Internet Engineering Task Force
Internet-Draft
Intended status: Standards Track
Expires: 1 October 2026

A. Luis Bustamante
Internet of Thinger SL

30 March 2026

Packed Sensor Object Notation (PSON)
draft-bustamante-pson-00

Abstract

PSON (Packed Sensor Object Notation) is a compact binary data encoding format designed for IoT environments where bandwidth, memory, and processing power are constrained. PSON efficiently represents the data types commonly found in sensor telemetry and device interaction: integers, floating-point numbers, booleans, strings, binary blobs, arrays, and key-value maps. It achieves significant size reductions over JSON (40-75% for typical IoT payloads) through inline small value encoding and variable-length integers. PSON is self-describing (no external schema required for decoding) and designed for minimal implementation complexity on microcontrollers.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at https://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on 1 October 2026.

Copyright Notice

Table of Contents

1. Introduction

1.1. Purpose

PSON is a binary serialization format that provides a compact, self-describing encoding for structured data. It is designed as a drop-in binary replacement for JSON in environments where bandwidth and processing resources are limited -- particularly IoT devices, embedded systems, and constrained networks.

PSON has been deployed in production IoT systems since 2015 as the native encoding format of the Thinger.io platform, processing sensor data across thousands of connected devices. This specification formalizes the encoding that has been validated through a decade of production use.

1.2. Background

IoT devices frequently exchange structured data: sensor readings, configuration parameters, device metadata, and command payloads. JSON [RFC8259] is the de facto standard for structured data interchange but imposes significant overhead on constrained devices:

* *Verbose encoding:* Field names and values are repeated as text in every message.
* *Parsing cost:* Text parsing requires string-to-number conversion and delimiter scanning.
* *Size inefficiency:* Common IoT values like small integers, booleans, and short strings are disproportionately expensive in JSON.

PSON addresses these issues through a binary encoding that preserves JSON's self-describing nature while dramatically reducing wire size and parsing complexity.

1.3. Relationship to CBOR

PSON shares structural similarities with CBOR [RFC8949]. Both use a tag byte with 3 bits for type identification and 5 bits for an inline value. However, PSON and CBOR make different design trade-offs reflecting different goals: CBOR is a general-purpose binary encoding designed for broad applicability; PSON is a minimal encoding designed specifically for integration with the IOTMP protocol on constrained IoT devices.

Protocol-Encoding Integration. PSON's primary design motivation is architectural. PSON and the IOTMP protocol share identical encoding primitives: the same tag-byte structure (3-bit type + 5-bit inline value) and the same varint encoding for variable-length integers. A single encoder/decoder implementation on a constrained device serves both the protocol framing layer and the application data layer. Adopting CBOR would require maintaining two independent type systems -- CBOR's major types alongside IOTMP's own field encoding -- increasing code size and complexity for no functional benefit on these devices.

Reduced Scope. CBOR is designed to cover a wide range of use cases through extensibility mechanisms that add implementation complexity:

* *Semantic tags* (CBOR major type 6): Over 350 IANA-registered tags for dates, bignums, decimal fractions, URIs, regular expressions, and more. Each tag number can be 1-9 bytes. Decoders must handle unknown tags gracefully. PSON has no semantic tag system -- the 8 wire types cover all data types needed for IoT applications.

* *Indefinite-length encoding*: CBOR allows arrays, maps, and strings of unknown length at encode time, terminated by a break stop code (0xFF). This roughly doubles the decoder's state machine complexity. PSON requires definite lengths, which is sufficient for IoT data that is typically fully known in memory before encoding.

* *Half-precision floats* (IEEE 754 binary16): CBOR supports 16-bit, 32-bit, and 64-bit floats, with a preferred serialization rule that requires testing each value against all three sizes. Most platforms lack native float16 support, requiring dedicated conversion routines. PSON supports only 32-bit and 64-bit floats -- the two sizes natively available on all target platforms.

* *Simple values*: CBOR defines a 256-value extensible space for simple values (currently: false, true, null, undefined, plus unassigned slots). PSON defines exactly three discrete values: false, true, and null.

* *Arbitrary map key types*: CBOR allows any data type as a map key (integers, arrays, nested maps). PSON restricts map keys to strings, matching JSON's object model and simplifying decoder implementation.

