BASON: Binary Adaptation of Structured Object Notation

Version: 0.1
Status: Draft
Date: February 2026

1. Introduction

BASON is a minimalistic binary encoding for JSON-compatible data. It encodes values as self-describing Tag-Length-Value (TLV) records using only ten tag bytes derived from the mnemonic "BASON" in uppercase and lowercase forms.

BASON supports two modes of operation: nested (hierarchical containers) and flat (path-keyed leaf records), and both may coexist within a single stream depending on context.

1.1. Design Goals

- Minimal tag set (10 values)
- Self-describing records without external schema
- Encoder and decoder implementable in under 200 lines each
- Support for both hierarchical and flat key-value representations
- Configurable strictness via bitmask

1.2. Terminology

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119.

2. Data Model

BASON encodes the JSON data model with the following value types:

Type	Description
Boolean	true, false, or null
Array	Ordered sequence of indexed values
String	UTF-8 text
Object	Unordered collection of named values
Number	Numeric value in text representation

3. RON64 Encoding

Array indices are encoded using RON64, a base-64 numeral system with the following alphabet:

```
Position:  0       1       2       3
           012345678901234567890123456789012345678901
Alphabet:  0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ_abc
           defghijklmnopqrstuvwxyz~
```

The complete alphabet string:

```
0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ_abcdefghijklmnopqrstuvwxyz~
```

Positions	Characters
0-9	0 through 9
10-35	A through Z
36	_
37-62	a through z
63	~

The alphabet is in strict ASCII order (0x30..0x39, 0x41..0x5A, 0x5F, 0x61..0x7A, 0x7E).

RON64 numbers are encoded in big-endian digit order (most significant digit first), yielding lexicographic sortability: a byte-wise comparison of two RON64-encoded numbers produces the correct numeric ordering.

3.1. RON64 Examples

Decimal	RON64
0	0
9	9
10	A
36	_
63	~

Decimal	RON64
64	10
100	1W
4095	~~
4096	100

4. Record Format

Every BASON record begins with a one-byte tag, followed by length fields and then key and value data.

4.1. Tag Bytes

```
Lowercase (short form):  b  a  s  o  n
                         0x62 0x61 0x73 0x6F 0x6E

Uppercase (long form):   B  A  S  O  N
                         0x42 0x41 0x53 0x4F 0x4E
```

Tag	Type	Form
b	Boolean	Short
B	Boolean	Long
a	Array	Short
A	Array	Long
s	String	Short
S	String	Long
o	Object	Short
O	Object	Long
n	Number	Short
N	Number	Long

4.2. Short Record Layout

Short records (lowercase tag) use a single byte to encode both the key length and value length. Maximum key length: 15 bytes. Maximum value length: 15 bytes.

```
+------+------------+-----+-------+
| tag  | lengths    | key | value |
| 1B   | 1B         | 0-15B | 0-15B |
+------+------------+-----+-------+
```

The lengths byte encodes:

- Bits 7-4 (upper nibble): key length in bytes (0-15)
- Bits 3-0 (lower nibble): value length in bytes (0-15)

4.3. Long Record Layout

Long records (uppercase tag) use five bytes for lengths. Maximum key length: 255 bytes. Maximum value length: 2^32 - 1 bytes.

```
+------+-----------+---------+-----+-------+
| tag  | val_len   | key_len | key | value |
| 1B   | 4B LE     | 1B      | 0-255B | ... |
+------+-----------+---------+-----+-------+
```

- val_len: value length as a 4-byte unsigned little-endian integer
- key_len: key length as a 1-byte unsigned integer

4.4. Key Encoding

The interpretation of the key field depends on context:

Context	Key contains
Inside an Array (A/a)	RON64-encoded array index (big-endian)
Inside an Object (O/o)	UTF-8 string (the field name)
Root element	UTF-8 path (e.g., `key/nested/inner`)
Flat mode leaf record	UTF-8 path (e.g., `users/0/name`)

The root element or flat-mode records MAY have an empty key (key length of zero).

4.5. Value Encoding

Type	Value payload
Boolean	`true`, `false` (UTF-8 text), or empty for null
String	UTF-8 text
Number	UTF-8 text representation of the numeric value
Array	Concatenated child records (recursive)
Object	Concatenated child records (recursive)

For container types (Array, Object), the value length field gives the total byte length of all concatenated child records. This enables skipping over a container without parsing its contents.

