

CBOR  
Internet-Draft  
Intended status: Standards Track  
Expires: 25 October 2026

L. Lundblade  
Security Theory LLC  
23 April 2026

CBOR Serialization and Determinism  
draft-ietf-cbor-serialization-06

## Abstract

This document defines two CBOR serializations: "preferred-plus serialization" and "deterministic serialization." It also introduces the term "general serialization" to name the full, variable set of serialization options defined in RFC 8949. 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

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 25 October 2026.

## Copyright Notice

Copyright (c) 2026 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<https://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Revised BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Revised BSD License.

## Table of Contents

|  |    |
|--|----|
| 1. Introduction . . . . .                                    | 3  |
| 1.1. Interoperability . . . . .                              | 4  |
| 1.2. Determinism . . . . .                                   | 5  |
| 1.3. Relation to RFC 8949 . . . . .                          | 5  |
| 2. Recommendations Summary . . . . .                         | 6  |
| 2.1. Protocol Specifications . . . . .                       | 6  |
| 2.1.1. Framework Protocols . . . . .                         | 6  |
| 2.1.2. End-to-End Protocols . . . . .                        | 7  |
| 2.2. Libraries . . . . .                                     | 7  |
| 2.2.1. CBOR Libraries . . . . .                              | 7  |
| 2.2.2. Libraries for Framework Protocols . . . . .           | 8  |
| 2.2.3. Libraries for End-to-End Protocols . . . . .          | 8  |
| 3. General Serialization . . . . .                           | 9  |
| 3.1. When To Use General Serialization . . . . .             | 9  |
| 3.2. General Serialization is the Default . . . . .          | 10 |
| 4. Preferred-Plus Serialization . . . . .                    | 10 |
| 4.1. Encoder Requirements . . . . .                          | 10 |
| 4.2. Decoder Requirements . . . . .                          | 11 |
| 4.3. When to use preferred-plus serialization . . . . .      | 12 |
| 4.4. Relation To Preferred Serialization . . . . .           | 13 |
| 5. Deterministic Serialization . . . . .                     | 13 |
| 5.1. Encoder Requirements . . . . .                          | 14 |
| 5.2. Decoder Requirements . . . . .                          | 14 |
| 5.3. When to use Deterministic Serialization . . . . .       | 14 |
| 5.3.1. Not Commonly Needed for Hashing and Signing . . . . . | 14 |

|                  |   |    |
|------------------|---|----|
| 5.3.2.           | Decoding Deterministic Serialization and Relation to Preferred-Plus Serialization . . . . . | 15 |
| 5.3.3.           | No Map Ordering Semantics . . . . .   | 15 |
| 6.               | Special Serializations . . . . .  | 16 |
| 7.               | New Tag Data Model Rule . . . . .   | 16 |
| 8.               | CDDL Control Operators . . . . .  | 17 |
| 9.               | Security Considerations . . . . .   | 17 |
| 10.              | IANA Considerations . . . . .   | 17 |
| 11.              | References . . . . .  | 18 |
| 11.1.            | Normative References . . . . .  | 18 |
| 11.2.            | Informative References . . . . .  | 19 |
| Appendix A.      | Information Model, Data Model and Serialization . . . . .                                   | 20 |
| Appendix B.      | General Protocol Considerations for Determinism . . . . .                                   | 21 |
| Appendix C.      | IEEE 754 NaN . . . . .  | 22 |
| C.1.             | Basics . . . . .  | 22 |
| C.2.             | Implementation Support for Non-Trivial NaNs . . . . .                                       | 23 |
| C.3.             | Use and Non-use for Non-Trivial NaNs . . . . .  | 24 |
| C.4.             | Clarification of RFC 8949 . . . . .   | 25 |
| C.5.             | Divergence from STD94 . . . . .   | 26 |
| C.6.             | Recommendations for Use of Non-Trivial NaNs . . . . .                                       | 27 |
| Appendix D.      | Big Numbers and the CBOR Data Model . . . . .   | 27 |
| Appendix E.      | Big Number Implementation Strategies . . . . .  | 28 |
| Appendix F.      | Serialization Checking . . . . .  | 29 |
| F.1.             | Serialization Checking Use Cases . . . . .  | 29 |
| Appendix G.      | CBOR Byte String Wrapping . . . . .   | 30 |
| G.1.             | Purpose . . . . .   | 30 |
| G.2.             | Wrapping Recommendations . . . . .  | 31 |
| G.3.             | CBOR Library Implementation Suggestion . . . . .  | 31 |
| Appendix H.      | Serialization for COSE . . . . .  | 31 |
| H.1.             | COSE Payload Serialization . . . . .  | 32 |
| H.2.             | COSE Sig_structure . . . . .  | 32 |
| H.3.             | The Encoded Message . . . . .   | 33 |
| Appendix I.      | Examples . . . . .  | 34 |
| I.1.             | Use for Testing . . . . .   | 34 |
| I.1.1.           | Encode Test . . . . .   | 35 |
| I.1.2.           | Decode Test . . . . .   | 35 |
| I.1.3.           | Checking Decoder Test . . . . .   | 35 |
| I.1.4.           | Non-Checking Decoder Test . . . . .   | 36 |
| I.2.             | Example Data Items . . . . .  | 36 |
| Contributors     | . . . . .   | 45 |
| Author's Address | . . . . .   | 46 |

## 1. Introduction

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

CBOR intentionally allows multiple valid serializations of the same data item. For example, the array [1, 2] can be serialized in more than one way:

| Type              | Description                                      | Bytes               |
|-------------------|--|---------------------|
| Definite-length   | The array length (2) is encoded at the beginning | 0x82 0x01 0x02      |
| Indefinite-length | The array is terminated by the break byte (0xff) | 0x9f 0x01 0x02 0xff |

Table 1

Similar variation exists for integers, maps, strings, and floating-point numbers.

This variability is deliberate. CBOR is designed to allow encodings to be selected according to the constraints and requirements of a particular environment. The flexibility is a core design feature. (CBOR is not unique in this regard; BER and DER encoding for ASN.1 are a similar design choice.)

For example, indefinite-length serialization is suited for streaming large arrays in constrained environments, where the total length is not known in advance. Conversely, definite-length serialization works well to decode small arrays in constrained environments.

As a result, CBOR libraries and protocol implementations commonly support only the serialization forms required for their intended use cases. This behavior is expected and aligns with CBOR's design goals.

However, this flexibility introduces two challenges: interoperability and determinism.

### 1.1. Interoperability

The interoperability challenge arises because partial implementations are both permitted and expected. For example, an encoder might transmit an indefinite-length array to a decoder that does not support indefinite-length encodings. Both implementations are compliant with [STD94].

Decoders, in particular, frequently choose not to support all serialization forms. This may be due to operation in constrained environments or because implementing a full general decoder is significantly more work (particularly in languages like C and Rust, which lack built-in support for dynamic arrays, maps, and strings).

In practice, most CBOR usage occurs outside highly constrained environments. This makes it both feasible and beneficial to define a common serialization suitable for general use.

Protocol specifications can reference such a serialization rather than restating detailed encoding rules, and library implementations can prioritize support for it.

