CBOR
Internet-Draft
Intended status: Standards Track
Expires: 23 May 2026

L. Lundblade
Security Theory LLC
19 November 2025

CBOR Serialization and Determinism
draft-ietf-cbor-serialization-00

Abstract

This document defines two CBOR serializations: "ordinary serialization" and "deterministic serialization." It also introduces the term "general serialization" to name the full, variable set of serialization options defined in [STD94]. Together, these three form a complete set of serializations that cover the majority of CBOR serialization use cases.

These serializations are largely compatible with those widely implemented by the CBOR community.

About This Document

This note is to be removed before publishing as an RFC.

Status information for this document may be found at https://datatracker.ietf.org/doc/draft-ietf-cbor-serialization/.

Discussion of this document takes place on the CBOR Working Group mailing list (mailto:cbor@ietf.org), which is archived at https://mailarchive.ietf.org/arch/browse/cbor/. Subscribe at https://www.ietf.org/mailman/listinfo/cbor/.

Source for this draft and an issue tracker can be found at https://github.com/cbor-wg/draft-ietf-cbor-serialization.

Status of This Memo

Copyright Notice

Table of Contents

1. Introduction 3
2. General Serialization 3
3. Ordinary Serialization 4
  3.1. Encoder Requirements 4
  3.2. Decoder Requirements 6
  3.3. When to use ordinary serialization 6
  3.4. Relation To Preferred Serialization 7
4. Deterministic Serialization 8
  4.1. Encoder Requirements 8
  4.2. Decoder Requirements 9
  4.3. When to use Deterministic Serialization 9
5. CDDL Support 9
6. Security Considerations 10
7. IANA Considerations 10
8. References 10
  8.1. Normative References 10
  8.2. Informative References 10
Appendix A. Information Model, Data Model and Serialization . . 11
Appendix B. General Protocol Considerations for Determinism . . 12
Appendix C. Deterministic Encoding for Popular Tags 13
  C.1. Date Strings, Tag 0 13
  C.2. Epoch Date, Tag 1 13
    C.2.1. Encoder Requirements 14
    C.2.2. Decoder Requirements 14
  C.3. Big Numbers, Tags 2 and 3 14

C.4. Big Floats and Decimal Fractions, Tags 4 and 5 14
C.4.1. Encoder Requirements 14
C.4.2. Decoder Requirements 14
Appendix D. IEEE 754 NaN 14
D.1. Basics 14
D.2. Implementation Support for Non-Trivial NaNs 15
D.3. Protocol Use and Non-use for Non-Trivial NaNs 16
D.4. Incompatibility with STD94 16
D.5. Recommendations for Use of Non-Trival NaNs in CBOR 17
Appendix E. Examples and Test Vectors 17
Contributors 18
Author’s Address 18

1. Introduction

Background material on serialization and determinism concepts is provided in Appendix A. Readers may wish to review this background information first.

This document defines new serializations rather than attempting to clarify those in [STD94] (that need clarification). This approach enables the serialization requirements to be expressed directly in normative [RFC2119] language, and to be consolidated in this single comprehensive specification. This approach provides clarity and simplicity for implementers and the CBOR community over the long term.

The serializations defined herein are formally new, but largely interchangeable with the way the serializations desecribed in [STD94] are implemented.

For example, preferred serialization described in [STD94] is commonly implemented without support for indefinite-lengths. Ordinary serialization is defined here is largely the same preferred serialization without indefinite-lengths, so it is largely interchangeable with what is commonly implemented.

2. General Serialization

This section assigns the name "general serialization" to the full set of serialization options standardized in Section 3 of [STD94]. This full set was not explicitly named in [STD94].

General serialization consists of all of these:

* Any length CBOR argument (e.g., the integer 0 may be encoded as 0x00, 0x1800 or or 0x190000 and so on).

* Any length floating point regardless of value (e.g. 0.00 can be 0xf900, 0xfa000000000 and so on).

* Both definite or indefinite-length strings, arrays and maps are allowed.

* Big numbers can represent values that are also representable by major types 0 and 1 (e.g., 0 can be encoded as a big number, as 0xc34100).

A decoder that supports general serialization is able to decode all of these.