5. Nested Mode

In nested mode, containers (Array and Object records) hold child records as their value payload. The structure mirrors the JSON tree.

5.1. Nested Encoding Example

JSON:

```json
{"name": "Alice", "scores": [95, 87]}
```

BASON (nested, annotated):

```
O                   tag: Object (long form)
05 00 00 00             val_len: total child bytes (little-endian)
00                   key_len: 0 (root, empty key)
                    --- children of root object ---
s                   tag: String (short form)
45                   key_len=4, val_len=5
6E 61 6D 65             key: "name"
41 6C 69 63 65           value: "Alice"
a                   tag: Array (short form)  -- or A if children exceed 15B
...                 key: "scores", value: children
  n                 tag: Number (short form)
  12                 key_len=1, val_len=2
  30                 key: "0" (RON64 index 0)
  39 35                value: "95"
  n                 tag: Number (short form)
  12                 key_len=1, val_len=2
  31                 key: "1" (RON64 index 1)
  38 37                value: "87"
```

6. Flat Mode

In flat mode, no container records are emitted. Each leaf value is encoded as an independent record whose key is the full path from the root, using `/` as a separator.

6.1. Flat Encoding Example

The same JSON as above, in flat mode:

```
s  45  6E616D65  416C696365       "name" = "Alice"
n  82  73636F7265732F30  3935      "scores/0" = "95"
n  82  73636F7265732F31  3837      "scores/1" = "87"
```

6.2. Mixed Mode

A single stream MAY contain both container records and flat leaf records. The interpretation depends on the application context. For example, a stream might use a container record for a top-level object while inlining deeply nested leaves as path-keyed flat records.

7. Strictness Levels

BASON defines a bitmask of strictness options. Encoders SHOULD advertise which bits they guarantee. Decoders SHOULD accept a bitmask specifying which rules to enforce. Two systems MAY negotiate common strictness by ANDing their respective masks.

7.1. Strictness Bits

```
Bit  0 (0x001):  Shortest encoding required. Encoder MUST use short
                 form when both key and value fit in 15 bytes.

Bit  1 (0x002):  Canonical number format. No leading zeros (except
                 the value "0" itself), no leading '+', no trailing
                 decimal point, no scientific notation.

Bit  2 (0x004):  Valid UTF-8 required. All keys and values MUST be
                 well-formed UTF-8.

Bit  3 (0x008):  No duplicate keys. Object containers MUST NOT
                 contain multiple records with the same key.

Bit  4 (0x010):  Contiguous array indices. Array containers MUST
                 have indices forming a contiguous range starting
                 from 0.

Bit  5 (0x020):  Ordered array indices. Records within an array
                 container MUST appear in ascending index order.

Bit  6 (0x040):  Sorted object keys. Records within an object
                 container MUST appear in lexicographic order of
                 their UTF-8 key bytes.

Bit  7 (0x080):  Canonical boolean text. Boolean values MUST be
                 exactly "true", "false", or "" (empty for null).

Bit  8 (0x100):  Minimal RON64. Array indices MUST use the shortest
                 RON64 representation (no leading zero digits).

Bit  9 (0x200):  Canonical path format. Path keys MUST NOT contain
                 leading slashes, trailing slashes, or consecutive
                 slashes.

Bit 10 (0x400):  No flat/nested mixing. A single stream MUST use
                 exclusively nested mode or exclusively flat mode,
                 not both.
```

7.2. Predefined Levels

```
Permissive  = 0x000  (0)
  Accept anything that can be mechanically decoded.

Standard    = 0x1FF  (511, bits 0-8)
```

Well-formed data suitable for interchange between systems.

Strict = 0x7FF (2047, bits 0-10)
Canonical encoding suitable for content hashing, signing,
deduplication, and structural comparison.

7.3. Strictness Negotiation

When two systems exchange BASON data, they MAY negotiate strictness:

- Encoder advertises a bitmask of guarantees it provides.
- Decoder advertises a bitmask of requirements it enforces.
- The encoder's mask MUST be a superset of the decoder's mask (i.e., `encoder_mask & decoder_mask == decoder_mask`).

