no longer sufficient. Blocks support this through a path-prefix construction. In addition further selection criteria may be used: block-origin and block-identifier.

The block specification includes the endianness specification and thus allows exchange of blocks between different endianness machines.

Blocks wrap Item(s) and thus add to the overhead. The theoretical minimum block size is 72 bytes, though more often a minimum of 80 bytes will be needed. However this overhead is only incurred once per “file”.

Like Items, the maximum Block size is limited to about 4 Gbytes. Larger data structures are planned to be supported through special item types (to be introduced later) that cross block boundaries.

Block

The purpose of a block header is storage and retrieval as well as the support of data structures larger than 2^32 bytes and to allow for streaming data.

Optionally the block header as well as the data in the block may be encrypted to facilitate proof of authenticity.

Several APIs can be constructed on top of the block specification at different levels of complexity:

At the lowest level is an API that reads a single file and accesses the items in the block by using a path consisting of strings and indices. No support from the block header is needed for this type of API.

At a higher level an API may be able to open several files at once and needs a way to distinguish between files. For this case the block-header supports a path prefix that is prepend to all item paths in the block. This prefix also consists of strings and indices and uses a notation as in this example: “games/14”. The slash is used as separator. (Note: Using the filename as a prefix was considered but rejected as the end-user can change file names at will.)

To guarantee authenticity encryption is used. A block-header can specify the location (URL) of a public key that has to be used to decrypt the items in the block as well as the encrypted copy of the block header. By comparing the decrypted block header with the encrypted block header certainty is achieved that the key from the proper location is used. The same key is then used to decrypt the items in the block.

To allow streaming data a block is started with 4 synchronization bytes. To validate that a block header is present the byte count of the block has to be read from the assumed position after the synchronization bytes. If the byte count is higher than the minimum block size it is used to read the entire header and calculate a checksum over the header (excluding the provided checksum at the end). This checksum is compared with the checksum at the end of the header. If the values match, the block is deemed to be a valid block. Though it can of course still contain invalid values if the block creator has made errors. If the calculated checksum does not match the checksum in the header, scanning for a renewed occurrence of the synchronization bytes should resume at the byte following the now discarded synchronization bytes.

A block can use either big endian or little endian coding. To distinguish between these the 4th synchronization byte is different.

A `Block` is defined as follows:

```
Block =
      Block_Header
      [,Unencrypted] | [,Encrypted]
      [,Block_Footer]
```

A block is defined as a block-header followed by either unencrypted content or encrypted content and can be followed by a block-footer.

Of these only the `Block_Header` is mandatory. Whether or not a footer is present depends on the block type.

Block Header

A `Block_Header` is defined as follows:

```
Block_Header =
```

```
Block_Synchronization_Byte_1,              (* at 0x00, 1 Byte *)
Block_Synchronization_Byte_2,              (* at 0x01, 1 Byte *)
Block_Synchronization_Byte_3,              (* at 0x02, 1 Byte *)
Block_Synchronization_Byte_4,              (* at 0x03, 1 Byte *)
Block_Type,                                (* at 0x04, 2 Bytes *)
Reserved_1,                                (* at 0x06, 2 Bytes *)

Block_Byte_Count,                          (* at 0x08, 4 Bytes *)
Block_Header_Byte_Count,                   (* at 0x0C, 2 Bytes *)
Block_Encrypted_Header_Byte_Count,         (* at 0x0E, 2 Bytes *)

Block_Origin_CRC16,                        (* at 0x10, 2 Bytes *)
Block_Identifier_CRC16,                    (* at 0x12, 2 Bytes *)
Block_Extension_CRC16,                     (* at 0x14, 2 Bytes *)
Block_Path_Prefix_CRC16,                   (* at 0x16, 2 Bytes *)

Block_Origin_Byte_Count,                   (* at 0x18, 1 Byte *)
Block_Identifier_Byte_Count,               (* at 0x19, 1 Byte *)
Block_Extension_Byte_Count,                (* at 0x1A, 1 Byte *)
Block_Path_Prefix_Byte_Count,              (* at 0x1B, 1 Byte *)
Block_Origin_Offset,                       (* at 0x1C, 2 Bytes *)
Block_Identifier_Offset,                   (* at 0x1E, 2 Bytes *)

Block_Extension_Offset,                    (* at 0x20, 2 Bytes *)
Block_Path_Prefix_Offset,                  (* at 0x22, 2 Bytes *)
Reserved_2,                                (* at 0x24, 4 Bytes *)

Block_Target_List_Byte_Count,              (* at 0x28, 2 Bytes *)
Block_Target_List_Offset,                  (* at 0x2A, 2 Bytes *)
Block_Public_Key_URL_Byte_Count,           (* at 0x2C, 2 Bytes *)
Block_Public_Key_URL_Offset,               (* at 0x2E, 2 Bytes *)

Block_Creation_Timestamp,                  (* at 0x30, 8 Bytes *)
Block_Modification_Timestamp,              (* at 0x38, 8 Bytes *)
Block_Expiry_Timestamp,                    (* at 0x40, 8 bytes *)

Block_Type_Dependent_Header,               (* N*8 Bytes *)

Block_Header_Field_Storage,                (* M*8 Bytes *)

Reserved_3,                                (* at BBC-7, 6 Bytes *)
Block_Header_CRC_16,                       (* at BBC-1, 2 Bytes *)
```

Block Synchronization Bytes

The first of a series of bytes that allow identification of the possible start of a block header in a random stream of data.

