Internet Engineering Task Force
Internet-Draft
Intended status: Informational
Expires: 4 April 2026
A. Rundgren, Ed.
Independent
1 October 2025

CBOR::Core - CBOR Cross Platform Profile
draft-rundgren-cbor-core-13

Abstract

This document defines CBOR::Core, a platform profile for CBOR (RFC 8949) intended to serve as a viable replacement for JSON in computationally advanced systems like Internet browsers, mobile phones, and Web servers. To foster interoperability, deterministic encoding is mandated. Furthermore, the document outlines how deterministic encoding combined with enhanced CBOR tools, enable cryptographic methods like signing and hashing, to optionally use "raw" (non-wrapped) CBOR data as input. This document mainly targets CBOR tool developers.

Status of This Memo

Copyright Notice

Table of Contents

1. Introduction

The CBOR::Core specification is based on CBOR [RFC8949]. While there are different ways you can encode certain CBOR objects, this is non-trivial to support in general purpose platform-based tools, not to mention the limited utility of such measures. To cope with this, CBOR::Core defines a specific (non-variant) encoding scheme, aka "Deterministic Encoding". The selected encoding scheme is believed to be _compatible_ with most existing systems using CBOR. See also Appendix C.

CBOR::Core is intended to be agnostic with respect to programming languages and platforms.

By combining the compact binary representation and the rich set of data types offered by CBOR, with a deterministic encoding scheme, CBOR::Core could for _new designs_, serve as a viable alternative to JSON [RFC8259]. Although the mandated encoding scheme is deployable in [CONSTRAINED] environments, the primary target is rather general-purpose computing platforms like mobile phones and Web servers.

However, for unleashing the full power of deterministic encoding, the ability to perform cryptographic operations on "raw" (non-wrapped) CBOR data, compliant CBOR::Core tools need additional functionality. See also Appendix B.

1.1. Design Goals

The primary goal with this specification, is providing a foundation for CBOR tools that enable application developers to use CBOR without requiring insights in low-level details like encoding. In most cases, it should be sufficient to consult a list of supported data types. See also Section 2.3.2.

Section 2 contains the actual specification.

1.2. Requirements Language

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

1.3. Common Definitions

* This document uses the conventions defined in CDDL [RFC8610] for expressing the type of CBOR [RFC8949] data items.

* Examples showing CBOR data, are expressed in "diagnostic notation" (Section 2.3.4).

* The term "CBOR object" is equivalent to "CBOR data item" used in [RFC8949].

2. Detailed Description

This section describes the three pillars that CBOR::Core relies on.

2.1. Supported CBOR Objects

The following table shows the set of CBOR objects that compliant CBOR::Core implementations MUST support:

CBOR	Comment
int	Integer
bigint	Big integer
float	16-, 32-, and 64-bit [IEEE754] floating-point numbers
tstr	Text string encoded as UTF-8 [RFC3629]
bstr	Byte string
bool	Boolean true and false
null	Represents a null object
[]	Array
{}	Map
#6.nnn(type)	Tagged objects
#7.nnn	Simple values

Table 1: Supported CBOR Objects

Although extensions are imaginable, extensions will most likely cause _interoperability issues_ and are thus NOT RECOMMENDED.

However, nothing prevents developers from at the application (API) level, through CBOR tags and similar mapping concepts, support additional, "virtual" data types, analogous to how you map an application’s data model to the set of data types available, be it a data interchange format, a database, or a programming language.

OpenAPI [OPENAPI] is an example of an API that defines data types through mapping.

Application-specific implementations may (of course) only have to support the CBOR::Core objects required by the targeted application(s).

2.2. Deterministic Encoding Scheme

In CBOR::Core deterministic encoding is _mandatory_. The encoding scheme adheres to Section 4.2 of [RFC8949], but adds a few constraints (denoted by RFC+), where the RFC offers choices. The following list contains a summary of the CBOR::Core deterministic encoding rules:

* RFC+: Floating-point and integer objects MUST be treated as _distinct types_ regardless of their numeric value. This is compliant with "Rule 2" in Section 4.2.2 of [RFC8949].

* RFC: Integers, represented by the int and bigint types, MUST use the int type if the value is between -2^64 and 2^64-1, otherwise the bigint type MUST be used. Appendix A.1 features a list of integer sample values and their expected encoding.

* RFC+: The optimized representation of integers (aka "Preferred Serialization"), MUST also be applied to string lengths, array/map counts, and tag numbers.

* RFC: Floating-point numbers MUST always use the shortest [IEEE754] variant that preserves the precision of the original value. Appendix A.2 features a list of floating-point sample values and their expected encoding.

* RFC: Map keys MUST be sorted in the bytewise lexicographic order of their deterministic encoding. Duplicate keys MUST be _rejected_.