This document defines that serialization: preferred-plus serialization.

## 1.2. Determinism

The determinism challenge arises because there are multiple ways to serialize the same data item. The example serialization of the array [1,2] above shows this. This is a problem in some protocols that hash or sign encoded CBOR.

Many approaches to deterministic serialization are possible, each optimized for different environmental constraints or application requirements. However, as noted earlier, the majority of CBOR usage occurs outside constrained environments. It is therefore practical to define a single deterministic serialization suitable for general use.

Protocol specifications and library implementations can reference this serialization instead of defining their own deterministic encoding rules.

This document defines that serialization: deterministic serialization.

## 1.3. Relation to RFC 8949

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 described in [STD94] are implemented.

For example, preferred serialization described in [STD94] is commonly implemented without support for indefinite-lengths.

Preferred-plus serialization as defined here is largely the same as preferred serialization without indefinite-lengths, so it is largely interchangeable with what is commonly implemented.

## 2. Recommendations Summary

### 2.1. Protocol Specifications

#### 2.1.1. Framework Protocols

Framework protocols are those that offer a set of options and alternatives, with interoperability depending on the sender and receiver making compatible choices. These protocols sometimes make use of profiles to define interoperability requirements for specific uses. Framework protocols are sometimes described as toolbox or building-block protocols, reflecting their role as collections of reusable mechanisms rather than end-to-end protocols. CWT, COSE, EAT, and CBOR itself are examples of framework protocols.

It is RECOMMENDED that CBOR-based framework protocols not state serialization requirements, enabling individual uses and profiles to choose serialization to suit their environments and constraints.

CBOR-based framework protocols MAY impose serialization requirements. For example, if a protocol is never expected to be deployed in constrained environments where map sorting is too expensive, it may mandate deterministic serialization for all implementations in order to eliminate all serialization variability.

There is one situation in which a framework protocol MUST require deterministic serialization, though typically limited to a specific subset of the protocol. This requirement arises when the protocol design requires the involved parties to independently construct and serialize data to be hashed or signed, rather than transmitting the exact serialized bytes that were hashed or signed. See Section 5.3.

See Appendix H for a COSE-based example.

### 2.1.2. End-to-End Protocols

End-to-end protocols are specified such that interoperability is assured when they are implemented in accordance with their specification. When such a protocol includes optional features, they are typically selected through real-time negotiation. Such protocols often have formal interoperability compliance programs or organize interoperability testing events (for example, "bake-offs"). TLS, HTTP, and FIDO are examples of end-to-end protocols.

End-to-end protocols **MUST** define a serialization strategy that ensures the sender and receiver use interoperable serialization.

The strategy most highly **RECOMMENDED** is to normatively require preferred-plus serialization. If a protocol does not need to be deployed where map sorting is too expensive, requiring deterministic serialization is also **RECOMMENDED**.

An end-to-end protocol **MAY** instead define its own specialized serialization (see Section 6). In such cases, it **MUST** explicitly specify the permitted serialization behaviors necessary to ensure interoperability. For example, if a sender is permitted to use indefinite-length serialization, the protocol **MUST** require that receivers be capable of decoding indefinite-length items.

As with framework protocols, deterministic serialization may be required for parts of the protocol using hashing or signing. See Section 5.3.

If no specific serialization is required, general serialization (see Section 3) applies by default. In this case, the sender **MAY** use any valid serialization, and the receiver **MUST** be able to decode it. Defaulting to general serialization is **NOT RECOMMENDED**, because some serializations like indefinite-lengths are not widely supported.

## 2.2. Libraries

### 2.2.1. CBOR Libraries

It is **RECOMMENDED** that CBOR libraries support preferred-plus serialization. This can be achieved by conforming to the decoding requirements in Section 4.2 and by making the encoding behavior defined in Section 4.1 the default or primary encoding API.

Preferred-plus serialization is recommended because it is suitable for the majority of CBOR-based protocols. In practice, preferred-plus serialization is equivalent to preferred serialization Section 4.1 of [STD94] for most use cases.

It is also RECOMMENDED that CBOR libraries support deterministic serialization, as some protocols (for example, COSE) require it. Relative to preferred-plus serialization, the only additional requirement for deterministic serialization is that encoded maps be sorted. This recommendation is particularly strong for environments in which map sorting is easy to implement (for example, Python, Go, and Ruby).

A CBOR library may choose to implement only deterministic serialization and make it the default. Deterministic serialization is a superset of preferred-plus serialization; therefore, if deterministic serialization is fully supported, explicit support for preferred-plus serialization may be omitted.

A CBOR library MAY also choose to support some or all aspects of general serialization (see Section 3) thereby enabling support for protocols that use specialized serializations (see Section 6).

#### 2.2.2. Libraries for Framework Protocols

When a framework protocol specification does not mandate a specific serialization, it is RECOMMENDED that it implement preferred-plus serialization. For example, it is recommended that a library implementing CWT or COSE implement preferred-plus serialization.

However, a library MAY choose to support only deterministic serialization if this aligns with its deployment environment and design goals.

When a framework protocol mandates serialization requirements, libraries must of course conform. For instance, certain parts of COSE mandate deterministic serialization. See Appendix H for a COSE-based example.

#### 2.2.3. Libraries for End-to-End Protocols

End-to-end protocols should have explicit serialization requirements to ensure interoperability. Libraries for end-to-end protocols should fulfill them.

If an end-to-end protocol specification does not state serialization requirements, the library is free to choose, but it is RECOMMENDED that they implement preferred-plus serialization.



### 3. 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).
- \* Floating-point values may be encoded using any length (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.

#### 3.1. When To Use General Serialization

Preferred-plus serialization (Section 4) satisfies the vast majority of CBOR use cases; therefore, the need for general serialization is rare and arises only in unusual circumstances. The following are representative examples:

- \* Enable on-the-fly, streaming encoding of strings, arrays, and maps with indefinite lengths. This is useful when an array, map, or string is many times larger than the available memory on the encoding device.
- \* Directly encode or decode integer values from hardware registers with fixed-size integer encoding. CBOR is sufficiently simple that encoders and decoders for some protocols can be implemented solely in hardware without any software. Fixed-size integer encoding allows values to be copied directly in and out of hardware registers.
- \* Enable in place update of the lengths of strings, arrays and maps by using fixed-size encoding of their lengths. For example, if the length of a string is always encoded in 32 bits, increasing the length from  $2^{16}$  to  $2^{16}+1$ , requires only overwriting the length field rather than shifting all  $2^{16}$  bytes of content.

- \* Transmission of non-trivial NaNs in floating-point values (see Appendix C).

Except for non-trivial NaNs, the other serializations can encode the same data types and value ranges as general serialization. Its purpose is solely to simplify or optimize encoding in atypical constrained environments. The choice of serialization is orthogonal to the data model. See also the section on special serializations in Section 6.

### 3.2. General Serialization is the Default

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 arguments 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. This is also true in practice of other protocols implementations like those for [RFC9052].

## 4. Preferred-Plus Serialization

This section defines a serialization named "preferred-plus serialization."