Name	Type	Value
Block_Synchronization_Byte_1	Uint 8	0x96
Block_Synchronization_Byte_2	Uint 8	0x7F

`Block_Synchronization_Byte_3`	Uint 8	0x81
`Block_Synchronization_Byte_4`	Uint 8	0x5A = Little endian 0xA5 = Big endian

The sequence of the block synchronization bytes is fixed and independent of the endianness of the block. The last synchronization byte determines the endianness of multi-byte values used in the block (header, content and footer)

A block is identified in two steps:

First the block synchronization bytes must appear in the correct order.

Secondly the block-header-byte-count will be read and a CRC-16 will be calculated over the entire block header excluding the `Block_Header_CRC_16`. (Using the reverse polynomial 0x8005 (= 0xA001 reversed) and 0 as the start seed).

If the calculated CRC-16 is not equal to the `Block_Header_CRC_16` then the block synchronization bytes do not indicate the start of a block header. Scanning for block synchronization bytes should continue at the first byte after the evaluated block start markers. If the byte sequence reader is synchronized (as will often be the case for files that contain only a single block) then the failure to verify a block start is an error.

The above process may be preempted if the block header byte count is smaller than the minimum block header count of 72 bytes.

The maximum header count is 2^16 – 1 = 65,535 bytes (64 KByte).

Block Type

The type of content contained in this block.

Name	Type	Valid Range
`Block_Type`	Uint 16	See table below

Valid Range:

Type	Purpose	Uses Block Byte Count	Uses Footer
1	Data storage with a single top level item.	Yes.	Yes

No other block types are defined at this time.

If this table specifies ‘Yes’ for either Uses_Block_Byte_Count or Uses_Footer, then these values MUST be used and present. Absence would violate the block usability and should lead to rejection of the block.

If this table specifies ‘Optional’ for either Uses_Block_Byte_Count or Uses_Footer, then these values may be used and present. Other conditions may apply as specified for the block type.

Reserved

Currently not used, test for zero and reject block usage if non zero to ensure upward compatibility.

Name	Type	Value
Reserved_1	Uint 16	0x0000
Reserved_2	Uint 32	0x0000_0000
Reserved_3	Uint 48	0x0000_0000_0000

Block Byte Count

The total number of bytes in this block.

Name	Type	Valid Range
Block_Byte_Count	Uint 32	64 .. 2^32-2 (4,294,967,294)

If the Block_Byte_Count is not used, it should be set to 0xFFFF_FFFF

If a Block_Byte_Count is deemed to be invalid, the block should be considered unusable.

Block Header Byte Count

The number of bytes in the unencrypted block header.

Name	Type	Valid Range
Block_Header_Byte_Count	Uint 16	72 .. 65,535

This count shall not include the bytes used by the encrypted header block (if present).

If a count of less than the minimum size is encountered the block must be considered invalid.

Block Encrypted Header Byte Count

The number of bytes in the encrypted header.

Name	Type	Valid Range
Block_Encrypted_Header_Byte_Count	Uint 16	0 .. 65,535

If this number is zero, the public key URL field must be empty.

This block should be considered unusable if this value is zero and a public key URL field is present.

This block should be considered unusable if this value is non-zero and no public key URL field is present.

Block ... CRC16

Name	Type	Valid Range
Block_Source_CRC_16	Uint 16	all
Block_Identifier_CRC_16	Uint 16	all
Block_Extension_CRC_16	Uint 16	all
Block_Path Prefix_CRC_16	Uint 16	all

These CRC values can be used to quickly discard blocks that do not match a search criteria.

The values are calculated over the byte sequence of their corresponding fields using using the reverse polynomial 0x8005 (= 0xA001 reversed) and 0 as the start seed.

Block Origin

Name	Type	Valid Range
`Block_Origin_Byte_Count`	Uint 8	0 .. 255
`Block_Origin_Offset`	Uint 16	72 .. 65,535

The block origin field is used for block selection. It should be a unique value. It is recommended to use a domain name that is under control of the block creator. If additional specialization is needed, consider using the block extension field or the block path prefix field.

Note that this is indicative only and does not guarantee that the block was obtained from the specified source or can be obtained from it.

