

Network Working Group
Internet-Draft
Intended status: Standards Track
Expires: January 3, 2020

C. Bormann
Universitaet Bremen TZI
P. Hoffman
ICANN
July 02, 2019

Concise Binary Object Representation (CBOR)
draft-ietf-cbor-7049bis-06

Abstract

The Concise Binary Object Representation (CBOR) is a data format whose design goals include the possibility of extremely small code size, fairly small message size, and extensibility without the need for version negotiation. These design goals make it different from earlier binary serializations such as ASN.1 and MessagePack.

This document obsoletes [RFC 7049](#).

Contributing

This document is being worked on in the CBOR Working Group. Please contribute on the mailing list there, or in the GitHub repository for this draft: <https://github.com/cbor-wg/CBORbis>

The charter for the CBOR Working Group says that the WG will update [RFC 7049](#) to fix verified errata. Security issues and clarifications may be addressed, but changes to this document will ensure backward compatibility for popular deployed codebases. This document will be targeted at becoming an Internet Standard.

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 January 3, 2020.

Copyright Notice

Copyright (c) 2019 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to [BCP 78](https://trustee.ietf.org/license-info) 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 Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

Table of Contents

1.	Introduction	3
1.1.	Objectives	4
1.2.	Terminology	5
2.	CBOR Data Models	7
2.1.	Extended Generic Data Models	8
2.2.	Specific Data Models	8
3.	Specification of the CBOR Encoding	9
3.1.	Major Types	10
3.2.	Indefinite Lengths for Some Major Types	12
3.2.1.	The "break" Stop Code	12
3.2.2.	Indefinite-Length Arrays and Maps	12
3.2.3.	Indefinite-Length Byte Strings and Text Strings	14
3.3.	Floating-Point Numbers and Values with No Content	15
3.4.	Optional Tagging of Items	17
3.4.1.	Date and Time	19
3.4.2.	Standard Date/Time String	19
3.4.3.	Epoch-based Date/Time	19
3.4.4.	Bignums	20
3.4.5.	Decimal Fractions and Bigfloats	21
3.4.6.	Content Hints	22
3.4.6.1.	Encoded CBOR Data Item	22
3.4.6.2.	Expected Later Encoding for CBOR-to-JSON Converters	23
3.4.6.3.	Encoded Text	23
3.4.7.	Self-Described CBOR	24
4.	Serialization Considerations	25
4.1.	Preferred Serialization	25
4.2.	Deterministically Encoded CBOR	26
4.2.1.	Core Deterministic Encoding Requirements	26
4.2.2.	Additional Deterministic Encoding Considerations	27
4.2.3.	Length-first map key ordering	28

5.	Creating CBOR-Based Protocols	29
5.1.	CBOR in Streaming Applications	30
5.2.	Generic Encoders and Decoders	30
5.3.	Invalid Items	31
5.4.	Handling Unknown Simple Values and Tags	32
5.5.	Numbers	32
5.6.	Specifying Keys for Maps	33
5.6.1.	Equivalence of Keys	34
5.7.	Undefined Values	35
5.8.	Strict Decoding Mode	35
6.	Converting Data between CBOR and JSON	36
6.1.	Converting from CBOR to JSON	36
6.2.	Converting from JSON to CBOR	38
7.	Future Evolution of CBOR	39
7.1.	Extension Points	39
7.2.	Curating the Additional Information Space	40
8.	Diagnostic Notation	40
8.1.	Encoding Indicators	41
9.	IANA Considerations	42
9.1.	Simple Values Registry	42
9.2.	Tags Registry	42
9.3.	Media Type ("MIME Type")	43
9.4.	CoAP Content-Format	44
9.5.	The +cbor Structured Syntax Suffix Registration	44
10.	Security Considerations	45
11.	References	47
11.1.	Normative References	47
11.2.	Informative References	48
Appendix A.	Examples	50
Appendix B.	Jump Table	54
Appendix C.	Pseudocode	57
Appendix D.	Half-Precision	59
Appendix E.	Comparison of Other Binary Formats to CBOR's Design Objectives	60
E.1.	ASN.1 DER, BER, and PER	61
E.2.	MessagePack	61
E.3.	BSON	62
E.4.	MSDTP: RFC 713	62
E.5.	Conciseness on the Wire	62
Appendix F.	Changes from RFC 7049	63
	Acknowledgements	63
	Authors' Addresses	64

[1.](#) Introduction

There are hundreds of standardized formats for binary representation of structured data (also known as binary serialization formats). Of those, some are for specific domains of information, while others are

generalized for arbitrary data. In the IETF, probably the best-known formats in the latter category are ASN.1's BER and DER [[ASN.1](#)].