### 4.1. 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 lose precision if not encoded as double-precision so it MUST be encoded as 0xfb3fb99999999999a. 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 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:
    - \* Leading 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.

#### 4.2. 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 MUST be ignored.
- \* An empty byte string MUST be accepted and treated as the value zero.

See also Appendix D and Appendix E for further background on big numbers.

#### 4.3. When to use preferred-plus serialization

The purpose of preferred-plus 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.

Preferred-plus 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 preferred-plus serialization the best choice for most CBOR protocols. To some degree it is a de facto standard for common CBOR protocols.

See Section 3.1 for uses cases where preferred-plus serialization may not be suitable. Otherwise, for the vast majority of use cases, preferred-plus serialization provides interoperability, small encoded size and low implementation costs.

#### 4.4. Relation To Preferred Serialization

Preferred-plus 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.
- \* For big numbers, leading zeros must be ignored and the empty string must be accepted as zero.

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

Section 3 of [STD94] states that in preferred serialization the use of definite-length encoding is a "preference", not a requirement. Technically that means preferred serialization decoders must support indefinite lengths, 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 by many implementers until several 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 C.5 for a detailed discussion.

Thus preferred-plus serialization is largely interchangeable with preferred serialization in the real world.

#### 5. 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 4 and Appendix C for details on, and the 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 preferred-plus serialization (Section 4), just as Section 4.2.1 of [STD94] inherits this requirement from preferred serialization.

### 5.1. Encoder Requirements

1. All of preferred-plus serialization defined in Section 4.1 MUST be used.
2. If a map is encoded, the items in it MUST be sorted in the bitwise 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] / Section 4.2.3 of [STD94].)

### 5.2. Decoder Requirements

1. Decoders MUST meet the decoder requirements described in Section 4.2. That is, deterministic encoding imposes no requirements over and above the requirements for decoding preferred-plus serialization.

### 5.3. When to use Deterministic Serialization

#### 5.3.1. Not Commonly Needed for Hashing and Signing

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 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.

See the more detailed, COSE-based example in Appendix H.

#### 5.3.2. Decoding Deterministic Serialization and Relation to Preferred-Plus Serialization

The only difference between preferred-plus and deterministic serialization is that in deterministic serialization, maps are required to be sorted by their keys. Preferred-plus serialization exists as a separate mode solely because map sorting can be too expensive in some constrained environments.

Map decoding must never depend on the sort order of a map, even when maps are required to be sorted. As a result, deterministic serialization (Section 5) can always be decoded by a decoder that supports preferred-plus serialization (Section 4). Because of this property, deterministic serialization can always be used in place of preferred-plus serialization. In environments where map sorting is not costly, it is both acceptable and beneficial to always use deterministic serialization. In such environments, a CBOR encoder may produce deterministic encoding by default and may even omit support for preferred-plus encoding entirely.

However, note that deterministic serialization is never a substitute for general serialization where uses cases may require indefinite lengths, separate big numbers from integers in the data model, or need non-trivial NaNs.

#### 5.3.3. No Map Ordering Semantics

In the basic generic data model, maps are unordered (See Section 5.6 of [STD94]). Applications **MUST NOT** rely on any particular map ordering, even if deterministic serialization was used. A CBOR library is not required to preserve the order of keys when decoding a map, and the underlying programming language may not preserve map order either — for example, the Go programming language provides no ordering guarantees for maps. The sole purpose of map sorting in deterministic serialization is to ensure reproducibility of the encoded byte stream, not to provide any semantic ordering of map

entries. If an application requires a map to be ordered, it is responsible for applying its own sorting.

## 6. Special Serializations

Although discouraged, defining special serializations that differ from those specified here is permitted. For example, a use case might require determinism from a protocol that uses indefinite lengths. For another example, a protocol may require only a subset of general serialization features — for instance, fixed-length integer encodings but not indefinite lengths.

A recommended way to define a special serialization is to describe it as preferred-plus or deterministic serialization with additional constraints or extensions. For example, a protocol requiring deterministic streaming of maps and arrays can be defined as follows:

Deterministic serialization MUST be used, but all maps and arrays MUST be encoded with indefinite lengths, never definite lengths. Strings are still encoded with definite lengths. Maps are still to be ordered as required by deterministic serialization.

## 7. New Tag Data Model Rule

```
// This section is new in draft-03. The author thinks it may be out
// of place in this document, but there's no other good place for it
// yet.
```

Section 2 of [STD94] states that each new CBOR tag definition introduces a new and distinct data type. In contrast, the definitions of Tags 2 and 3 (bignums) in Section 3.4.3 of [STD94] do not introduce a separate data type; instead, they attach directly to the integer type and extend its numeric range. As a result, the generic data model's integer type is modified rather than augmented with a new, independent type (see Appendix D).

This document establishes a new rule that prohibits future tag definitions from having such effects:

All future CBOR tag definitions MUST NOT incorporate, modify, or otherwise affect any data types other than the type defined by the tag itself. A set of tags MAY affect each other, provided that all defining authorities for those tags explicitly agree.

Tags 2 and 3 are exempt from this rule, as they were defined prior to the establishment of this requirement.



## 8. CDDL Control Operators

Four new control operators are defined for use in CDDL [RFC8610].

| Name      | Purpose  |
|-----------|--|
| .prefp    | Use preferred-plus serialization for a data item     |
| .prefpseq | Use preferred-plus serialization for a CBOR sequence |
| .dtrm     | Use deterministic serialization for a data item      |
| .dtrmseq  | Use deterministic serialization for a CBOR sequence  |

Table 2

These operators have the same semantics as the .cbor and .cborseq operators (See Section 3.8.4 of [RFC8610]) with the additional requirement for preferred-plus or deterministic serialization. These specify that what is in the “controller” (the right side of the operator) be serialized as indicated.

For example, a byte string containing embedded CBOR that must be deterministically encoded can be described in CDDL as:

```
leaf = #6.24(bytes .dtrm any)
```

The scope of these operators applies recursively through nested arrays and maps, but does not extend into byte strings or other data items that happen to contain encoded CBOR. Every instance of embedded CBOR that requires constrained serialization must specify that constraint explicitly. See also Appendix G.

## 9. Security Considerations

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

## 10. IANA Considerations

```
// RFC Editor: please replace RFCXXXX with the RFC number of this RFC
// and remove this note.
```

This document requests IANA to register the contents of Table 3 into the registry "CDDL Control Operators" of the [IANA.cddl] registry group:

| Name      | Reference |
|-----------|-----------|
| .prefp    | [RFCXXXX] |
| .prefpseq | [RFCXXXX] |
| .dtrm     | [RFCXXXX] |
| .dtrmseq  | [RFCXXXX] |

Table 3: New control operators to be registered

IANA is requested to add a reference to Section 7 to the CBOR tag registry [IANA.cbor-tags].

## 11. References

### 11.1. Normative References

#### [IANA.cbor-tags]

IANA, "Concise Binary Object Representation (CBOR) Tags", <<https://www.iana.org/assignments/cbor-tags>>.