* RFC+: Since CBOR encodings according to this specification maintain uniqueness, there are no specific restrictions or tests needed in order to determine map key equivalence. As an (extreme) example, the floating-point numbers 0.0 and -0.0, and the integer number 0 could represent the distinct keys f90000, f98000, and 00 respectively.

* RFC: Indefinite length objects MUST be _rejected_.

2.3. Implementation Considerations

In CBOR::Core there are three distinguishable levels:

Encoding level:
"Wire format" as described in Section 2.2.

Encoder/decoder level:
This section.

Application level:
Constraints on data imposed by _applications_, like limiting ISO DateTime objects to UTC notation, or requiring integers representing enumerations to be in a specific range, is _out-of-scope_ for CBOR::Core.

2.3.1. API Requirements

An important feature that deterministic encoding brings to the table is that wrapping CBOR data to be signed in bstr objects, like specified by COSE in Section 2 of [RFC9052], no longer is a prerequisite. That is, cryptographic operations can _optionally_ be performed on "raw" CBOR data. Turn to Appendix B for an example of an application depending on such features.

However, to make this a reality, the following functionality MUST be provided by CBOR tools compliant with this specification:

* Decoded CBOR primitives MUST remain _immutable_, regardless if they are stand-alone or being a part of a tagged object like bigfloat (see Section 3.4.4 of [RFC8949]).

* To support _variant_ CBOR data, it MUST be possible to find out the type of a CBOR object, _before_ it is referenced.

* It MUST be possible to _add_, _delete_, and _update_ the contents of CBOR map and array objects, of decoded CBOR data.

* It MUST be possible to _reserialize_ decoded CBOR data, be it updated or not.

* Irrespective of if CBOR data is decoded, updated, or created programmatically, deterministic encoding MUST be maintained.

* Invalid or unsupported CBOR constructs, as well as CBOR data not adhering to the deterministic encoding scheme MUST be _rejected_. See also Appendix C and Appendix A.4.

As a consequence of these rules, CBOR data and application / platform-level data, MUST be _separated_ for cases where _reserialization_ could present a problem, like in this Chrome browser console example:

```
let date = new Date('2025-03-02T13:08:55.0201+03:00');
console.log(date.toISOString());
> 2025-03-02T10:08:55.020Z
```

How this separation actually is accomplished is out of scope for this specification. However, _encapsulation_ of CBOR data in _high-level_, and _self-rendering wrapper objects_, represents an established method, featured in similar tools for ASN.1. If applied to the date example above, you would get something like following, here using [CBOR.JS]:

```
// "2025-03-02T13:08:55.0201+03:00"
let cbor = CBOR.fromHex("781e323032352d30332d30325431333a30383a35352e303230312b30333a3
030");
let cborObject = CBOR.decode(cbor)

// JavaScript Date only supports milliseconds and converts time to UTC
console.log(cborObject.getDateTime().toISOString());
> 2025-03-02T10:08:55.020Z

console.log(CBOR.toHex(cborObject.encode()));  // Reencode returns identical CBOR data
> 781e323032352d30332d30325431333a30383a35352e303230312b30333a3030
```

Fully compliant CBOR::Core implementations SHOULD provide _bi-directional_ (symmetric) representations of all core objects (Section 2.1), without requiring a schema or similar.

Appendix B.2 shows an example that _updates_ and _reserializes_ decoded CBOR data.

2.3.2. Protocol Primitives

To facilitate cross-platform _protocol interoperability_, implementers of CBOR::Core compatible tools SHOULD include _decoder_ API support for the following primitive data types:

CBOR	Primitive	Comment	Note
int	Int8	8-bit signed integer	1
uint	Uint8	8-bit unsigned integer	1
int	Int16	16-bit signed integer	1
uint	Uint16	16-bit unsigned integer	1
int	Int32	32-bit signed integer	1
uint	Uint32	32-bit unsigned integer	1
int	Int64	64-bit signed integer	1
uint	Uint64	64-bit unsigned integer	1
integer	BigInt	Integer of arbitrary size	2
float16	Float16	16-bit floating-point number	3, 4
float16 / float32	Float32	32-bit floating-point number	3, 4
float	Float64	64-bit floating-point number	4
bool	Boolean	Boolean	
null	Null	Null	5
#7.nnn	Simple	Simple values	6
tstr	String	Text string	
bstr	Bytes	Byte string	
See note	DateTime	Time object expressed as a text string	7
See note	EpochTime	Time object expressed as a number	7, 8

Table 2: Protocol Primitives

1. Range testing MUST be performed using the traditional ranges for _unsigned_ respectively _two-complement_ numbers. That is, a hypothetical getUint8() MUST reject numbers outside of 0 to 255, whereas a hypothetical getInt8(), MUST reject numbers outside of -128 to 127.