Wire Size. In terms of encoded size, PSON and CBOR produce comparable results for typical IoT data. PSON encodes unsigned integers 0-30 in a single byte (vs. CBOR's 0-23), providing a modest advantage for values 24-30. For most payloads, the size difference between PSON and CBOR is negligible. The justification for PSON is not superior compression -- it is the reduced implementation complexity and the shared encoding primitives with IOTMP.

Byte Ordering. CBOR encodes all multi-byte values in big-endian (network byte order). PSON uses little-endian for floating-point values, matching the native byte order of virtually all IoT microcontrollers (ARM Cortex-M, ESP32, RISC-V). See Section 7.

1.4. Relationship to MessagePack

MessagePack [MessagePack] is another binary encoding with similar goals. Like CBOR, it uses big-endian byte ordering and a more complex type system (with multiple fixed-width integer sizes and format families). PSON's simpler tag-byte design and varint-based overflow mechanism result in a smaller and more uniform implementation.

2. Terminology

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

3. Design Goals

* *Compactness:* Minimize encoded size for typical IoT data (sensor readings, configuration, commands).
* *Self-describing:* No external schema required for decoding. The wire type and structure are embedded in the data.
* *Simplicity:* Minimal implementation complexity. Encodable/decodable in a single pass with no lookahead.

* *JSON superset:* PSON represents all JSON data types (objects, arrays, strings, numbers, booleans, null) plus native binary data (wire type 5), enabling lossless round-trip conversion from JSON to PSON. Binary data has no JSON equivalent and requires application-level encoding (e.g., Base64) when converting to JSON.
* *Protocol integration:* Share encoding primitives (tag-byte structure, varint encoding) with IOTMP, enabling a single encoder/decoder to serve both protocol framing and application data.

4. Tag Byte Structure

Every PSON value begins with a single tag byte that encodes both the data type and, for small values, the value itself:

```
+-------------+--------------+
|  Wire Type  | Inline Value |
|  (bits 7-5) | (bits 4-0)   |
+-------------+--------------+
```

* *Wire Type* (3 most significant bits): Identifies the data type (0-7).
* *Inline Value* (5 least significant bits): For values 0-30, the value is stored directly in the tag byte, requiring no additional bytes. If the inline value is 31 (0x1F mask), the actual value follows the tag byte as a varint.

The tag byte formula is: tag = (wire_type << 5) | inline_value.

This design means that common small values (integers 0-30, short strings, small maps and arrays) are encoded in the minimum possible space.

5. Wire Types

Value	Name	Tag Range	Description
0	unsigned_t	0x00-0x1F	Unsigned integer. Inline 0-30 or varint.
1	signed_t	0x20-0x3F	Negative integer. Stored as absolute value.
2	floating_t	0x40-0x5F	IEEE 754 float (inline=0) or double (inline=1).
3	discrete_t	0x60-0x7F	Boolean or null. 0=false, 1=true, 2=null.

4	string_t	0x80-0x9F	UTF-8 string. Inline or varint = byte length.
5	bytes_t	0xA0-0xBF	Raw binary data. Inline or varint = byte length.
6	map_t	0xC0-0xDF	Key-value map. Inline or varint = entry count.
7	array_t	0xE0-0xFF	Ordered array. Inline or varint = element count.

Table 1

Wire types 0-7 are defined by this specification. No additional wire types can be defined (the 3-bit field is fully allocated).

6. Encoding Rules

6.1. Unsigned Integers (Wire Type 0)

Unsigned integers represent non-negative whole numbers.

* *Values 0-30:* Encoded in a single byte: (0 << 5) | value.
* *Values >= 31:* Tag byte is 0x1F, followed by the value encoded as a varint Section 8.

Examples:

Value	Encoded (hex)	Size
0	00	1 byte
5	05	1 byte
30	1E	1 byte
31	1F 1F	2 bytes
127	1F 7F	2 bytes
300	1F AC 02	3 bytes

Table 2

6.2. Signed Integers (Wire Type 1)

Negative integers are encoded using wire type 1 with the absolute value. Decoding: negate the stored value.

* *Values -1 to -30:* Single byte: (1 << 5) | abs(value).
* *Values <= -31:* Tag byte 0x3F, followed by the absolute value as a varint.

Zero and positive integers MUST use wire type 0 (unsigned_t). Wire type 1 is exclusively for negative values. Encoding zero as signed_t (tag byte 0x20) is invalid and a decoder MUST treat it as an error.