If a CBOR-based protocol specification does not explicitly specify serialization, general serialization is implied. This means that a compliant decoder for such a protocol is required to accept all forms allowed by general serialization including both definite and indefinite lengths. For example, CBOR Web Token, [RFC8392] does not specify serialization; therefore, a full and proper CWT decoder must be able to handle variable-length CBOR argments plus indefinite-length strings, arrays and maps.

In practice, however, it is widely recognized that some CWT decoders cannot process the full range of general serialization, particularly indefinite lengths. As a result, CWT encoders typically limit themselves to the subset of serializations that decoders can reliably handle, most notably by never encoding indefinite lengths. It is similar for other CBOR-based protocols like [RFC9052]. See also Section 3.

Note also that there is no shortest-length requirement for floating-point encoding in general serialization. Thus, IEEE 754 NaNs (See Appendix D) may be encoded with a desired size, regardless of their payload — a principle sometimes stated as “touch not the NaNs.”

Finally, note also that general serialization is inherently non-deterministic because some CBOR data items can be serialized in multiple ways.

3. Ordinary Serialization

This section defines a serialization named "ordinary serialization."

3.1. Encoder Requirements

1. The shortest-form of the CBOR argument must be used for all major types. The shortest-form encoding for any argument that is not a floating point value is:

   * 0 to 23 and -1 to -24 MUST be encoded in the same byte as the major type.

   * 24 to 255 and -25 to -256 MUST be encoded only with an additional byte (ai = 0x18).

   * 256 to 65535 and -257 to -65536 MUST be encoded only with an additional two bytes (ai = 0x19).

   * 65536 to 4294967295 and -65537 to -4294967296 MUST be encoded only with an additional four bytes (ai = 0x1a).

2. If maps or arrays are encoded, they MUST use definite-length encoding (never indefinite-length).

3. If text or byte strings are encoded, they MUST use definite-length encoding (never indefinite-length).

4. If floating-point numbers are encoded, the following apply:

   * Half-precision MUST be supported

   * Values MUST be encoded in the shortest of double, single or half-precision that preserves precision. For example, 0.0 can always be reduced to half-precision so it MUST be encoded as 0xf90000 For another example, 0.1 would loose precision if not encoded as double-precision so it MUST be encoded as 0xfb3fb999999999999a. Subnormal numbers MUST be supported in this shortest-length encoding.

   * The only NaN that may be encoded is a half-precision quiet NaN (the sign bit and all but the highest payload bit is clear), specifically 0xf97e00.

   * Aside from the the requirement allowing only the half-precision quiet NaN, these are the same floating-point requirements as Section 4.1 of [STD94] and also as Section 4.2.1 of [STD94].

5. If big numbers (tags 2 and 3) are encoded, the following apply:

   * Leadings zeros MUST NOT be encoded.

   * If a value can be encoded using major type 0 or 1, then it MUST be encoded with major type 0 or 1, never as a big number.

3.2. Decoder Requirements

1. Decoders MUST accept shortest-form encoded arguments.

2. If arrays or maps are supported, definite-length arrays or maps MUST be accepted.

3. If text or byte strings are supported, definite-length text or byte strings MUST be accepted.

4. If floating-point numbers are supported, the following apply:

    * Half-precision values MUST be accepted.

    * Double- and single-precision values SHOULD be accepted; leaving these out is only foreseen for decoders that need to work in exceptionally constrained environments.

    * If double-precision values are accepted, single-precision values MUST be accepted.

5. If big numbers (tags 2 and 3) are accepted, the following apply:

    * Big numbers described in Section 3.4.3 of [STD94] MUST be accepted.

    * Leading zeros SHOULD be ignored.

    * An empty string SHOULD be accepted and treated as the value zero.

3.3. When to use ordinary serialization

The purpose of ordinary serialization is to provide interoperability without requiring support for indefinite-length decoding. If an encoder never produces indefinite-length items, the decoder can safely treat them as errors. Supporting indefinite-length decoding, especially for strings, introduces additional complexity and often necessitates dynamic memory allocation, so omitting it significantly reduces the implementation burden.

Ordinary serialization also provides a size efficiency gain by encoding the CBOR argument in the shortest form. Implementations typically find encoding and decoding in this form to be straightforward.