2. Note that a hypothetical getBigInt() MUST also accept CBOR int objects since int is used for integers that fit in CBOR major type 0 and 1 objects. See also Appendix A.1 and Appendix C.

3. Some platforms do not natively support float32 and/or float16. In this case a hypothetical getFloat16() would need to use a bigger floating-point type for the return value.

   Note that a hypothetical getFloat16() MUST reject encountered Float32 and Float64 objects. See also Appendix C.

4. See Section 2.3.2.1.

5. Since a CBOR null typically represents the absence of a value, a decoder MUST provide a test-function, like isNull().

6. Simple values include the ranges 0-23 and 32-255. Note that bool and null actually are simple values.

7. Since CBOR lacks a native-level time object, Section 3.4 of [RFC8949] introduces two variants of time objects using the CBOR tags 0 and 1. The time objects SHOULD also be supported _without_ the tag construct.

8. EpochTime objects containing non-finite numbers MUST be rejected. See also Section 2.3.2.1.

If a call does not match the underlying CBOR type, the call MUST be rejected,

Due to considerable variations between platforms, corresponding _encoder_ API support does not appear to be meaningful to specify in detail: Java doesn’t have built-in support for unsigned integers, whereas JavaScript requires the use of the JavaScript BigInt type for dealing with 64-bit integers.

2.3.2.1. Non-Finite Numbers

Since non-finite numbers like NaN, are rarely used in application protocols, conforming CBOR::Core implementations SHOULD support an option making the decoder reject non-finite numbers. Such an option obviates the need for applications having to explicitly test decoded float objects for being finite ("regular") floating-point numbers.

In the listed reference implementations, a more elaborate arrangement has been applied, permitting application protocols to _selectively_ accept non-finite numbers, including distinguishing between simple NaNs (f97e00) and NaN with payloads like fa7f801000. Due to limited platform-native support for non-simple NaNs, the reference implementations (internally) treat non-finite numbers as a _distinct, emulated data type_, enabling the full range of floating-point numbers. Selection of the level of non-finite number support (none, simple, and full), required for a specific float object in a protocol, is accomplished through the use of different API access methods. Also see [NON-FINITE] for a concrete solution.

2.3.3. Media Type

Protocols building on CBOR::Core, are RECOMMENDED using the media type: application/cbor.

2.3.4. Diagnostic Notation

Compliant CBOR::Core implementations SHOULD include support for _bi-directional_ diagnostic notation, to facilitate:

* Generation of developer-friendly debugging and logging data

* Easy creation of test and configuration data

Note that decoders for diagnostic notation, MUST always produce deterministically encoded CBOR data, compliant with this specification. This includes _automatic_ sorting of map keys as well.

The supported notation is compliant with a subset of Section 8 of [RFC8949] (b32’ and encoding indicators were left out), but adds a few items to make diagnostic notation slightly more adapted for parsing, like single-line comments:

CBOR	Syntax	Comment	Notes
	/ _comment text_ /	Multi-line comment. Multi-line comments are treated as whitespace and may thus also be used _between_ CBOR objects.	6
	# _comment text_	Single-line comment. Single-line comments are terminated by a newline character ('\n') or EOF. Single-line comments may also terminate lines holding regular CBOR items.	6
integer	_{sign}{_0b|0o|0x_}n_	Arbitrary sized integers without fractional components or exponents. See also CBOR integer encoding. For _input_ data in diagnostic notation, binary, octal, and hexadecimal notation is also supported by prepending numbers with 0b, 0o, and 0x respectively. The latter also permit arbitrary insertions of '_' characters between digits to enable grouping of data like 0b100_000000001.	1, 2
float	_{sign}n_._n{_e±_n}_	Floating-point values MUST include a decimal point and at least one fractional digit, whereas exponents are _optional_.	1, 2
float	float'_hex-data_'	Any valid 16, 32, or 64-bit float value, including NaN with	

		payloads like float’7ff0800000000001’.	
float	NaN	Not a number (NaN) in the default CBOR encoding (f97e00).	
float	_{sign}_Infinity	Infinity.	2
bstr	h’_hex-data_’	Byte data provided in hexadecimal notation. Each byte MUST be represented by two hexadecimal digits.	3
bstr	b64’_base64-data_’	Byte data provided in base64 or base64URL notation. Padding with ’=’ characters is _optional_.	3, 6
bstr	’_text_’	Byte data provided as UTF-8 encoded text.	4, 5, 6
bstr	<< _object..._ >>	Construct holding zero or more comma-separated CBOR objects that are subsequently wrapped in a byte string.	6
tstr	"_text_"	UTF-8 encoded text string.	4, 5
bool	true|false	Boolean value.	
null	null	Null value.	
[]	[_object..._]	Array with zero or more comma-separated CBOR objects.	
{}	{ _key_:_value..._ }	Map with zero or more comma-separated key/ value pairs. Key and value pairs are expressed as CBOR objects, separated by a ’:’ character.	