Examples:

Value	Encoded (hex)	Size
-1	21	1 byte
-15	2F	1 byte
-30	3E	1 byte
-31	3F 1F	2 bytes
-300	3F AC 02	3 bytes

Table 3

6.3. Floating-Point Numbers (Wire Type 2)

IEEE 754 [IEEE754] floating-point values. The inline value selects the precision:

Inline Value	Precision	Bytes Following Tag
0	32-bit float (IEEE 754 binary32)	4 bytes, little-endian
1	64-bit double (IEEE 754 binary64)	8 bytes, little-endian

Table 4

Inline values 2-30 are reserved for future use. Inline value 31 (varint extension) is not used for this wire type. A decoder that encounters inline value 31 or any reserved inline value (2-30) for this wire type MUST treat it as a decode error.

Negative zero: IEEE 754 defines negative zero (-0.0) as distinct from positive zero (+0.0). Encoders MUST encode -0.0 as a floating-point value (wire type 2), not as unsigned integer zero (wire type 0). Decoders MUST preserve the sign of zero when converting from PSON to IEEE 754.

NaN and Infinity: Encoders MUST encode NaN and Infinity values as 32-bit or 64-bit floats (wire type 2). These values MUST NOT be promoted to integers. Encoders SHOULD use a canonical NaN representation (quiet NaN with all-zero payload) when the specific NaN payload is not significant.

Examples:

Value	Encoded (hex)	Notes
3.14	40 C3 F5 48 40	32-bit float
3.141592653	41 38 E9 2F 54 FB 21 09 40	64-bit double

Table 5

6.4. Discrete Values (Wire Type 3)

Boolean and null values.

Inline Value	Meaning	Encoded (hex)
0	false	60
1	true	61
2	null	62

Table 6

Inline values 3-30 are reserved for future use. Inline value 31 (varint extension) is not used for this wire type. A decoder that encounters inline value 31 or any reserved inline value (3-30) for this wire type MUST treat it as a decode error.

6.5. Strings (Wire Type 4)

UTF-8 encoded text strings. The inline value (or subsequent varint if inline = 31) indicates the byte length of the string. The string bytes follow immediately, with no null terminator.

* *Strings of 0-30 bytes:* Single tag byte with inline length, followed by the string bytes.
* *Strings of >= 31 bytes:* Tag byte 0x9F, followed by a varint length, followed by the string bytes.

Implementations MUST encode strings as valid UTF-8. A decoder that encounters invalid UTF-8 sequences SHOULD treat it as a decode error.

Examples:

Value	Encoded (hex)	Size
"" (empty)	80	1 byte
"hi"	82 68 69	3 bytes
"temperature"	8B 74 65 6D 70 65 72 61 74 75 72 65	12 bytes

Table 7

6.6. Binary Data (Wire Type 5)

Raw binary (opaque byte sequences). Encoding is identical to strings: the inline value (or varint) indicates byte length, followed by the raw bytes.

* *0-30 bytes:* Single tag byte with inline length, followed by data.
* *>= 31 bytes:* Tag byte 0xBF, followed by a varint length, followed by data.

Unlike strings, binary data has no encoding requirement (no UTF-8 constraint).

6.7. Maps / Objects (Wire Type 6)

Key-value maps (equivalent to JSON objects). The inline value (or varint) indicates the number of key-value pairs.

Each entry consists of:

1. A PSON string (the key). Map keys MUST be strings.
2. A PSON value (any wire type, including nested maps and arrays).

* *0-30 entries:* Single tag byte with inline count.
* *>= 31 entries:* Tag byte 0xDF, followed by a varint count.

Key ordering: PSON does not define a canonical key ordering. However, implementations SHOULD preserve insertion order to support use cases where iteration order is significant (e.g., compact streaming mode in IOTMP).

Duplicate keys: A PSON map SHOULD NOT contain duplicate keys. A decoder that encounters duplicate keys SHOULD reject the map or use the last value for the duplicated key. The behavior for duplicate keys is undefined and implementations MUST NOT rely on it.

Example -- {"temp": 25, "hum": 60}:

```
C2                  # map_t, 2 entries
  84 74 65 6D 70    # string "temp" (key, 4 bytes)
  19                # unsigned 25 (value)
  83 68 75 6D       # string "hum" (key, 3 bytes)
  1F 3C             # unsigned 60 (varint)
```

Total: 13 bytes. Equivalent JSON {"temp":25,"hum":60} = 20 bytes (35% savings).