#### [IANA.cddl]

IANA, "Concise Data Definition Language (CDDL)", <<https://www.iana.org/assignments/cddl>>.

[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>>.

## 11.2. Informative References

- [CTAP2] W3C, "Client To Authenticator Protocol v2",  
<<https://fidoalliance.org/specs/fido-v2.0-ps-20190130/fido-client-to-authenticator-protocol-v2.0-ps-20190130.html>>.
- [Examples-Repo] IETF CBOR WG, "draft-ietf-cbor-serialization",  
<<https://github.com/cbor-wg/draft-ietf-cbor-serialization/tree/main/examples>>.
- [I-D.mcnally-deterministic-cbor] McNally, W., Allen, C., Bormann, C., and L. Lundblade,  
"dCBOR: Deterministic CBOR", Work in Progress, Internet-Draft, draft-mcnally-deterministic-cbor-17, 11 February 2026, <<https://datatracker.ietf.org/doc/html/draft-mcnally-deterministic-cbor-17>>.
- [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>>.
- [RFC9413] Thomson, M. and D. Schinazi, "Maintaining Robust Protocols", RFC 9413, DOI 10.17487/RFC9413, June 2023, <<https://www.rfc-editor.org/rfc/rfc9413>>.
- [UML] "OMG Unified Modeling Language (OMG UML) Version 2.5.1", December 2017, <<https://www.omg.org/spec/UML/2.5.1/PDF>>.

## 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                     | [UML]                        | CDDL  | CBOR                                       |
| Implementation Representation | n/a                          | API Input to CBOR encoder library, output from CBOR decoder library | Encoded CBOR in memory or for transmission |

Table 4

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.

// Please remove above sentence before publication

In addition to Section 5, 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 5, 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 [I-D.mcnally-deterministic-cbor].

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. IEEE 754 NaN

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

### C.1. Basics

[IEEE754] defines the most widely used representation for floating-point numbers. It includes 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. 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.
- \* A non-trivial NaN is used to carry additional, protocol-specific information within floating-point values.

## C.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, Python and Rust do not 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 sometimes 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.

### C.3. Use and Non-use for Non-Trivial NaNs

Non-trivial NaNs, excluding signaling NaNs, are not produced by standard floating-point operations. They are typically created at the application level, where software may take advantage of unused bits in the NaN payload. Such uses are rare and unusual, but they do exist.

One example is the R programming language, which is designed for statistical computing and therefore operates heavily on numeric data. R uses NaN payloads to distinguish various error or missing-data conditions beyond standard computational exceptions such as division by zero.

Another example is NaNboxing (see [NaNBoxing]), a technique used by some language runtimes — such as certain JavaScript engines — to efficiently represent multiple data types within a single 64-bit word by storing type tags or pointers in the NaN payload. (CBOR can represent such payloads, but NaNboxed pointers are generally not meaningful or portable across machines, and therefore are usually unsuitable for network transmission or file storage.)

CBOR's NaN-payload support can be leveraged if data from these systems must be transmitted over a network or written to persistent storage.

A designer of a new protocol that makes extensive use of floating-point values might be tempted to use NaN payloads to encode out-of-band information such as error conditions. For example, NaN payloads could be used to distinguish situations such as sensor offline, sensor absent, sensor error, or sensor out of calibration. While this is technically possible in CBOR, it comes with significant drawbacks:

- \* Preferred-plus and deterministic serialization cannot be used for this protocol.
- \* Support for NaN payloads is unreliable across programming environments and CBOR libraries.
- \* Values cannot be translated directly to JSON, which does not support NaNs of any kind.



#### C.4. Clarification of RFC 8949

This is a clarifying restatement of how NaNs are to be treated according to [STD94].

NaNs represented in floating-point values of different lengths are considered equivalent in the basic generic data model if:

- \* Their sign bits are identical, and
- \* Their significands are identical after both significands are zero-extended on the right to 64 bits

This equivalence is established for the entire CBOR basic generic data model. A NaN encoded as half-, single-, or double-precision is equivalent whenever it satisfies the rules above. This remains true regardless of how a CBOR library accepts, stores, or presents a NaN in its API. At the application layer, the equivalence still holds. The only way to avoid this equivalence is by using a tag specifically designed to carry NaNs without these equivalence rules, since tags extend the data model unless otherwise specified.

The equivalence is similar to how the floating-point value 1.0 is treated as the same value regardless of the precision used to encode it. Some floating-point values cannot be represented in shorter formats (e.g., 2.0e+50 cannot be encoded in half-precision). The same is true for some NaNs.

In preferred serialization, this equivalence **MUST** be used to shorten encoding length. If a NaN can be represented equivalently in a shorter form (e.g., half-precision rather than single-precision), then the shorter representation **MUS** be used.

This equivalence also applies when floating-point values are used as map keys. A map key encoded as half-precision **MUST** be considered a duplicate of one encoded as double-precision if they meet the equivalence rules above.

However, this equivalence does not apply to map sorting. Sorting operates on the fully encoded and serialized representation, not on the abstract data model.

It is Section 2 of [STD94] that establishes this equivalence by stating that the number of bytes used to encode a floating-point value is not visible in the data model. Section 4.1 of [STD94] defines preferred serialization. It requires shortest-length encoding of NaNs including instructions on how to do it. Section 5.6.1 of [STD94] describes how NaNs are treated as equivalent when used as map keys. These three parts of [STD94] are consistent and are the basis of this restatement.

Since Section 4.2.1 of [STD94], (Core Deterministic Encoding Requirements), explicitly requires preferred serialization, compliant deterministic encodings must use the shortest equivalent representation of NaNs.

Finally, Section 4.2.2 of [STD94] discusses alternative approaches to deterministic encoding. It suggests, for example, that all NaNs may be encoded as a half-precision quiet NaN. This section is distinct from the Core Deterministic Encoding Requirements and represents an optional alternative for handling NaNs.

#### C.5. Divergence from [STD94]

Non-trivial NaNs are not permitted in either preferred-plus or deterministic serializations. This is in contrast to preferred serialization and Section 4.2.1 of [STD94].

Note that the prohibition of non-trivial NaNs is the sole difference between deterministic serialization (Section 5) and Section 4.2.1 of [STD94].

The divergence is justified by the following:

- \* Encoding and equivalence of non-trivial NaNs was a little unclear [STD94].
- \* IEEE 754 doesn't set requirements for their handling.
- \* Non-trivial NaNs are not well-supported across CPUs and programming environments.
- \* Because preferred serialization of non-trivial NaNs is difficult and error-prone to implement, many CBOR implementations don't encode and/or decode non-trivial NaNs, or don't encode or decode them correctly.
- \* Practical use cases for non-trivial NaNs are extremely rare.

- \* Reducing non-trivial NaNs to a half-precision quiet NaN is simple and supported by programming environments (e.g., `isnan()` can be used to detect all NaNs).
- \* Non-trivial NaNs remain supported by general serialization; the divergence is only for preferred-plus and deterministic serialization.
- \* A new CBOR tag can be defined in the future to explicitly support them.