#6.nnn	_n_(_object_)	Tag holding a CBOR object.	1
#7.nnn	simple(_n_)	Simple value.	1
	,	Separator character for CBOR sequences.	6

Table 3: Diagnostic Notation

1. The letter _n_ in the Syntax column denotes one or more digits.

2. The optional _{sign}_ MUST be a single hyphen ('-') character.

3. _Input only_: between tokens, the whitespace characters ' ', '\t', '\r', and '\n', are _ignored_.

4. _Input only_: inside of string quotes, the control character '\n' becomes a part of the text string. For normalizing line terminators, a single '\r' or the combination '\r\n' MUST (internally) be rewritten as '\n'. To _avoid_ getting newline characters ('\n') included in multi-line text strings, a _line continuation marker_ consisting of a backslash ('\') immediately preceding the newline may be used.

5. Text strings may also include the JavaScript compatible escape sequences '\'', '\"', '\\', '\b', '\f', '\n', '\r', '\t', and '\u_hhhh_'.

6. _Input only_.

The [PLAYGROUND] is an excellent way of getting acquainted with CBOR and diagnostic notation.

2.3.5. CBOR Sequences

Decoders compliant with this specification MUST support CBOR sequences [RFC8742].

For decoders of "true" (binary) CBOR, there are additional requirements:

* It MUST be possible to decode one CBOR object at a time.

* The decoder MUST NOT do any assumptions about the nature of unread code (it might not even be CBOR).

3. IANA Considerations

This memo includes no request to IANA.

4. Security Considerations

CBOR::Core does not introduce security issues beyond what is already applicable to [RFC8949].

Poorly written tools and applications may certainly introduce security issues, but this is out of scope for this specification.

5. References

5.1. Normative References

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

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

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

[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/info/rfc8610>.

[RFC3629] Yergeau, F., "UTF-8, a transformation format of ISO 10646", STD 63, RFC 3629, DOI 10.17487/RFC3629, November 2003, <https://www.rfc-editor.org/info/rfc3629>.

[RFC8742] Bormann, C., "Concise Binary Object Representation (CBOR) Sequences", RFC 8742, DOI 10.17487/RFC8742, February 2020, <https://www.rfc-editor.org/info/rfc8742>.

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

5.2. Informative References

[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/info/rfc9052>.

[RFC9053] Schaad, J., "CBOR Object Signing and Encryption (COSE): Initial Algorithms", RFC 9053, DOI 10.17487/RFC9053, August 2022, <https://www.rfc-editor.org/info/rfc9053>.

[RFC8785] Rundgren, A., Jordan, B., and S. Erdtman, "JSON Canonicalization Scheme (JCS)", RFC 8785, DOI 10.17487/RFC8785, June 2020, <https://www.rfc-editor.org/info/rfc8785>.

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

[CSF] "CBOR Signature Format (CSF)", <https://cyberphone.github.io/javaapi/org/webpki/cbor/doc-files/signatures.html>.

[COTX] Rundgren, A., "CBOR Object Type Extension (COTX)", <https://www.ietf.org/archive/id/draft-rundgren-cotx-04.html>.

[CONSTRAINED] "D-CBOR for Constrained Devices", <https://github.com/cyberphone/D-CBOR/blob/main/d-cbor-4-constrained-devices.md>.

[NODE.JS] "Node.js - JavaScript server", <https://nodejs.org/>.

[NON-FINITE] "Non-Finite Numbers", <https://cyberphone.github.io/CBOR.js/doc/non-finite-numbers.html>.

[CBOR.JS] "CBOR.js - CBOR for JavaScript", <https://github.com/cyberphone/CBOR.js>.

[CSF-LAB] "Online CBOR and CSF test tool", <https://test.webpki.org/csf-lab>.

[PLAYGROUND]
"Online CBOR testing tool", <https://cyberphone.github.io/CBOR.js/doc/playground.html>.

[OPENKEYSTORE]
"Java library supporting JSON, CBOR, and Crypto", <https://github.com/cyberphone/openkeystore>.

[ANDROID-CBOR]
"Android/Java library supporting CBOR and Crypto", <https://github.com/cyberphone/android-cbor>.

[OPENAPI] "The OpenAPI Initiative", <https://openapis.org/>.

[CREDENTIALS]
Sporny (et al), M., "Verifiable Credential Data Integrity 1.0", 2025, <https://www.w3.org/TR/vc-data-integrity/>.