The easy implementation and broad usefulness makes ordinary serialization the best choice for most CBOR protocols. To some degree it is a de facto standard for common CBOR protocols.

However, it is not suitable if determinism is needed because the order of items in a map is allowed to vary. See Section 4.3.

It may also not be suitable in some cases where special functionality is needed like the following:

* Streaming of of strings, arrays and maps in constrained environments where the length is not known

* Non-trival NaNs need to be supported

* Hardware environments where integers are encoded/decoded directly from/to hardware registers and shortest-length CBOR arguments would be burdensome

In those cases, a special/custom serialization can be defined.

But, for the vast majority of use cases, ordinary serialization provides interoperaibility, small encoded size and low implementation costs.

3.4. Relation To Preferred Serialization

Ordinary serialization is defined to be the long-term replacement for preferred serialization.

The differences are:

* Definite lengths are a requirement, not a preference.

* The only NaN allowed is the half-precision quiet NaN.

These differences are not of significance in real-world implementations, so ordinary serialization is already largely supported.

In Section 3 of [STD94] it states that in preferred serialization the use of definite-length encoding is a "preference", not a requirement. Technically that means preferred seriaization decoders must support indefinite legnths, but in reality many do not. Indefinite lengths, particularly for strings, are often not supported because they are more complex to implement than other parts of CBOR. Because of this, the implementation of most CBOR protocols use only definite lengths.

Further, much of the CBOR community didn’t notice the use of the word "preference" and realize its implications for decoder implementations. It was somewhat assumed that preferred serialization didn’t allow indefinite lengths. That preferred serialization decoders are technically required to support indefinite lengths wasn’t noticed until many years after the publication of [STD94].

Briefly stated, the reason that the divergence on NaNs is not of consequence in the real world, is that their non-trivial forms are used extremely rarely and support for them in programming environments and CBOR libraries is unreliable. See Appendix D.4 for a detailed discussion.

Thus ordinary serialization is largely interchangable with preferred serialization in the real world.

4. Deterministic Serialization

This section defines a serialization named "deterministic serialization"

Deterministic serialization is the same as described in Section 4.2.1 of [STD94] except for the encoding of floating-point NaNs. See Section 3 and Appendix D for details on and rationale for NaN encoding.

Note that in deterministic serialization, any big number that can be represented as an integer must be encoded as an integer. This rule is inherited from ordinary serialization (Section 3), just as Section 4.2.1 of [STD94] inherits this requirement from preferred serialization.

4.1. Encoder Requirements

1. All of ordinary serialization defined in Section 3.1 MUST be used.

2. If a map is encoded, the items in it MUST be sorted in the bytewise lexicographic order of their deterministic encodings of the map keys. (Note that this is the same as the sorting in Section 4.2.1 of [STD94] and not the same as Section 3.9 of [RFC7049].

4.2. Decoder Requirements

1. Decoders MUST meet the decoder requirements for Section 3.2. That is, deterministic encoding imposes no requirements over and above the requirements for decoding ordinary serialization.

4.3. When to use Deterministic Serialization

Most applications do not require deterministic encoding — even those that employ signing or hashing to authenticate or protect the integrity of data. For example, the payload of a COSE_Sign message (See [RFC9052]) does not need to be encoded deterministically because it is transmitted along with the message. The recipient receives the exact same bytes that were signed.

Deterministic encoding becomes necessary only when the protected data is not transmitted as the exact bytes that are used for authenticity or integrity verification. In such cases, both the sender and the receiver must independently construct the exact same sequence of bytes. To guarantee this, the encoding must eliminate all variability and ambiguity. The Sig_structure, defined in Section 4.4 of [RFC9052], is an example of this requirement. Such designs are often chosen to reduce data size, preserve privacy, or meet other design constraints.

The only difference between ordinary and deterministic serialization is map key sorting. Sorting can be expensive in very constrained environments. This is the only reason these two are not combined into one.

Deterministically encoded data is always decodable, even by receivers that do not specifically support deterministic encoding. Deterministic encoding can be helpful for debugging and such. In environments where map sorting is not costly, it is acceptable and beneficial to always use it. In such an environment, a CBOR encoder may produce deterministic encoding by default and may even omit support for ordinary encoding entirely. But note that determinstic is never a substitue for general serialization where uses cases may require indefinite lengths, separate big numbers from integers in the data model, need non-trivial NaNs or other.