6.8. Arrays (Wire Type 7)

Ordered sequences of values (equivalent to JSON arrays). The inline value (or varint) indicates the number of elements.

Each element is a PSON-encoded value (any wire type).

* *0-30 elements:* Single tag byte with inline count.
* *>= 31 elements:* Tag byte 0xFF, followed by a varint count.

Example -- [1, 2, 3]:

```
E3                  # array_t, 3 elements
  01                # unsigned 1
  02                # unsigned 2
  03                # unsigned 3
```

Total: 4 bytes. Equivalent JSON [1,2,3] = 7 bytes (43% savings).

7. Byte Ordering

All multi-byte floating-point values (float32, float64) MUST be encoded in little-endian byte order (least significant byte first).

Rationale: Traditional Internet protocols use big-endian ("network byte order"), a convention established when dominant networking hardware was big-endian. Today, virtually all microcontrollers and processors used in IoT are little-endian: ARM Cortex-M, ESP32 (Xtensa), RISC-V, and x86. Little-endian encoding allows constrained devices to write float and double values directly from memory to the wire with no conversion. On the most constrained ARM cores (Cortex-M0/M0+), which lack a hardware byte-reversal instruction (REV), byte swapping a 32-bit float requires 4 instructions per value.

Integer values use varint encoding Section 8, which is byte-order independent by design.

This is the same design choice made by Protocol Buffers [ProtocolBuffers], which encodes fixed-width floats and doubles in little-endian regardless of platform.

8. Varint Encoding

PSON uses Protocol Buffers-style variable-length integer encoding for values that exceed the inline capacity (>= 31).

8.1. Encoding Algorithm

* Each byte uses bits [6:0] for data (7 bits per byte).
* Bit [7] is a continuation flag: set (1) if more bytes follow, clear (0) for the last byte.
* Least significant group comes first (little-endian byte order).

8.2. Examples

Decimal	Hex	Varint Bytes (hex)	Size
0	0x00	00	1 byte
1	0x01	01	1 byte
127	0x7F	7F	1 byte
128	0x80	80 01	2 bytes
300	0x012C	AC 02	2 bytes
16384	0x4000	80 80 01	3 bytes

Table 8

8.3. Maximum Varint Size

Implementations MUST support varints up to 10 bytes, representing values up to 2^64 - 1 (the full uint64 range). A 64-bit unsigned integer requires at most 10 varint bytes (ceil(64/7) = 10). A receiver that encounters a varint that does not terminate within 10 bytes MUST treat it as a decode error and discard the data.

9. Size Limits

* *Strings and binary data:* The maximum byte length is constrained by the varint encoding (up to 2^64 - 1). In practice, implementations SHOULD impose limits appropriate to available memory. A RECOMMENDED limit for constrained devices is 65,535 bytes (16-bit addressable).

* *Maps and arrays:* Element counts have no protocol-enforced limit beyond the varint range, but implementations SHOULD impose reasonable limits based on available memory.

* *Nesting depth:* Implementations SHOULD limit nesting depth to prevent stack overflow. A limit of 16 levels is RECOMMENDED for constrained devices.

10. Encoding Optimizations

The following optimizations are RECOMMENDED for encoders but are not required for protocol conformance. A conformant decoder MUST be able to decode data produced by any conformant encoder, whether or not these optimizations are applied.

10.1. Float-to-Integer Promotion

When encoding a floating-point number that has no fractional part and whose absolute value fits within the uint64 range (0 to 2^64-1), implementations SHOULD encode it as an unsigned integer (wire type 0) or signed integer (wire type 1) instead of a float (wire type 2). This saves 3-7 bytes per value and exploits the fact that many IoT sensor readings are whole numbers (e.g., humidity percentages, relay states, counters).

This promotion MUST NOT be applied to negative zero (-0.0), NaN, or Infinity values.

A decoder that receives an integer where a float was expected MUST convert it to the appropriate floating-point type. This promotion is transparent to the application.

Size savings:

Value	As float (type 2)	As integer (type 0/1)	Savings
0.0	5 bytes	1 byte	4 bytes
25.0	5 bytes	1 byte	4 bytes
-3.0	5 bytes	1 byte	4 bytes
100.0	5 bytes	3 bytes	2 bytes

Table 9

Note: CBOR’s core specification ([RFC8949], Section 4.2.2) treats float-to-integer promotion as optional and application-dependent. PSON follows the same approach: this optimization is RECOMMENDED but not required for conformance.