[CBOR.ME] Bormann, C., "Online CBOR testing tool", <https://cbor.me/>.

[ECMASCRIPT]
Ecma International, "ECMAScript 2024 Language Specification", Standard ECMA-262, 15th Edition, June 2024, <https://www.ecma-international.org/publications/standards/Ecma-262.htm>.

[WALLET] Rundgren, A., "Defensive publication: Partial Encryption, Full Signature", <https://cyberphone.github.io/doc/defensive-publications/partial-encryption-full-signature.pdf>.

Appendix A. Deterministic Encoding Samples

A.1. Integers

This _normative_ section holds a selection of CBOR integer values, with an emphasize on edge cases.

Diagnostic Notation	CBOR Encoding	Comment
0	00	Smallest positive implicit int
-1	20	Smallest

		negative implicit int
23	17	Largest positive implicit int
-24	37	Largest negative implicit int
24	1818	Smallest positive one-byte int
-25	3818	Smallest negative one-byte int
255	18ff	Largest positive one-byte int
-256	38ff	Largest negative one-byte int
256	190100	Smallest positive two-byte int
-257	390100	Smallest negative two-byte int
65535	19ffff	Largest positive two-byte int
-65536	39ffff	Largest negative two-byte int
65536	1a00010000	Smallest positive four-byte int
-65537	3a00010000	Smallest

		negative four-byte int
4294967295	1affffffff	Largest positive four-byte int
-4294967296	3affffffff	Largest negative four-byte int
4294967296	1b0000000100000000	Smallest positive eight-byte int
-4294967297	3b0000000100000000	Smallest negative eight-byte int
18446744073709551615	1bffffffffffffffff	Largest positive eight-byte int
-18446744073709551616	3bffffffffffffffff	Largest negative eight-byte int
18446744073709551616	c249010000000000000000	Smallest positive bigint
-18446744073709551617	c349010000000000000000	Smallest negative bigint

Table 4: Integers

A.2. Floating-Point Numbers

This _normative_ section holds a selection of [IEEE754] 16, 32, and 64-bit values, with an emphasize on edge cases.

The textual representation of the values is based on the serialization method for the Number data type, defined by [ECMASCRIPT] with one change: to comply with diagnostic notation (Section 2.3.4), all values are expressed as floating-point numbers. The rationale for using [ECMASCRIPT] serialization is because it is supposed to generate the shortest and most correct representation of [IEEE754] numbers.

Diagnostic Notation	CBOR Encoding	Comment
0.0	f90000	Zero
-0.0	f98000	Negative zero
Infinity	f97c00	Infinity
-Infinity	f9fc00	Negative infinity
NaN	f97e00	Not a number
5.960464477539063e-8	f90001	Smallest positive subnormal float16
0.00006097555160522461	f903ff	Largest positive subnormal float16
0.00006103515625	f90400	Smallest positive float16
65504.0	f97bff	Largest positive float16
1.401298464324817e-45	fa00000001	Smallest positive subnormal float32
1.1754942106924411e-38	fa007fffff	Largest positive

		subnormal float32
1.1754943508222875e-38	fa00800000	Smallest positive float32
3.4028234663852886e+38	fa7f7fffff	Largest positive float32
5.0e-324	fb0000000000000001	Smallest positive subnormal float64
2.225073858507201e-308	fb000fffffffffffff	Largest positive subnormal float64
2.2250738585072014e-308	fb0010000000000000	Smallest positive float64
1.7976931348623157e+308	fb7fefffffffffffff	Largest positive float64
-0.0000033333333333333333	fbbecbf647612f3696	Randomly selected number
10.559998512268066	fa4128f5c1	-"-
10.559998512268068	fb40251eb820000001	Next in succession
295147905179352830000.0	fa61800000	2^68 (diagnostic notation truncates precision)
2.0	f94000	Number without a fractional part

-5.960464477539063e-8	f98001	Smallest negative subnormal float16
-5.960464477539062e-8	fbbe6fffffffffffff	Adjacent smallest negative subnormal float16
-5.960464477539064e-8	fbbe70000000000001	-"-
-5.960465188081798e-8	fab3800001	-"-
0.0000609755516052246	fb3f0ff7ffffffffff	Adjacent largest subnormal float16
0.000060975551605224616	fb3f0ff80000000001	-"-
0.000060975555243203416	fa387fc001	-"-
0.00006103515624999999	fb3f0fffffffffffff	Adjacent smallest float16
0.00006103515625000001	fb3f10000000000001	-"-
0.00006103516352595761	fa38800001	-"-
65503.99999999999	fb40effbffffffffff	Adjacent largest float16
65504.00000000001	fb40effc0000000001	-"-
65504.00390625	fa477fe001	-"-
1.4012984643248169e-45	fb369fffffffffffff	Adjacent smallest subnormal float32
1.4012984643248174e-45	fb36a0000000000001	-"-