5. CDDL Support

TODO -- complete work and remove this comment

6. Security Considerations

The security considerations in Section 10 of [STD94] apply.

7. IANA Considerations

TODO -- complete work and remove this comment before publication

8. References

8.1. Normative References

[IEEE754] IEEE, "IEEE Standard for Floating-Point Arithmetic", IEEE Std 754-2019, DOI 10.1109/IEEESTD.2019.8766229, <https://ieeexplore.ieee.org/document/8766229>.

[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/rfc/rfc2119>.

[RFC8610] Birkholz, H., Vigano, C., and C. Bormann, "Concise Data Definition Language (CDDL): A Notational Convention to Express Concise Binary Object Representation (CBOR) and JSON Data Structures", RFC 8610, DOI 10.17487/RFC8610, June 2019, <https://www.rfc-editor.org/rfc/rfc8610>.

[STD94] 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/rfc/rfc8949>.

8.2. Informative References

[NaNBoxing] Nystrom, R., "Crafting Interpreters", July 2021, <https://craftinginterpreters.com/optimization.html#nan-boxing>.

[RFC7049] Bormann, C. and P. Hoffman, "Concise Binary Object Representation (CBOR)", RFC 7049, DOI 10.17487/RFC7049, October 2013, <https://www.rfc-editor.org/rfc/rfc7049>.

[RFC8392] Jones, M., Wahlstroem, E., Erdtman, S., and H. Tschofenig, "CBOR Web Token (CWT)", RFC 8392, DOI 10.17487/RFC8392, May 2018, <https://www.rfc-editor.org/rfc/rfc8392>.

[RFC9052] Schaad, J., "CBOR Object Signing and Encryption (COSE): Structures and Process", STD 96, RFC 9052, DOI 10.17487/RFC9052, August 2022, <https://www.rfc-editor.org/rfc/rfc9052>.

Appendix A. Information Model, Data Model and Serialization

To understand CBOR serialization and determinism, it's helpful to distinguish between the general concepts of an information model, a data model, and serialization. These are broad concepts that can be applied to other serialization schemes like JSON and ASN.1

	Information Model	Data Model	Serialization
Abstraction Level	Top level; conceptual	Realization of information in data structures and data types	Actual bytes encoded for transmission
Example	The temperature of something	A floating-point number representing the temperature	Encoded CBOR of a floating-point number
Standards		CDDL	CBOR
Implementation Representation		API Input to CBOR encoder library, output from CBOR decoder library	Encoded CBOR in memory or for transmission

Table 1

CBOR doesn't provide facilities for information models. They are mentioned here for completeness and to provide some context.

CBOR defines a palette of basic types that are the usual integers, floating-point numbers, strings, arrays, maps and other. Extended types may be constructed from these basic types. These basic and extended types are used to construct the data model of a CBOR protocol. While not required, [RFC8610] may be used to describe the data model of a protocol. The types in the data model are serialized per [STD94] to create encoded CBOR.

CBOR allows certain data types to be serialized in multiple ways to facilitate easier implementation in constrained environments. For example, indefinite-length encoding enables strings, arrays, and maps to be streamed without knowing their length upfront.

Crucially, CBOR allows — and even expects — that some implementations will not support all serialization variants. In contrast, JSON permits variations (e.g., representing 1 as 1, 1.0, or 0.1e1), but expects all parsers to handle them. That is, the variation in JSON is for human readability, not to facilitate easier implementation in constrained environments.

Appendix B. General Protocol Considerations for Determinism

This is the section that covers what is know as ALDR in some discussions.

// RFC Editor: Please remove above sentence before publication

In addition to Section 4 and Appendix C, there are considerations in the design of any deterministic protocol.

For a protocol to be deterministic, both the encoding (serialization) and data model (application) layer must be deterministic. While deterministic serialization, Section 4, ensures determinism at the encoding layer, requirements at the application layer may also be necessary.

Here’s an example application layer specification:

> At the sender’s convenience, the birth date MAY be sent either as an integer epoch date or string date. The receiver MUST decode both formats.