10.2. Precision Selection

When encoding a floating-point value that requires fractional precision, implementations SHOULD choose 32-bit float (inline value 0) when the float32 representation of the value is identical to the original float64 value. Otherwise, 64-bit double (inline value 1) MUST be used. This saves 4 bytes per value.

More precisely: an encoder SHOULD encode a float64 value v as float32 if and only if (float64)(float32)v == v (the round-trip through float32 preserves the exact value).

11. PSON vs JSON Size Comparison

The following table compares encoded sizes for common IoT data patterns:

Data	JSON (bytes)	PSON (bytes)	Savings
25	2	1	50%
true	4	1	75%
null	4	1	75%
"hello"	7	6	14%
{"temp":25,"hum":60}	20	13	35%
{"temp":25.3,"hum":60.1,"co2":412}	34	27	21%
[1,2,3,4,5]	11	6	45%

Table 10

Note: PSON sizes assume float32 precision for floating-point values, which is typical for IoT sensor data. JSON sizes use compact encoding with no whitespace.

For typical IoT payloads (maps with numeric sensor values), PSON achieves 21-75% size reduction compared to JSON.

12. Complete Encoding Examples

12.1. Simple Sensor Reading

JSON: {"temperature": 23.5, "humidity": 60}

```
C2                        # map_t, 2 entries
  8B 74 65 6D 70 65 72 61 # string "temperature"
     74 75 72 65           #   (key, 11 bytes)
  40 00 00 BC 41           # float 23.5 (32-bit, LE)
  88 68 75 6D 69 64 69     # string "humidity"
     74 79                 #   (key, 8 bytes)
  1F 3C                    # unsigned 60 (varint)
```

JSON size: 34 bytes. PSON size: 29 bytes. Savings: 15%.

12.2. Device Credentials

JSON: ["user", "device1", "secretkey"]

```
E3                        # array_t, 3 elements
  84 75 73 65 72          # string "user" (4 bytes)
  87 64 65 76 69 63 65 31 # string "device1" (7 bytes)
  89 73 65 63 72 65 74    # string "secretkey"
     6B 65 79             #   (9 bytes)
```

JSON size: 30 bytes. PSON size: 24 bytes. Savings: 20%.

12.3. Boolean Configuration

JSON: {"enabled": true, "debug": false}

```
C2                        # map_t, 2 entries
  87 65 6E 61 62 6C 65 64 # string "enabled" (7 bytes)
  61                      # true
  85 64 65 62 75 67       # string "debug" (5 bytes)
  60                      # false
```

JSON size: 30 bytes. PSON size: 17 bytes. Savings: 43%.

12.4. Nested Structure

JSON: {"gps": {"lat": 40.4168, "lon": -3.7038}, "alt": 650}

```
C2                        # map_t, 2 entries
  83 67 70 73             # string "gps" (3 bytes)
  C2                      # map_t, 2 entries (nested)
    83 6C 61 74           # string "lat" (3 bytes)
    41 85 7C D0 B3        # double 40.4168
       59 35 44 40        #   (64-bit, LE)
    83 6C 6F 6E           # string "lon" (3 bytes)
    41 FE 65 F7 E4        # double -3.7038
       61 A1 0D C0        #   (64-bit, LE)
  83 61 6C 74             # string "alt" (3 bytes)
  1F 8A 05                # unsigned 650 (varint)
```

JSON size: 47 bytes. PSON size: 39 bytes. Savings: 17%.

(Note: GPS coordinates require double precision, limiting compaction. Integer values like altitude benefit most from PSON encoding.)

13. Security Considerations

13.1. Input Validation

Implementations MUST validate PSON data during decoding:

* *Length bounds:* String and binary lengths MUST NOT exceed the implementation's maximum Section 9. A decoder MUST NOT allocate memory based on an untrusted length field without bounds checking.

* *Nesting depth:* Deeply nested maps and arrays can cause stack overflow on constrained devices. Implementations SHOULD enforce a maximum nesting depth.

* *Map key uniqueness:* Implementations SHOULD reject maps with duplicate keys, as they may indicate an injection attempt.

* *Varint termination:* A varint that does not terminate (every byte has the continuation bit set) constitutes a denial-of-service vector. Implementations MUST enforce the maximum varint length Section 8.3.