#### C.6. Recommendations for Use of Non-Trivial NaNs

While non-trivial NaNs are excluded from preferred-plus and deterministic serialization, they are theoretically supported by [STD94]. General serialization does support them.

New protocol designs SHOULD avoid non-trivial NaNs. Support for them is unreliable, and it is straightforward to design CBOR-based protocols that do not depend on them. In many cases, the use of NaN can be replaced entirely with null. JSON requires use of null as it does not support NaNs at all.

The primary use case for non-trivial NaNs is existing systems that already use them. For example, a program that relies on non-trivial NaNs internally may need to serialize its data to run across machines connected by a network.

#### Appendix D. Big Numbers and the CBOR Data Model

The primary purpose of this document is to define preferred-plus and deterministic serialization. Accordingly, Section 4 describes CBOR's unified integer space in terms of serialization behavior. This is an effective and clear way to describe what implementors must do. An implementation that follows the requirements in Section 4 will be complete and correct with respect to serialization.

From a conceptual perspective, however, additional discussion is warranted regarding the CBOR data model itself. That discussion is provided in this appendix. (Please review Appendix A for background on the difference between serialization and the data model).

In the basic, generic CBOR data model, each tag represents a distinct data type (Section 2 of [STD94]). Tags are also distinct from the major types, such as numbers and strings. By this, an integer value such as 0 or 1 encoded as major type 0 is clearly distinct in the data model from the same integer value encoded as tag 2.

However, the text in Section 3.4.3 of [STD94] overrides this by defining these encodings to be equivalent rather than distinct. This text therefore modifies the CBOR data model. No other serialization requirement in [STD94] or in this document alters the data model; this equivalence is the sole exception. This is unusual because the data model is otherwise orthogonal to serialization.

Further, Section 3.4.3 of [STD94] along with text in Section 2 of [STD94] are interpreted such that there is never a CBOR data model where there is a distinction between these integer representations. That is, the equivalence applies regardless of the serialization even though much of the relevant text appears in proximity to discussions of serialization.

This document does not attempt to update or revise the text of Section 3.4.3 of [STD94]. Rather, it records the commonly accepted interpretation of that text and its implications for the CBOR data model.

This document does create a new rule for future tag definitions. See Section 7.

## Appendix E. Big Number Implementation Strategies

Appendix D describes how CBOR defines a single integer number space, in which big numbers are not distinct from values encoded using major types 0 and 1. This appendix discusses approaches for implementers to support that model.

Some programming environments provide strong native support for big numbers (e.g., Python, Ruby, and Go), while others do not (e.g., C, C++, and Rust). Even in environments that support big numbers, operations on native-sized integers (e.g., 64-bit integers) are typically much more efficient. It is therefore reasonable for a CBOR library to expose separate APIs for native-sized integers and for big numbers.

When a CBOR library provides a big number API, values that fall within the range of major types 0 and 1 must be encoded using those major types rather than tags 2 or 3. Similarly, decoding facilities that return big numbers must accept values encoded using major types 0 and 1, even though the returned representation is a big number.

Alternatively, some CBOR libraries may choose to return tags 2 and 3 as raw byte strings, as this approach is simpler than implementing full big number support. When a library adopts this approach, it should clearly document that the application layer is responsible for performing the integer unification. The application is also

responsible for handling CBOR's offset-by-one encoding of negative values and the extended negative integer range permitted by major type 1.

In most cases, these additional processing steps are straightforward when the application already uses a big number library.

Another acceptable approach is for a CBOR library to provide a generic mechanism that allows applications to register handlers for specific tags. In this case, handlers for tags 2 and 3 MUST perform the required unification with major types 0 and 1.

Finally, note that big numbers are not a widely used feature of CBOR. Some CBOR libraries may entirely omit support for tags 2 and 3.

## Appendix F. Serialization Checking

Serialization checking rejects input which, while well-formed CBOR, does not conform to a particular serialization rule set it is enforcing. For example, a decoder checking for deterministic serialization will error out if map keys are not in the required sorted order. Likewise, a decoder checking for preferred-plus serialization will reject any CBOR data item that is not encoded in its shortest form.

This type of checking goes beyond the basic requirement of verifying that input is well-formed CBOR. The data rejected by serialization checking is well-formed; it is rejected because it violates additional serialization constraints.

### F.1. Serialization Checking Use Cases

Some applications that rely on deterministic serialization may choose serialization checking in order to ensure that the data they consume is truly deterministic and that the assumptions their logic makes about determinism hold.

Some protocol environments may use serialization checking to minimize representational variants as a strategy to improve interoperability. Discouraging variants early prevents them from compounding. See [RFC9413] on maintaining robust protocols.

Serialization checking may enhance security in certain contexts, but such checking is never a substitute for correct and complete CBOR input validation. All CBOR decoders — regardless of their capabilities, modes, or optional features — must always perform full input validation. This includes rejecting CBOR features the decoder does not support. For example, a decoder that does not support indefinite-length items must reject them because they are unsupported, not because it is acting as a checking decoder.

Decoders that fail to perform this essential input validation are fundamentally inadequate and represent a security risk. The appropriate remedy is to fix their input validation, not to add the serialization checking described here.

## Appendix G. CBOR Byte String Wrapping

This appendix provides non-normative guidance on byte-string wrapping of CBOR. It applies primarily to tag 24 and the CDDL `.cbor` and `.cborseq` control operators, but also to the serialization-specifying control operators described in Section 8. It also applies when prose states the byte-string wrapping requirement, such as for the COSE protected headers. See also Appendix H.1.

### G.1. Purpose

**Error isolation:** Wrapping CBOR in a byte string prevents encoding errors in the wrapped data from causing the enclosing CBOR to fail during decoding. (CBOR decoding generally halts at the first error and lacks internal length redundancy found in formats like ASN.1/DER.)

**CBOR library support for signing and hashing:** When wrapped CBOR needs to be signed or hashed, its original encoded bytes must be available. Most CBOR libraries cannot directly extract the raw bytes of substructures, but byte-string wrapping provides direct access to the exact bytes for signing or hashing.

**Protocol embedding:** Byte-string wrapping is generally useful when messages from one CBOR-based protocol need to be embedded within another CBOR protocol.

**Special map keys:** Some CBOR libraries only support simple, non-aggregate map keys (e.g., integers or strings). To use complex data types like arrays and maps as map keys, they can be wrapped in a byte string.

## G.2. Wrapping Recommendations

The serialization requirements for the wrapping CBOR may differ from those for the wrapped CBOR. CBOR itself imposes no universal rule that they must match; this is determined by the design of the wrapping protocol.

The wrapping protocol should not impose serialization requirements on the wrapped message. The two should be treated as independent entities. This approach avoids potential conflicts between serialization rules.

For example, assume protocol XYZ wraps protocol ABC. If protocol ABC requires Canonical CBOR as specified in Section 3.9 of [RFC7049] (e.g., [CTAP2] from WebAuthn) while protocol XYZ requires deterministic serialization, Section 5, a conflict would arise.

Most CBOR data to be signed or hashed does not require a specific serialization. CBOR, being a modern, fully specified, binary protocol, does not need canonicalization, wrapping, or armoring like other data representation formats such as JSON. See the discussion in Section 5.3.