While this specification is interoperable, it lacks determinism. There is variability in the data model layer akin to variability in the CBOR encoding layer when deterministic serialization is not required.

To make this example application layer specification deterministic, specify one date format and prohibit the other.

A more interesting source of application layer variability comes from CBOR’s variety of number types. For instance, the number 2 can be represented as an integer, float, big number, decimal fraction and other. Most protocols designs will just specify one number type to use, and that will give determinism, but here’s an example specification that doesn’t:

> At the sender’s convenience, the fluid level measurement MAY be encoded as an integer or a floating-point number. This allows for minimal encoding size while supporting a large range. The receiver MUST be able to accept both integers and floating-point numbers for the measurement.

Again, this ensures interoperability but not determinism — identical fluid level measurements can be represented in more than one way. Determinism can be achieved by allowing only floating-point, though that doesn’t minimize encoding size.

A better solution requires the fluid level always be encoded using the smallest representation for every particular value. For example, a fluid level of 2 is always encoding as an integer, never as a floating-point number. 2.000001 is always be encoded as a floating-point number so as to not lose precision. See the numeric reduction defined by dCBOR.

Although this is not strictly a CBOR issue, deterministic CBOR protocol designers should be mindful of variability in Unicode text, as some characters can be encoded in multiple ways.

While this is not an exhaustive list of application-layer considerations for deterministic CBOR protocols, it highlights the nature of variability in the data model layer and some sources of variability in the CBOR data model (i.e., in the application layer).

Appendix C. Deterministic Encoding for Popular Tags

The definitions of the following tags in [RFC8610] allow variation in the data mode, thus it is useful to define a deterministic encoding for them should a particular deterministic protocol need one. The tags defined in [RFC8610] but not mentioned here have no variability in their data model.

C.1. Date Strings, Tag 0

TODO -- complete this work and remove this comment before publication

C.2. Epoch Date, Tag 1

C.2.1. Encoder Requirements

The integer form MUST be used unless one of the following applies: (1) the date is too far in the past or future to fit in a 64-bit integer of type 0 or 1, or (2) the date requires sub-second precision. In these cases, the floating-point form MUST be used instead.

C.2.2. Decoder Requirements

The decoder MUST decode both the integer and floating-point form.

C.3. Big Numbers, Tags 2 and 3

See Section 3.

C.4. Big Floats and Decimal Fractions, Tags 4 and 5

C.4.1. Encoder Requirements

The mantissa MUST be encoded in the preferred serialization form specified in Section 3.4.3 of RFC 8949.

The mantissa MUST NOT contain trailing zeros. For example, the decimal fraction with value 10 must be encoded with a mantissa of 1 and an exponent of 1. For big floats, the mantissa must not include any trailing zero bits if encoded as a type 0 or 1 integer, and no trailing zero bytes if encoded as a big number

C.4.2. Decoder Requirements

Both the integer and big number forms of the mantissa MUST be decoded.

Appendix D. IEEE 754 NaN

This section provides background information on [IEEE754] NaN (Not a Number) and its use in CBOR.

D.1. Basics

[IEEE754] defines the most widely used representation for floating-point numbers, including special values for infinity and NaN. NaN was originally designed to represent the result of invalid computations, such as division by zero. Although IEEE 754 intended NaN primarily for local computation, NaN values are sometimes transmitted in network protocols, and CBOR supports their representation.

An IEEE 754 NaN includes a payload of up to 52 bits (depending on precision), whose use is not formally defined. The original intent was for vendor-specific diagnostic information explaining why a computation failed. NaN values also include an unused sign bit.

IEEE 754 distinguishes between quiet NaNs (qNaNs) and signaling NaNs (sNaNs):

* A signaling NaN typically raises a floating-point exception when encountered.

* A quiet NaN does not raise an exception.

* The distinction is implementation-specific, but typically:

  - The highest bit of the payload is set --> quiet NaN.

  - Any other payload bit is set --> signaling NaN.

* At least one payload bit must be set for a signaling NaN to distinguish it from infinity.

In this document:

* A non-trivial NaN refers to any NaN that is not a quiet NaN.

* Non-trivial NaNs are often used to embed additional protocol information in the NaN payload.