8. Canonical Encoding

When Strict mode (0x7FF) is in effect, the encoding of any given JSON value is fully deterministic. Two encoders producing BASON from the same logical JSON value MUST produce byte-identical output. This enables:

- Content-addressable storage (hash of encoding identifies the value)
- Binary comparison of encoded values for equality
- Merkle tree construction over BASON records
- Digital signature verification

To achieve canonical encoding, all of the following MUST hold:

1. Shortest form is used (bit 0).
2. Numbers are in canonical text form (bit 1).
3. UTF-8 is valid (bit 2).
4. No duplicate keys (bit 3).
5. Array indices are contiguous from 0 (bit 4) and ordered (bit 5).
6. Object keys are sorted lexicographically (bit 6).
7. Booleans use exact canonical text (bit 7).
8. RON64 indices are minimal (bit 8).
9. Paths are in canonical form (bit 9).
10. No flat/nested mixing (bit 10).

9. Size Limits

Field	Short form	Long form
Key length	15 bytes	255 bytes
Value length	15 bytes	4 GiB - 1
Nesting depth	Unlimited (application-defined)	
Record overhead	2 bytes	6 bytes

10. ABNF Grammar

```abnf
```

```
bason-stream  = *record

record        = short-record / long-record

short-record  = short-tag lengths key value
long-record   = long-tag val-len key-len key value

short-tag     = %x62 / %x61 / %x73 / %x6F / %x6E
              ; b a s o n

long-tag      = %x42 / %x41 / %x53 / %x4F / %x4E
              ; B A S O N

lengths       = OCTET
              ; upper nibble = key length (0-15)
              ; lower nibble = value length (0-15)

val-len       = 4OCTET
              ; unsigned 32-bit little-endian

key-len       = OCTET
              ; unsigned 8-bit

key           = *OCTET
              ; length determined by lengths or key-len field
              ; contents depend on context (see Section 4.4)

value         = *OCTET
              ; length determined by lengths or val-len field
              ; contents depend on type (see Section 4.5)
              ; for containers: concatenated child records
```

11. Media Type

```
Type name:          application
Subtype name:       bason
Required parameters: none
Optional parameters:
  strictness  - decimal integer bitmask (0-2047)
  mode        - "nested", "flat", or "mixed" (default: "mixed")
File extension:     .bason
```

12. Comparison with Existing Formats

Property	BASON	CBOR	BSON	MessagePack
Tag values	10	8 major	~20	~30
Container skip	Yes	No*	Yes	No*
Numbers	Text	Binary	Binary	Binary
Max key length	255 B	Unlimited	Unlimited	Unlimited
Null-terminated keys	No	No	Yes	No
Flat path mode	Yes	No	No	No
Canonical form	Defined	Optional	No	No
Integer overhead (1)	2-3 B	1 B	5 B	1 B

* CBOR and MessagePack support definite-length containers with element counts, enabling skip by parsing children, but not by byte offset.

13. Implementation Considerations

13.1. Encoding

Encoders producing container records MUST know the total byte length of child records before writing the container's length field. Two strategies:

1. **Two-pass**: serialize children to a buffer, measure, then write the container header followed by the buffer.
2. **Backpatching**: write a placeholder length, serialize children, then seek back and overwrite the length.

For flat mode, no container records are produced, and records may be written in a single streaming pass.

13.2. Decoding

A decoder reads the tag byte, determines short or long form, reads the length fields, then reads the key and value. For container types, the value payload is decoded recursively as a sequence of child records.

A decoder MAY skip over any record without parsing its value by reading only the tag and length fields and advancing by the

indicated number of bytes.

13.3. Numbers as Text

Encoding numbers as UTF-8 text avoids integer width decisions and floating-point representation issues. The tradeoff is larger encoded size for numeric-heavy data compared to binary encodings. When canonical encoding (bit 1) is in effect, the text representation is deterministic, making comparison and hashing reliable.

Applications requiring high-performance numeric processing SHOULD parse number text into native types on decode and MAY cache parsed representations.

14. Security Considerations

- Decoders MUST validate that declared lengths do not exceed available data to prevent buffer overflows.
- Deeply nested containers MAY cause stack overflow in recursive decoders. Applications SHOULD impose a maximum nesting depth.
- The flat mode path key could be used for directory traversal attacks (e.g., `../../etc/passwd`). Applications interpreting paths MUST sanitize them.
- In Permissive mode, invalid UTF-8 sequences are passed through. Applications handling text SHOULD validate UTF-8 independently if security-relevant.

15. References

- RFC 2119: Key words for use in RFCs
- RFC 8949: Concise Binary Object Representation (CBOR)
- RFC 8259: The JavaScript Object Notation (JSON) Data Interchange Format
- BSON Specification: https://bsonspec.org/