1.175494210692441e-38	fb380fffffbfffffff	Adjacent largest subnormal float32
1.1754942106924412e-38	fb380fffffc0000001	-"-
1.1754943508222874e-38	fb380fffffffffffff	Adjacent smallest float32
1.1754943508222878e-38	fb3810000000000001	-"-
3.4028234663852882e+38	fb47efffffdfffffff	Adjacent largest float32
3.402823466385289e+38	fb47efffffe0000001	-"-

Table 5: Floating-Point Numbers

A.3. Miscellaneous Items

This _normative_ section holds a selection of miscellaneous CBOR objects and their encoding.

Diagnostic Notation	CBOR Encoding	Comment
true	f5	Boolean true
null	f6	Null
simple(99)	f863	Simple value
0("2025-03-30T12:24:16Z")	c074323032352d30332d33 305431323a32343a31365a	ISO date/ time
[1, [2, 3], [4, 5]]	8301820203820405	Array combinations
{ "a": 0, "b": 1, "aa": 2 }	a361610161620262616103	Map object
h'48656c6c6f2043424f5221'	4b48656c6c6f2043424f5221	Binary string
" science"	6cf09f9a8020736369656e6365	Text string with emoji
float'7f800001'	fa7f800001	NaN with payload

Table 6: Miscellaneous Items

A.4. Invalid Encodings

The following table holds a selection of CBOR-encoded objects that not permitted by CBOR::Core in _deterministic_ mode.

CBOR Encoding	Diagnostic Notation	Comment	Notes
a2616201616100	{ "b": 1, "a": 0 }	Improper map key ordering	1
98020405	[4, 5]	Improper array length indicator	1

1900ff	255	Number with leading zero bytes	1
c34a00010000000000000000	-18446744073709551617	Number with leading zero bytes	1
Fa41280000	10.5	not using shortest encoding	1
fa7fc00000	NaN	not using shortest encoding	1
fa7fffe000	float'7fff'	not using shortest encoding	1
c243010000	65536	Incorrect value for bigint	1
5f4101420203ff	(_ h'01', h'0203')	Indefinite length object	2
fc		Reserved	
f818		Invalid simple value	
5b0010000000000000		Extremely large bstr length indicator: 4503599627370496	

Table 7: Invalid Encodings

1. Supported by the measures mentioned in Appendix C.

2. Unsupported since it is in conflict with _deterministically_ encoded CBOR.

Appendix B. Embedded Signatures

This is a _non-normative_ appendix showing how CBOR::Core can be used for supporting embedded signatures.

The primary advantages with _embedded_ signatures compared to _enveloping_ signatures (like used by COSE [RFC9052]), include:

* Keeping the _structure_ of the original (unsigned) data intact, by simply making signatures an additional attribute.

* Enabling top-level, _object identifiers_ to become a part of the signed data as well:

  ```
  123456789({                              # CBOR tag (objectId)
    1: "This is not rocket science!",      # Object instance data
    2: [38.8882, -77.01988],               #          ""
    simple(99): signature covering the entire object
  })
  ```

  See also [COTX].

* Permitting signing CBOR data and associated security attributes (aka "headers"), _in one go_, without having to wrap data in CBOR "bstr" objects. Non-wrapped data also makes debugging and documentation easier.

Embedded signatures are for example featured in Verified Credentials [CREDENTIALS]. A drawback with designs based on JSON [RFC8259] is that they rely on _canonicalization schemes_ like JCS [RFC8785], that require specialized encoders and decoders, whereas CBOR::Core works "straight out of the box".

B.1. Sample Signature

Although this specification is not "married" to any particular signature schema, the following example uses the CBOR Signature Format [CSF]. For the sake of simplicity, the example uses an HMAC (see Appendix B.1.4) as signature algorithm.

For a more sophisticated use of CBOR::Core, combining signatures and encryption, see [WALLET].

B.1.1. Unsigned Data

Imagine you have a CBOR map object like the following that you want to sign:

```
{
  1: "data",
  2: "more data"
}
```

B.1.2. Signature Process

This section describes the steps required for adding an embedded signature to the CBOR map object in Appendix B.1.1. To avoid confusing CBOR map keys with cryptographic keys, the former are referred to as "labels".

1. Add an empty CSF container (a CBOR map) to the unsigned CBOR map using the CSF container label simple(99).

2. Add the designated signature algorithm to the CSF container using the CSF algorithm label (1).

3. _Optional_. Add other signature meta data to the CSF container. Not used in the example.

4. Generate a signature by invoking a (hypothetical) signature method with the following arguments:

   * the designated signature key.