D.2. Implementation Support for Non-Trivial NaNs

This section discusses the extent of programming language and CPU support for NaN payloads.

Although [IEEE754] has existed for decades, support for manipulating non-trivial NaNs has historically been limited and inconsistent. Some key points:

* Programming languages:

  - The programming languages C, C++, Java, Pyhton and Rust do no provide APIs to set or extract NaN payloads.

  - IEEE 754 is over thirty years old, enough time for support to be added if there was need.

* CPU hardware:

- CPUs use the distinction between signaling and quiet NaNs to determine whether to raise exceptions.

- A non-trivial NaN matching the CPU’s signaling NaN pattern may either trigger an exception or be converted into a quiet NaN.

- Instructions converting between single and double precision often discard or alter NaN payloads.

As a result, applications that rely on non-trivial NaNs generally cannot depend on CPU instructions, floating-point libraries, or programming environments. Instead, they usually need their own software implementation of IEEE 754 to encode and decode the full bit patterns to reliably process non-trivial NaNs.

D.3. Protocol Use and Non-use for Non-Trivial NaNs

One motivation for transmitting NaNs in CBOR is the technique known as NaN boxing (See [NaNBoxing]), used in some language runtimes (e.g., JavaScript engines) to represent multiple data types efficiently within a single 64-bit word. Another motivation arises when applications that internally rely on NaNs are split across a protocol boundary. For example, the R programming language uses non-trivial NaNs internally.

By contrast, JSON can encode IEEE 754 floating-point numbers but explicitly disallows NaN in all forms. As a result, CBOR protocols that allow NaN cannot be directly mapped to JSON.

Protocols often require an out-of-band indicator to signal the absence of a value. JSON uses null for this purpose, and CBOR protocols can also use null instead of NaN.

D.4. Incompatibility with [STD94]

Although [STD94] is not entirely explicit about non-trivial NaNs, it is generally interpreted as supporting non-trivial NaNs in the CBOR generic data model. It is also interpreted as requiring that non-trivial NaNs be reduce to their shortest form for preferred serialization — the opposite of "touch not the NaNs"

This document diverges from that interpretation:

* Ordinary serialization: Non-trivial NaNs are not allowed. While ordinary serialization typically aligns with preferred serialization, it does not in the case of non-trivial NaNs.

* Deterministic serialization: Because deterministic serialization inherits from ordinary serialization, it also does not allow non-trivial NaNs. This diverges from Section 4.2.1 of [STD94] in this one specific way.

The divergence is justified by the following:

* Non-trivial NaNs were not clearly specified in [STD94].

* They are not well-supported across CPUs and programming environments.

* Implementing preferred serialization for non-trivial NaNs is complex and error-prone; many CBOR implementations don't support it or don't support it correctly.

* Practical use cases for non-trivial NaNs are extremely rare.

* Reducing non-trivial NaNs to a half-precision quiet NaN is simple and widely supported (e.g., isnan() can be used to detect all NaNs).

* Non-trivial NaNs remain supported by general serialization; the divergence is only for ordinary and deterministic serialization.

* A new CBOR tag could be defined in the future to explicitly support them if needed.

D.5. Recommendations for Use of Non-Trival NaNs in CBOR

In summary, non-trival NaNs can be used in CBOR, but should primarily be used in systems that already use them and with full awareness that support in programming environments and CBOR libraries will be limited and inconsistent.

For new protocols, non-trival NaNs, even all NaNs, can be avoided by using other CBOR protocol elements like null. CBOR is powerful and flexible so as to allow data structures that can express an error detail or out-of-band value without using non-trival NaNs. The advantage of avoiding NaN in CBOR protocols is that they can more easily be JSON protocols and one does not need to worry about programming environment and CPU hardware support.

Appendix E. Examples and Test Vectors

TODO -- complete work and remove this comment before publication

Contributors

Rohan Mahy
Rohan Mahy Consulting Services
Email: rohan.ietf@gmail.com

Joe Hildebrand
Email: hildjj@cursive.net

Wolf McNally
Blockchain Commons
Email: wolf@wolfmcnally.com

Carsten Borman
Universitt Bremen TZI
Email: cabo@tzi.org

Author’s Address

Laurence Lundblade
Security Theory LLC
Email: lgl@securitytheory.com