## G.3. CBOR Library Implementation Suggestion

A straightforward implementation strategy is to instantiate a second CBOR encoder or decoder for the wrapped message. However, this may be suboptimal in memory-constrained environments, as it may require both a duplicate copy of the wrapped data and an additional encoder/decoder instance.

A more efficient approach can be for the CBOR library to treat the wrapped CBOR like a container (similar to arrays or maps). Many CBOR implementations already handle arrays and maps as containers without requiring a separate instance. Similarly, a byte-string wrapping encoded CBOR can be treated as a container that always contains exactly one item.

## Appendix H. Serialization for COSE

This appendix highlights how the topics in this document apply to CBOR Object Signing and Encryption (COSE [RFC9052]).

It focuses on the COSE\_Sign1 message (Section 4.2 of [RFC9052]), which is sufficient for illustrating the relevant considerations. COSE\_Sign1 is a simple structure for signing a payload. Its serialization can be described in three parts:

- \* The payload
- \* The Sig\_structure
- \* The encoded message (the header parameters and the array of four that is the COSE\_Sign1)

## H.1. COSE Payload Serialization

The signed payload may or may not be CBOR, but assume that it is, perhaps a CWT or EAT. The payload is transmitted from the signer/sender fully intact all the way to the verifier/receiver. Because it is transmitted fully intact, CBOR is a binary protocol and intermediaries do not do things like wrap long lines or add base 64 encoding or such, it is not special in any way and COSE imposes no serialization restrictions on it at all. That is, it can use any serialization it wants. The serialization is selected by the protocol that defines the payload, not by COSE.

This highlights the principle that determinism is often NOT needed for signing and hashing described in Section 5.3.

It is also worth noting that the payload is a byte string wrapped. This is not for determinism, armoring or canonicalization. It is so that the payload can be any data format, including not CBOR. It is also so CBOR libraries can return the CBOR-encoded payload for processing by the verification algorithms. Most CBOR libraries decoders do not provide access to any arbitrary chunk of encoded CBOR in the middle of a message. This is an example of byte string wrapping described in Appendix G.

## H.2. COSE Sig\_structure

The Sig\_structure Section 4.4 of [RFC9052] is used to aggregate all the items that are input to the signature algorithm — the payload, protected headers and other.

The Sig\_structure is not transmitted from the sender to the receiver; instead, it is constructed independently by both parties. COSE therefore explicitly requires deterministic encoding so that both the sender and receiver produce identical encoded CBOR representations. This requirement is specified in Section 9 of [RFC9052].

This COSE requirement is effectively equivalent to the deterministic serialization defined in Section 5, since no floating-point NaNs are involved. It is also effectively equivalent to preferred-plus serialization as defined in Section 4, because the Sig\_structure contains no maps.



The determinism requirement does not apply to the protected headers incorporated into the `Sig_structure`. Deterministic encoding of the headers is unnecessary because they are transmitted in the exact encoded form in which they are included in the `Sig_structure`.

Furthermore, determinism requirements do not extend into CBOR inside of byte strings. Once CBOR data is wrapped in a byte string, its internal encoding is treated as opaque and is not subject to surrounding serialization constraints.

This illustrates the general need for deterministic serialization when signed data is reconstructed rather than transmitted in the exact form that was signed. See Section 5.3.

### H.3. The Encoded Message

A `COSE_Sign1` message is an array of four elements containing two header parameter chunks, the payload, and the signature. The two header parameter chunks are maps that hold the various header parameters. COSE places no serialization requirements on these elements. The COSE protocol functions correctly regardless of the CBOR serialization used, as long as the decoder can decode what the encoder sends.

In this respect, the serialization of the `COSE_Sign1` message is no different from that of any other CBOR-based protocol message. Indefinite-length items may be used, and non-shortest CBOR arguments are permitted. The only requirement is that the serialization used by the encoder be decodable by the receiver.

Strictly speaking, COSE is a framework protocol intended for incorporation into an end-to-end protocol, which should explicitly define its serialization requirements. See Section 2.1.1 and Section 2.1.2.

In practice, some COSE libraries have implicitly implemented only the preferred (or preferred-plus) serialization, and end-to-end protocols have often defaulted to whatever behavior the underlying COSE library provides. While this generally works — particularly because the preferred serialization aligns with the recommendations here — it is more robust for an end-to-end protocol to state its serialization requirements explicitly.

## Appendix I. Examples

This appendix provides examples of the serializations described in this document. Each example is a single data item. Collectively, the examples cover the major CBOR data types and some special cases. Table 5 describes the five fields provided for each example.

| field                         | description  |
|-------------------------------|--|
| description                   | Text describing the item                                   |
| edn-representations           | Unencoded value(s) for the data item                       |
| general-serializations        | Encoded representation(s) for general serialization        |
| preferred-plus-serializations | Encoded representation(s) for preferred-plus serialization |
| deterministic-serializations  | Encoded representation for deterministic serialization     |

Table 5: Example Data Item Fields

### I.1. Use for Testing

These examples are designed to support testing of CBOR libraries. They cover only what is defined in this document and therefore do not provide complete or general CBOR test coverage. All examples are well-formed and valid.

While the CBOR-encoded serializations for each item can be used directly as test input, the EDN representation usually must be incorporated into the test manually. For example, the EDN string `"-5.0e-324"` will likely need to be passed as a value of type double to the API of a C-language CBOR library that encodes double-precision numbers.

Not all CBOR libraries support every data type represented in these examples. This is acceptable: tests for unsupported types may be skipped, or used to verify that an appropriate "unsupported" error is returned.

#### I.1.1.1. Encode Test

To test encoding, invoke the encoder for each example data item. The encoder input is the item's EDN representations. If an example provides multiple EDN representations, each of them should be tested.

The encoder should be either configured for, or to default to, one of the three serialization types described in this document. A test succeeds if the encoder produces any of the encoded representations given in the example for that serialization type.

If an encoder supports multiple serialization types, each type can be tested in turn.

#### I.1.1.2. Decode Test

To test decoding, invoke the decoder for each example data.

The decoder should be either to be configured for, or to default to, one of the three serialization types described in this document. For the selected serialization type, process every encoded representation defined for the target type. A test passes if the decoded output matches the value specified by the corresponding EDN representation.

If a decoder supports multiple serialization types, each type can be tested in turn.

#### I.1.1.3. Checking Decoder Test

Checking decoders are described in Appendix F.

This test verifies that a checking decoder rejects encodings allowed by general serialization but non-conforming for the target serialization type. It applies only to CBOR libraries that implement serialization conformance checking.

Testing a checking decoder for a target serialization type is typically performed as follows: for each example, supply the decoder with every representation permitted under general serialization except those allowed for the target serialization type. Decoding each such input must result in a conformance-checking error.

General-serialization decoders are not tested in this way, since they must accept all valid serialization forms.

#### I.1.4. Non-Checking Decoder Test

A non-checking decoder may accept encodings beyond what is required for the target serialization type. For example, a preferred-plus decoder will often accept non-shortest-length arguments, even though it is not required to do so, and such encodings are not permitted under preferred-plus. These examples can be used to test such extended decoding.