If source authenticity is needed, use encryption and the block public key URL field.

It is recommended to use UTF-8 encoding in this field.

The byte count specifies how many bytes there are in this field.

If the field is not used the byte-count should be set to zero.

The offset to this field is counted from the first block synchronization byte, starting with zero.

Block Identifier

Name	Type	Valid Range
`Block_Identifier_Byte_Count`	Uint 8	0 .. 255
`Block_Identifier_Offset`	Uint 16	72 .. 65,535

The block identifier field is used for block selection in much the same way a filename is used. However the block identifier is stored inside the block and thus not subject to filename changes by an end user. Using it however will require a BRBON API with block management capabilities.

It is recommended to use UTF-8 encoding in this field.

If no end user interaction is needed with the content of this field, it is recommended to use a version 4 UUID value.

The byte count specifies how many bytes there are in this field.

If the field is not used the byte count should be set to zero.

The offset to this field is counted from the first block synchronization byte, starting with zero.

Block Extension

Name	Type	Valid Range
`Block_Extension_Byte_Count`	Uint 8	0 .. 255
`Block_Extension_Offset`	Uint 16	72 .. 65,535

The block extension field can be used to specify which application should be able to read the content in much same way a filename extension is used. However the block extension field is stored inside the block

and thus not subject to changes invoked by an end user. Using it however will require a BRBON API with block management capabilities.

No facilities exist to guarantee uniqueness.

It is recommended to use UTF-8 encoding in this field.

The byte count specifies how many bytes there are in this field.

If the field is not used the bytecount should be set to zero.

The offset to this field is counted from the first block synchronization byte, starting with zero.

Block Path Prefix

Name	Type	Valid Range
`Block_Path_Prefix_Byte_Count`	Uint 8	0 .. 255
`Block_Path_Prefix_Offset`	Uint 16	72 .. 65,535

The block path prefix field contains a sequence of bytes (usually a readable string) that is to be included before the path of any item in the block.

The ASCII-slash (0x2F, “/”) is used as a separator character between sub sequences.

A sequence of slash characters must be treated as a single slash. Note that this makes it impossible to use a slash as part of a prefix sub sequence.

A prefix consisting only of slash characters must be treated as an empty prefix.

The byte count specifies how many bytes there are in this field.

If the field is not used the byte count should be set to zero.

The offset to this field is counted from the first block synchronization byte, starting with zero.

Block Target List

Name	Type	Valid Range
`Block_Target_List_Byte_Count`	Uint 16	0 .. 65,535
`Block_Target_List_Offset`	Uint 16	0 .. 65,535

A block can be targeted to specific receivers. These receivers are listed in the block target list.

The target list field contains a sequence of byte sequences separated by the ASCII-blank (0x20) character code.

There are no other requirements on the content of this list. The content has to be agreed between the distributor and consumer.

The byte count specifies how many bytes there are in this field.

If the field is not used the byte count should be set to zero.

The offset to this field is counted from the first block synchronization byte, starting with zero.

Block Public Key URL

Name	Type	Valid Range
`Block_Public_Key_URL_Byte_Count`	Uint 16	0 .. 65,535
`Block_Public_Key_URL_Offset`	Uint 16	0 .. 65,535

The public key URL field allows authentication of a block.

If authentication is used the block header is followed by an encrypted block header that upon decryption with the key at the specified URL yields an exact copy of the block header (including the `Block_Header_CRC_16`). Only when all fields of the decrypted block header are identical to those in the unencrypted bock header is the block header authenticated.

Note that when authentication is used the items contained in the block are also encrypted with the same public key and have to be decrypted before use.

The public key URL must use UTF-8 coding.

The key must initially be fetched from the specified location, however a BRBON API may decide to store the key locally for subsequent use.

The byte count specifies how many bytes there are in this field.

If the field is not used the byte count should be set to zero.

The offset is counted from the first block synchronization byte starting with zero.

Block Timestamps

Name	Type	Valid Range
`Block_Creation_Timestamp`	Uint 64	All
`Block_Modification_Timestamp`	Uint 64	All
`Block_Expiry_Timestamp`	Uint 64	All

All times and duration's are in milliseconds since 1 Jan 1970.

Creation refers to the time this block was created.

Modification time refers to the time of the most recent update of an item in this block.

Expiry time refers to the time after which this block should no longer be used.

Impossible combinations of timestamps should prevent usage of the block.

Block Type Dependent Header

Name	Type	Valid Range
`Block_Type_Dependent_Header`	-	-

Depending on the type of block, additional header fields may be defined.

Currently only block type 1 is defined which does not have a type dependent header.

The total byte count for this area is a multiple of 8 bytes. Unused space should be set to zero.

Block Header Field Storage

Name	Type	Valid Range
`Block_Header_Field_Storage`	-	-

All variable type fields specified by a byte-count and offset in the block header will have their fields located in this area as specified by their offset parameter.