   * the designated signature algorithm.

   * the _deterministic encoding_ of the current CBOR object in its _entirety_. In the example that would be a301646461746102696d6f7265206461746120a10105, if expressed in hex code.

5. Add the returned signature value to the CSF container using the CSF signature label (6).

The result after the final step (using the parameters from Appendix B.1.4), should match the following CBOR object:

```
{
  1: "data",
  2: "more data",
  simple(99): {
    1: 5,
    6: h'237e674c7be1818ddd7eaacf40ca80415b9ad816880751d2136c45385207420c'
  }
}
```

Note that the signature covers the _entire_ CBOR object except for the CSF signature value and label (6).

B.1.3. Validation Process

In order to validate the embedded signature created in the Appendix B.1.2, the following steps are performed:

1. Fetch a _reference_ to the CSF container using the CSF container label simple(99). Next perform the following operations using the reference:

    1. Retrieve the signature algorithm using the CSF algorithm label (1).

    2. Retrieve the signature value using the CSF algorithm label (6).

    3. Remove the CSF algorithm label (6) and its associated value.

    Now we should have exactly the same CBOR object as we had _before_ step #4 in Appendix B.1.2. That is:

```
{
  1: "data",
  2: "more data",
  -1: {
    1: 5
  }
}
```

2. Validate the signature data by invoking a (hypothetical) signature validation method with the following arguments:

    * the designated signature key (in the example taken from Appendix B.1.4).

    * the signature algorithm retrieved in step #1.

    * the signature value retrieved in step #1.

    * the _deterministic encoding_ of the current CBOR object in its _entirety_.

Note: this is a "bare-bones" validation process, lacking the ruggedness of a real-world implementation.

B.1.4. Example Parameters

The signature and validation processes depend on the COSE [RFC9053] algorithm "HMAC 256/256" and an associated 256-bit key, here provided in hex code:

```
7fdd851a3b9d2dafc5f0d00030e22b9343900cd42ede4948568a4a2ee655291a
```

B.2. Code Example

Using a JavaScript implementation [CBOR.JS] of CBOR::Core, together with Node.js [NODE.JS], basic signature creation and validation supporting the example in Appendix B.1, could be performed by the following code:

```
// hmac.mjs
import CBOR from 'cbor-object';
const crypto = await import('node:crypto');

// Application independent CSF constants
const CSF_CONTAINER_LBL = CBOR.Simple(99);
const CSF_ALG_LBL = CBOR.Int(1);
const CSF_SIG_LBL = CBOR.Int(6);

// COSE => Node.js algorithm translation
const HASH_ALGORITHMS = new Map()
    .set(5, "sha256").set(6, "sha384").set(7, "sha512");

function hmac(coseAlg, key, data) {
    let alg = HASH_ALGORITHMS.get(coseAlg);
    if (alg === undefined) throw "Unknown alg: " + coseAlg;
    return crypto.createHmac(alg, key).update(data).digest();
}

const SHARED_KEY = crypto.createSecretKey(
    '7fdd851a3b9d2dafc5f0d00030e22b9343900cd42ede4948568a4a2ee655291a', 'hex');

const APP_P1_LBL  = CBOR.Int(1);                   // Application label
const APP_P2_LBL  = CBOR.Int(2);                   //        ""

////////////////////////////////////
// Create an unsigned CBOR object //
////////////////////////////////////
let object = CBOR.Map()
    .set(APP_P1_LBL, CBOR.String("data"))          // Application data
    .set(APP_P2_LBL, CBOR.String("more data"));    //        ""

////////////////////////////////////////
```

```
// Add a signature to the CBOR object //
////////////////////////////////////////
const COSE_ALG = 5;                                   // Selected HMAC algorithm

let csf = CBOR.Map()                                  // Create CSF container and
    .set(CSF_ALG_LBL, CBOR.Int(COSE_ALG));            // add COSE algorithm to it
object.set(CSF_CONTAINER_LBL, csf);                   // Add CSF container to object
let sig = hmac(COSE_ALG,                              // Generate signature over
               SHARED_KEY,                            // the current object
               object.encode());                      // encode(): all we got so far
csf.set(CSF_SIG_LBL, CBOR.Bytes(sig));                // Add signature to CSF container
let cborBinary = object.encode();                     // Return CBOR as an Uint8Array

console.log(object.toString());                       // Show in Diagnostic Notation

/////////////////////////////////////
// Validate the signed CBOR object //
/////////////////////////////////////
object = CBOR.decode(cborBinary);                     // Decode CBOR object
csf = object.get(CSF_CONTAINER_LBL);                  // Get CSF container
let alg = csf.get(CSF_ALG_LBL).getInt32();            // Get COSE algorithm
let readSig = csf.remove(CSF_SIG_LBL).getBytes();  // Get and REMOVE signature value
let actualSig = hmac(alg,                             // Calculate signature over
                     SHARED_KEY,                      // the current object
                     object.encode());                // encode(): all but the signature
if (CBOR.compareArrays(readSig, actualSig)) {         // HMAC validation
    throw "Signature did not validate";
}
// Validated object, access the "payload":
let p1 = object.get(APP_P1_LBL).getString();          // p1 should now contain "data"
```

Note that this code depends heavily on the API features outlined in Section 2.3.1.