Testing proceeds similarly to that for a checking decoder: inputs outside the target serialization type are supplied to the decoder. The difference is that, for a non-checking decoder, many of these inputs may successfully decode rather than producing a conformance error. When decoding does fail, the expected error is typically “unsupported.”

Which encoding forms are accepted and which are rejected as unsupported is entirely dependent on the additional capabilities a CBOR library chooses to support and therefore not specified here.

Note the following:

- \* It is common for preferred-plus and deterministic decoders to accept non-shortest-length arguments.
- \* If the floating-point data type is supported, all serialization types described in this document require support for decoding half-precision representations and subnormals.

#### I.2. Example Data Items

These are available as individual files at [Examples-Repo].

All general serialization examples of strings, arrays and maps include indefinite-length encodings so as to provide full test cases. CBOR libraries that don't support indefinite-length decoding can not claim to support general serialization even if they support most of the rest of general serialization. For the purpose of classification by this document they are preferred-plus libraries with extra decoding features. The extra decoding features can be tested as described in Appendix I.1.4.

File: zero.edn

```
{
  "description": "The integer 0" ,
  "edn-representations" : ["0"] ,
  "general-serializations": [h'00',
                              h'1800',
                              h'190000',
                              h'1a00000000',
                              h'1b0000000000000000',
                              h'c2420000',
                              h'c240'],
  "preferred-plus-serializations": [h'00'],
  "deterministic-serialization": [h'00']
}
```

File: three.edn

```
{
  "description": "The integer 3" ,
  "edn-representations" : ["3"] ,
  "general-serializations": [h'03',
                              h'1803',
                              h'190003',
                              h'1a00000003',
                              h'1b0000000000000003',
                              h'c2420003' ],
  "preferred-plus-serializations": [h'03'],
  "deterministic-serialization": [h'03']
}
```

File: minus\_twenty\_five.edn

```
{
  "description": "The integer -25",
  "edn-representations" : ["-25"],
  "general-serializations": [h'3818',
                              h'390018',
                              h'3a00000018',
                              h'3b00000000000000018',
                              h'c3420018' ],
  "preferred-plus-serializations": [h'3818'],
  "deterministic-serialization": [h'3818']
}
```

File: 65\_bit\_neg.edn

```
{
  "description": "Largest negative integer" ,
  "edn-representations" : ["-18446744073709551616"] ,
  "general-serializations": [h'3bffffffffffffffffff',
                             h'c348ffffffffffffffffff'],
  "preferred-plus-serializations": [h'3bffffffffffffffffff'],
  "deterministic-serialization": [h'3bffffffffffffffffff']
}
```

File: byte\_string.edn

```
{
  "description": "The byte string of length 3: 0x010203",
  "edn-representations" : [h'010203'],
  "general-serializations": [h'43010203',
                             h'5f4101420203ff',
                             h'5f5801015a000000020203ff'],
  "preferred-plus-serializations": [h'43010203'],
  "deterministic-serialization": [h'43010203']
}
```

File: text\_string.edn

```
{
  "description": "The text string 'hi there'",
  "edn-representations" : ["\"hi there\""],
  "general-serializations": [
    h'686869207468657265',
    h'7f686869207468657265ff',
    h'7f64686920746468657265ff',
    h'7f790004686920747B00000000000000468657265ff'],
  "preferred-plus-serializations": [
    h'686869207468657265'],
  "deterministic-serialization": [
    h'686869207468657265']
}
```

File: array.edn

```
{
  "description": "The array [1, 2, 3]",
  "edn-representations" : ["[1, 2, 3]"],
  "general-serializations": [h'83010203',
                             h'9f010203ff'],
  "preferred-plus-serializations": [h'83010203'],
  "deterministic-serialization": [h'83010203']
}
```

File: map.edn

Note that map order is not significant in the CBOR data model. Maps are only sorted to provide deterministic encoding.

```
{
  "description": "Map with three items using integer map keys",
  "edn-representations" : [
    "{1:\\\"x\\\", 2:\\\"y\\\", 3:\\\"z\\\"}",
    "{1:\\\"x\\\", 3:\\\"z\\\", 2:\\\"y\\\"}",
    "{2:\\\"y\\\", 3:\\\"z\\\", 1:\\\"x\\\"}",
    "{2:\\\"y\\\", 1:\\\"x\\\", 3:\\\"z\\\"}",
    "{3:\\\"z\\\", 1:\\\"x\\\", 2:\\\"y\\\"}",
    "{3:\\\"z\\\", 2:\\\"y\\\", 1:\\\"x\\\"}" ],
  "general-serializations": [
    h'a301617802617903617a',
    h'a301617803617a026179',
    h'a302617903617a016178',
    h'a302617901617803617a',
    h'a303617a016178026179',
    h'a303617a026179016178',
    h'bf03617a026179016178ff',
    h'a31a00000003617a19000261791B000000000000000016178'],
  "preferred-plus-serializations": [
    h'a301617802617903617a',
    h'a301617803617a026179',
    h'a302617903617a016178',
    h'a302617901617803617a',
    h'a303617a016178026179',
    h'a303617a026179016178'],
  "deterministic-serialization": [
    h'a301617802617903617a']
}
```

File: map\_strings.edn

```
{
  "description": "Three item map with string keys",
  "edn-representations" : [
    {"abc": 1, "def": 2, "ghi": 3},
    {"abc": 1, "ghi": 3, "def": 2},
    {"def": 2, "abc": 1, "ghi": 3},
    {"def": 2, "ghi": 3, "abc": 1},
    {"ghi": 3, "abc": 1, "def": 2},
    {"ghi": 3, "def": 2, "abc": 1} ],
  "general-serializations": [
    h'a3636162630163646566026367686903',
    h'a3636162630163676869036364656602',
    h'a3636465660263616263016367686903',
    h'a3636465660263676869036361626301',
    h'a3636768690363616263016364656602',
    h'a3636768690363646566026361626301',
    h'bf6162630163646566026367686903ff',
    h'bf7f6161626263ff017f6264656166ff027f63676869ff03ff'],
  "preferred-plus-serializations": [
    h'a3636162630163646566026367686903',
    h'a3636162630163676869036364656602',
    h'a3636465660263616263016367686903',
    h'a3636465660263676869036361626301',
    h'a3636768690363616263016364656602',
    h'a3636768690363646566026361626301',
    h'bf6162630163646566026367686903ff' ],
  "deterministic-serialization": [
    h'a3636162630163646566026367686903' ]
}
```

File: positive\_bignum.edn

Note that big numbers are included in the test data because preferred-plus serialization requires their unification with integers.

```
{
  "description": "A positive big num",
  "edn-representations" : ["79228162514264337593543950335"],
  "general-serializations": [h'c24cffffffffffffffffffffffffffff',
                              h'c24e0000ffffffffffffffffffffffffffff' ],
  "preferred-plus-serializations": [h'c24cffffffffffffffffffffffffffff'],
  "deterministic-serialization": [h'c24cffffffffffffffffffffffffffff']
}
```