The total byte count for this area is a multiple of 8 bytes. Unused space should be set to zero.

Block Header CRC-16

The header CRC-16 is calculated using the reverse polynomial 0x8005 (= 0xA001 reversed) and 0 as the start seed.

Name	Type	Value
`Block_Header_CRC_16`	Uint 16	0 .. 65535

Unencrypted

The `Unencrypted` part is defined as follows:

Unencrypted = Item [,Item]

That is, it can consist of 1 or more Item. (These are defined later)

Encrypted

The `Encrypted` part is defined as follows:

```
Encrypted = Encrypted_Header [,Encrypted_Item]
```

It consists of an encrypted header and zero or more encrypted items.

Encrypted Header

The encrypted header starts with the byte following the CRC-16 of the unencrypted header. It continues for block-encrypted-header-byte-count. This entire sequence should be decrypted with the key from the location specified by the URL in the block-public-key-location.

Once the encrypted header is decrypted it should be compared with the unencrypted header. If there are no differences, the key is authenticated.

The same public key should then be used to decode the block content (items).

Note that a block footer that may be present is not encrypted.

Block Footer

The block footer content depends on the block type.

All block footers will have a byte count of a multiple of 8 bytes.

Block Type 1

Name	Offset (Hex)	Type	Valid Range
Reserved	0x00	Uint 32	0x0000_0000
Block_Content_CRC_32	0x04	Uint 32	0 .. 2^32 - 1

The content CRC-32 is calculated only over the item contained in the block.

The CRC-32 is calculated with a seed value of -1.

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
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-ASCII-Byte-Count,(* 1 byte, the number of used ASCII bytes *)
      Name-ASCII-Code       (* ASCII bytes range 0x20..0x7E *)
      [, 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-ASCII-Byte-Count`: This is the number of bytes used by the ASCII code that constitutes the name.

`Name-ASCII-Code`: The ASCII code that can be converted into a valid string. Note that each ASCII characters must be in the range 0x20 .. 0x7E.

`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 ASCII 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 ASCII 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 -1.

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	Count
0	Byte Count	UInt32	4
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 -1.

The layout is as follows:

Offset	Content	Units	Count
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
5	Zero	Bytes	3
8	Element-Count	UInt32	4
12	Element-Byte-Count	UInt32	4
16	Array	Element	Element-Count * Element-Byte-Count
16 + Element-Count * 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 (columns) are only present once.

```
Table =
      RowCount,                  (* 4 bytes, UInt32 *)
      ColumnCount,               (* 4 bytes, UInt32 *)
      FieldsOffset,              (* 4 bytes, UInt32 *)
      RowByteCount               (* 4 bytes, UInt32 *)
      [{, ColumnDescriptor }]    (* 16 bytes * number of columns *)[{,
      ColumnName }]              (* N, 8 byte boundary, * num of cols *)
      [{, Row }]                 (* N, 8 byte boundary, * num of cols *)

ColumnDescriptor =
      ColumnNameCrc16,           (* 2 bytes *)
      ColumnNameFieldByteCount,  (* 1 bytes *)
      ColumnFieldType,           (* 1 bytes *)
      ColumnNameFieldOffset,     (* 4 bytes *)
      ColumnFieldOffset,         (* 4 bytes *)
      ColumnFieldByteCount       (* 4 bytes *)

ColumnName = Ascii-Byte-Count {, Ascii-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)

`FieldsOffset`: 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 ASCII-byte code that is the name of the column. Using the reverse polynomial 0x8005 (= 0xA001 reversed) and 0 as the start seed.

`ColumnNameFieldByteCount`: 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)

`ColumnFieldType`: The type of item stored in this column. ColumnNameUtf8Offset: The offset from the start of the value area to the sequence of ASCII byte codes that specify the name of the column.

`ColumnNameFieldOffset`: The offset from the start of the value area where this column ’s name field is located.

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

`ColumnFieldByteCount`: 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 ASCII 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	Fields Offset	UInt32	4
12	Row Byte Count	UInt32	4
16	Column Descriptors	Column Descriptor	16 * Column Count

16 + 16 * Column Count	Column Names	Column Name	As needed
Rows Offset	Rows (Fields)	Rows	Row Count * Row Byte Count
Rows Offset + Row Count * 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 Field Offset	UInt32	4
8	Column Field Offset	UInt32	4
12	Column Field Byte Count	UInt32	4

The layout of the Column Name Fields is as follows:

Offset	Content	Units	Bytes
0	Column Name Byte Count	UInt8	1
4	Column Name ASCII 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 layout is as follows:

Offset	Content	Units	Bytes
0	Red	UInt8	1
1	Green	UInt8	1
2	Blue	UInt8	1

3	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 String	UTF8	Family Size
6 + Family Size	Name String	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