Appendix C. Backward Compatibility

It is assumed that _most_ systems using CBOR are able to process an (_application specific_), selection of CBOR data items that are encoded in compliance with [RFC8949]. Since the deterministic encoding scheme mandated by CBOR::Core, also is compliant with [RFC8949] (as well as with [CBOR.ME]), there should be no major interoperability issues. That is, if the previous assumption actually is correct

However, in the _other_ direction (CBOR::Core tools processing data from systems using "legacy" CBOR encoding schemes), the situation is likely to be considerably more challenging since deterministic encoding "by design" is _strict_. Due to this potential obstacle, implementers of CBOR::Core tools, are RECOMMENDED to offer _decoder_ options that permit "relaxing" the rigidness of deterministic encoding with respect to:

Numbers:
Numbers MUST still be compliant with [RFC8949], including "Rule 2" in section 4.2.2.

Sorted maps:
Duplicate keys MUST still be rejected.

Note that regardless of the format of decoded CBOR data, a compliant CBOR::Core implementation MUST maintain deterministic encoding. See also Appendix A.4.

Appendix D. Compatible Online Tools

For testing and learning about CBOR::Core, there are currently a number of compatible online tools (subject to availability...).

[PLAYGROUND]:
Browser-based CBOR "playground"

[CSF-LAB]:
Server-based CBOR and [CSF] test system

Appendix E. Compatible Implementations

For using CBOR::Core in applications, there are currently a number of compatible libraries.

[CBOR.JS]:
JavaScript-based implementation supporting browsers as well as [NODE.JS]

[OPENKEYSTORE]:
Java-based implementation that also supports [CSF]

[ANDROID-CBOR]:
Android Java-based implementation that also supports [CSF]

Document History

// RFC Editor: Please remove this section before publication

* 00. First cut.

* 01. Editorial. Changed order of columns in invalid encoding.

* 02. Editorial. "unwrapped" changed to "non-wrapped".

* 03:

  Tweaking the abstract.

  Protocol Primitives sub-section added.

  Diagnostic Notation sub-section added.

  Updated CBOR Tool Requirements

  Updated code example to actually use crypto

  Updated Acknowledgements.

  Updated Security Considerations.

* 04:

  Minor addition in CBOR tools

  Updated Acknowledgements

* 05:

  Regression bug fix

* 06:

  Media type added

* 07->00:

  Renamed from "Universal CBOR" to "CBOR Base"

  Design Goals added

  CBOR Sequences added

* 01:

  #7.nnn (simple) added

  Language nits

* 02->00:

  Renamed from "CBOR Base" to "CBOR Core"

  Language nits

  Miscellaneous Items added

* 01:

  Language nits

  <table align="left">

* 02:

  Editorial

* 03:

  Added Date decode/reencode example

* 04:

  Editorial

  Added bstr and tstr to Protocol Primitives

* 05:

  Enveloped => Embedded (signature)

* 06:

  Updated Acknowledgements

* 07:

  Dropped CBOR/c. Now it is just CBOR::Core

  Introduced API Level Considerations

  Updated Date example, and added Rule 2 to Supporting Existing Systems

* 08:

  Elaborated on "levels"

CBOR/Core => CBOR::Core (showing its close ties to software)

* 09:

  Embedded signature sample => simple(99)

* 10:

  Editorial

  CBOR Profile => CBOR Platform Profile

* 11:

  Editorial

  Supporting Existing Systems => Backward Compatibility

* 12:

  NaN with payloads added

* 13:

  Editorial

  Non-finite number support

Acknowledgements

For verifying the correctness of the encoding scheme, the [CBOR.ME] on-line CBOR tool, by the [RFC8949] author, Carsten Bormann, proved to be invaluable.

Non-exhaustive list of people who directly (and sometimes indirectly) contributed to this specification include: Carsten Bormann, Alan DeKok, Vadim Goncharov, Joe Hildebrand, Eliot Lear, Laurence Lundblade, Rohan Mahy, Michael Richardson, Gran Selander, and Orie Steele.

Author’s Address

Anders Rundgren (editor)
Independent
Montpellier
France
Email: anders.rundgren.net@gmail.com
URI: https://www.linkedin.com/in/andersrundgren/