File: negative\_bignum.edn



Note that this is the value closest that can be represented as a big number, not a type 1 integer for preferred-plus serialization.

```
{
  "description": "Negative bignum on boundary with type 1 integer",
  "edn-representations" : ["-18446744073709551617"],
  "general-serializations": [
    h'c34901000000000000000000',
    h'c34c000000001000000000000000',
    h'c35f4500000000014800000000000000ff'],
  "preferred-plus-serializations": [
    h'c34901000000000000000000'],
  "deterministic-serialization": [
    h'c34901000000000000000000']
}
```

File: date\_epoch\_tag.edn

Note that this is provided as a test case for tags. There are no requirements for dates in this document. The tag content in this example does vary by serialization type.

```
{
  "description": "An epoch date tag" ,
  "edn-representations" : ["1(1776614355)"] ,
  "general-serializations": [h'c11a69e4fbd3',
    h'd8011a69e4fbd3',
    h'd900011a69e4fbd3',
    h'da000000011a69e4fbd3',
    h'db00000000000000011a69e4fbd3',
    h'c11b0000000069e4fbd3'],
  "preferred-plus-serializations": [h'c11a69e4fbd3'],
  "deterministic-serialization": [h'c11a69e4fbd3']
}
```

File: date\_string\_tag.edn

Note that this is provided as a test case for tags. There are no requirements for dates in this document. The tag content in this example does vary by serialization type.

```
{
  "description": "A date string tag" ,
  "edn-representations" : ["0(\"2026-04-19T03:59:15Z\")"] ,
  "general-serializations": [
    h'c074323032362d30342d31395430333a35393a31355a' ,
    h'd80074323032362d30342d31395430333a35393a31355a' ,
    h'd9000074323032362d30342d31395430333a35393a31355a' ,
    h'da0000000074323032362d30342d31395430333a35393a31355a' ,
    h'db000000000000000074323032362d30342d31395430333a35393a31355a' ,
    h'c07f74323032362d30342d31395430333a35393a31355aff' ,
    h'c07f6232307232362d30342d31395430333a35393a31355aff' ],
  "preferred-plus-serializations": [
    h'c074323032362d30342d31395430333a35393a31355a' ],
  "deterministic-serialization": [
    h'c074323032362d30342d31395430333a35393a31355a' ]
}
```

File: true.edn

```
{
  "description": "The simple value 'true'" ,
  "edn-representations" : ["true"] ,
  "general-serializations": [h'f5'],
  "preferred-plus-serializations": [h'f5'],
  "deterministic-serialization": [h'f5']
}
```

File: simple111.edn

Note that the simple value 111 is of not particular significance. It was selected because it is an unassigned simple value.

```
{
  "description": "The simple value 111" ,
  "edn-representations" : ["simple(111)"] ,
  "general-serializations": [h'f86f'],
  "preferred-plus-serializations": [h'f86f'],
  "deterministic-serialization": [h'f86f']
}
```

File: float\_zero.edn

```
{
  "description": "Floating-point positive zero",
  "edn-representations" : ["0.0"] ,
  "general-serializations": [h'f90000',
                             h'fa00000000',
                             h'fb0000000000000000'],
  "preferred-plus-serializations": [h'f90000'],
  "deterministic-serialization": [h'f90000']
}
```

File: float\_double.edn

```
{
  "description": "Double-precision normal float" ,
  "edn-representations" : ["1.7976931348623157e+308"] ,
  "general-serializations": [h'fb7fefffffffffffffff'],
  "preferred-plus-serializations": [h'fb7fefffffffffffffff'],
  "deterministic-serialization": [h'fb7fefffffffffffffff']
}
```

File: float\_double\_subnormal.edn

Note that full subnormal support is required for all serializations defined in this document.

```
{
  "description": "Double-precision negative subnormal float",
  "edn-representations": ["-5.0e-324"],
  "general-serializations": [h'fb8000000000000001' ],
  "preferred-plus-serializations": [h'fb8000000000000001' ],
  "deterministic-serialization": [h'fb8000000000000001']
}
```

File: float\_single.edn

```
{
  "description": "Single-precision normal float" ,
  "edn-representations" : ["-16777216.0"] ,
  "general-serializations": [h'fbc170000000000000',
                             h'facb800000'],
  "preferred-plus-serializations": [h'facb800000'],
  "deterministic-serialization": [h'facb800000']
}
```

File: float\_single\_subnormal.edn

```
{
  "description": "Single-precision subnormal float" ,
  "edn-representations" : ["5.8774717541114375E-39"] ,
  "general-serializations": [h'fb38000000000000',
                             h'fa00400000' ],
  "preferred-plus-serializations": [h'fa00400000'],
  "deterministic-serialization": [h'fa00400000']
}
```

File: float\_half.edn

```
{
  "description": "half-precision normal float" ,
  "edn-representations" : ["65504.0"] ,
  "general-serializations": [h'fb40effc0000000000',
                             h'fa477fe000',
                             h'f97bff' ],
  "preferred-plus-serializations": [h'f97bff'],
  "deterministic-serialization": [h'f97bff']
}
```

File: float\_half\_subnormal.edn

```
{
  "description": "Half-precision subnormal float" ,
  "edn-representations" : ["3.0517578125E-5"] ,
  "general-serializations": [h'fb3f00000000000000',
                             h'fa38000000',
                             h'f90200' ],
  "preferred-plus-serializations": [h'f90200'],
  "deterministic-serialization": [h'f90200']
}
```

File: float\_neg\_infinity.edn

```
{
  "description": "Negative Infinity",
  "edn-representations" : ["-Infinity"] ,
  "general-serializations": [h'f9fc00',
                             h'faff800000',
                             h'fbffff00000000000000'],
  "preferred-plus-serializations": [h'f9fc00'],
  "deterministic-serialization": [h'f9fc00']
}
```

File: float\_quiet\_nan.edn

```
{
  "description": "Floating-point quiet NaN",
  "edn-representations" : ["NaN"] ,
  "general-serializations": [h'f97e00',
                              h'fa7fc00000',
                              h'fb7ff8000000000000'],
  "preferred-plus-serializations": [h'f97e00'],
  "deterministic-serialization": [h'f97e00']
}
```

File: float\_nan\_payload.edn

The NaN payload is a special case. For preferred-plus and deterministic serialization, the decode should fail. For general serialization a NaN with a payload should be returned, but there is no EDN representation for that.

```
{
  "description": " Floating-point NaN payload/significand is 0x1ff",
  "edn-representations" : [] ,
  "general-serializations": [h'f97dff',
                              h'fa7fbfe000',
                              h'fb7ff7fc00000000000'],
  "preferred-plus-serializations": [],
  "deterministic-serialization": []
}
```

#### Contributors

Rohan Mahy  
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

Anders Rundgren

Email: anders.rundgren.net@gmail.com

Vadim Goncharov

Email: vadimnuclight@gmail.com

Ken Takayama

Email: ken.takayama.ietf@gmail.com

#### Author's Address

Laurence Lundblade

Security Theory LLC

Email: lgl@securitytheory.com