* *Wire type validation:* A decoder MUST reject values with reserved inline values (e.g., floating_t with inline value > 1, discrete_t with inline value > 2).

13.2. Denial of Service

PSON data from untrusted sources can be crafted to consume excessive resources:

* *Large allocations:* A malicious length prefix can trigger large memory allocations. Implementations MUST impose application-appropriate limits and MUST NOT allocate memory based solely on untrusted length fields.

* *Processing time:* Deeply nested structures can cause excessive processing time. Depth limits Section 9 mitigate this.

13.3. Confidentiality

PSON does not provide encryption. When transmitting sensitive data, PSON MUST be used within an encrypted transport (e.g., TLS).

14. IANA Considerations

14.1. Media Type Registration

This specification requests registration of the following media type in the "Media Types" registry:

* Type name: application
* Subtype name: pson
* Required parameters: none
* Optional parameters: none
* Encoding considerations: binary
* Security considerations: See Section 13 of this document
* Interoperability considerations: PSON uses little-endian byte ordering for floating-point values, which differs from CBOR and most Internet protocols. Implementations on big-endian platforms MUST perform byte swapping when encoding and decoding floats and doubles.
* Published specification: This document
* Applications that use this media type: IoT device communication, sensor data encoding, IOTMP protocol payloads
* Fragment identifier considerations: N/A
* Contact: Alvaro Luis Bustamante, alvaro@thinger.io
* Change controller: IETF

15. References

15.1. Normative References

[IEEE754] IEEE, "IEEE Standard for Floating-Point Arithmetic", IEEE 754-2019, 2019.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/info/rfc2119>.

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <https://www.rfc-editor.org/info/rfc8174>.

15.2. Informative References

[MessagePack] Furuhashi, S., "MessagePack specification", <https://msgpack.org/>.

[ProtocolBuffers] Google, "Protocol Buffers Encoding", <https://protobuf.dev/programming-guides/encoding/>.

[RFC8259] Bray, T., Ed., "The JavaScript Object Notation (JSON) Data Interchange Format", STD 90, RFC 8259, DOI 10.17487/RFC8259, December 2017, <https://www.rfc-editor.org/info/rfc8259>.

[RFC8949] Bormann, C. and P. Hoffman, "Concise Binary Object Representation (CBOR)", STD 94, RFC 8949, DOI 10.17487/RFC8949, December 2020, <https://www.rfc-editor.org/info/rfc8949>.

Appendix A. Conformance Test Vectors

The following test vectors allow implementations to verify correct encoding and decoding. Each entry shows a logical value and its expected PSON encoding in hexadecimal.

A.1. Integer Test Vectors

Value	Encoded (hex)	Size
Unsigned 0	00	1 byte
Unsigned 25	19	1 byte
Unsigned 30	1E	1 byte
Unsigned 31	1F 1F	2 bytes

Unsigned 300	1F AC 02	3 bytes
Signed -1	21	1 byte
Signed -30	3E	1 byte
Signed -300	3F AC 02	3 bytes

Table 11

A.2. Floating-Point Test Vectors

Value	Encoded (hex)	Size
Float 23.5	40 00 00 BC 41	5 bytes
Float 3.14	40 C3 F5 48 40	5 bytes
Double 3.141592653	41 38 E9 2F 54 FB 21 09 40	9 bytes

Table 12

A.3. Discrete Test Vectors

Value	Encoded (hex)	Size
False	60	1 byte
True	61	1 byte
Null	62	1 byte

Table 13

A.4. String Test Vectors

Value	Encoded (hex)	Size
"" (empty)	80	1 byte
"hi"	82 68 69	3 bytes

| "temperature" | 8B 74 65 6D 70 65 72 61 74 75 72 65 | 12 bytes |

Table 14

A.5. Container Test Vectors

Value	Encoded (hex)	Size
Empty map {}	C0	1 byte
Empty array []	E0	1 byte
Array [1,2,3]	E3 01 02 03	4 bytes

Table 15

A.6. Composite Test Vector

Map {"temp": 25, "hum": 60} -- 13 bytes total:

```
C2                  # map_t, 2 entries
  84 74 65 6D 70    # string "temp" (key, 4 bytes)
  19                # unsigned 25 (value)
  83 68 75 6D       # string "hum" (key, 3 bytes)
  1F 3C             # unsigned 60 (varint)
```

Hex: C2 84 74 65 6D 70 19 83 68 75 6D 1F 3C

Appendix B. Implementation Complexity Comparison

This appendix provides a quantitative comparison of implementation complexity between PSON and two widely-used CBOR libraries for constrained devices. All measurements use the same test payload: {"temperature": 23.5, "humidity": 60, "pressure": 1013, "label": "outdoor"}.

B.1. Source Code Size

Implementation	Source Lines (encoder + decoder)
PSON (standalone C)	344
NanoCBOR (C, minimal CBOR)	2,223

| TinyCBOR (C, Intel) | 5,619 |

Table 16

PSON's reduced source size results from its narrower scope: 8 wire types with a single varint overflow mechanism, no semantic tags, no indefinite-length encoding, no half-precision floats, and string-only map keys.

B.2. Compiled Binary Size (ESP32, ESP-IDF v5.4)

All projects compiled for ESP32 using espressif/idf:v5.4 Docker image with default optimization (-Og). No networking, no TLS -- pure encoding benchmark.

Impl.	Library	App Code	Total	Full Image
PSON	0 B (inline)	2,589 B	2,589 B	195,044 B
NanoCBOR	2,306 B	1,510 B	3,816 B	196,076 B
TinyCBOR	4,445 B	2,159 B	6,604 B	200,888 B

Table 17

PSON's compiled encoding code is 32% smaller than NanoCBOR and 61% smaller than TinyCBOR for identical functionality. Full image sizes differ by less than 3% because the ESP-IDF base (FreeRTOS, HAL, libc) dominates at ~190 KB.

NanoCBOR is the most minimal CBOR implementation available for constrained devices. The difference with TinyCBOR illustrates how CBOR's broader feature set (validation, pretty-printing, JSON conversion) increases footprint even when those features are not used by the application.

B.3. Encoded Wire Size

All three encodings produce comparable wire sizes for the same payload:

Implementation	Encoded Size	Notes
PSON	55 bytes	float32 for 23.5
NanoCBOR	53 bytes	float16 for 23.5 (half-precision)
TinyCBOR	55 bytes	float32 for 23.5

Table 18

The 2-byte difference with NanoCBOR is due to CBOR's half-precision float support (IEEE 754 binary16), which PSON deliberately omits to avoid the implementation complexity of float16 conversion routines. For typical IoT payloads, wire size differences between PSON and CBOR are negligible.

B.4. Encoding and Decoding Performance (ESP32, 240 MHz)

All benchmarks executed on the same ESP32 hardware (Xtensa LX6, 240 MHz, single core used). Each measurement averages 1,000,000 iterations of encoding or decoding the test payload.

Implementation	Encode (us/iter)	Decode (us/iter)
PSON	10.00	5.93
NanoCBOR	19.65	16.86
TinyCBOR	20.04	52.78

Table 19

Encoding: PSON encodes approximately 2x faster than both CBOR implementations. PSON's single-pass encoder with varint overflow requires fewer branches than CBOR's fixed-width (1/2/4/8 byte) argument encoding.

Decoding: PSON decodes 2.8x faster than NanoCBOR and 8.9x faster than TinyCBOR. The decoder benefits from the simpler tag structure (no indefinite-length containers to check, no semantic tags to skip, no half-precision float conversion, string-only map keys). TinyCBOR's significantly slower decoding reflects its more complex parser, which must handle the full CBOR data model including container tracking and validation.

These performance differences are a direct consequence of PSON's reduced format complexity, not implementation quality -- NanoCBOR is a well-optimized, production-quality CBOR library specifically designed for constrained devices.

B.5. Summary

Metric	PSON vs NanoCBOR	PSON vs TinyCBOR
Wire size	Comparable (+2 bytes)	Equal
Compiled code	32% smaller	61% smaller
Encode speed	2.0x faster	2.0x faster
Decode speed	2.8x faster	8.9x faster
Source lines	6.5x fewer	16.3x fewer

Table 20

Note: The primary justification for PSON is not performance superiority over CBOR, but the architectural integration with the IOTMP protocol -- shared encoding primitives eliminate the need for two independent type systems on constrained devices (see Section 1.3). The performance and complexity advantages documented here are a consequence of that narrower design scope.

Appendix C. Revision History

Version	Date	Changes
0.1	2026-03-30	Initial public draft.

Table 21

Author’s Address

Alvaro Luis Bustamante
Internet of Thinger SL
Spain
Email: alvaro@thinger.io