The format defined here follows some specific design goals that are not well met by current formats. The underlying data model is an extended version of the JSON data model [[RFC8259](#)]. It is important to note that this is not a proposal that the grammar in [RFC 8259](#) be extended in general, since doing so would cause a significant backwards incompatibility with already deployed JSON documents. Instead, this document simply defines its own data model that starts from JSON.

[Appendix E](#) lists some existing binary formats and discusses how well they do or do not fit the design objectives of the Concise Binary Object Representation (CBOR).

This document obsoletes [[RFC7049](#)].

[1.1](#). Objectives

The objectives of CBOR, roughly in decreasing order of importance, are:

1. The representation must be able to unambiguously encode most common data formats used in Internet standards.
 - * It must represent a reasonable set of basic data types and structures using binary encoding. "Reasonable" here is largely influenced by the capabilities of JSON, with the major addition of binary byte strings. The structures supported are limited to arrays and trees; loops and lattice-style graphs are not supported.
 - * There is no requirement that all data formats be uniquely encoded; that is, it is acceptable that the number "7" might be encoded in multiple different ways.
2. The code for an encoder or decoder must be able to be compact in order to support systems with very limited memory, processor power, and instruction sets.
 - * An encoder and a decoder need to be implementable in a very small amount of code (for example, in class 1 constrained nodes as defined in [[RFC7228](#)]).
 - * The format should use contemporary machine representations of data (for example, not requiring binary-to-decimal conversion).

3. Data must be able to be decoded without a schema description.
 - * Similar to JSON, encoded data should be self-describing so that a generic decoder can be written.
4. The serialization must be reasonably compact, but data compactness is secondary to code compactness for the encoder and decoder.
 - * "Reasonable" here is bounded by JSON as an upper bound in size, and by implementation complexity maintaining a lower bound. Using either general compression schemes or extensive bit-fiddling violates the complexity goals.
5. The format must be applicable to both constrained nodes and high-volume applications.
 - * This means it must be reasonably frugal in CPU usage for both encoding and decoding. This is relevant both for constrained nodes and for potential usage in applications with a very high volume of data.
6. The format must support all JSON data types for conversion to and from JSON.
 - * It must support a reasonable level of conversion as long as the data represented is within the capabilities of JSON. It must be possible to define a unidirectional mapping towards JSON for all types of data.
7. The format must be extensible, and the extended data must be decodable by earlier decoders.
 - * The format is designed for decades of use.
 - * The format must support a form of extensibility that allows fallback so that a decoder that does not understand an extension can still decode the message.
 - * The format must be able to be extended in the future by later IETF standards.

1.2. Terminology

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

[BCP 14](#) [[RFC2119](#)] [[RFC8174](#)] when, and only when, they appear in all capitals, as shown here.

The term "byte" is used in its now-customary sense as a synonym for "octet". All multi-byte values are encoded in network byte order (that is, most significant byte first, also known as "big-endian").

This specification makes use of the following terminology:

Data item: A single piece of CBOR data. The structure of a data item may contain zero, one, or more nested data items. The term is used both for the data item in representation format and for the abstract idea that can be derived from that by a decoder.

Decoder: A process that decodes a well-formed CBOR data item and makes it available to an application. Formally speaking, a decoder contains a parser to break up the input using the syntax rules of CBOR, as well as a semantic processor to prepare the data in a form suitable to the application.

Encoder: A process that generates the representation format of a CBOR data item from application information.

Data Stream: A sequence of zero or more data items, not further assembled into a larger containing data item. The independent data items that make up a data stream are sometimes also referred to as "top-level data items".

Well-formed: A data item that follows the syntactic structure of CBOR. A well-formed data item uses the initial bytes and the byte strings and/or data items that are implied by their values as defined in CBOR and does not include following extraneous data. CBOR decoders by definition only return contents from well-formed data items.

Valid: A data item that is well-formed and also follows the semantic restrictions that apply to CBOR data items.

Stream decoder: A process that decodes a data stream and makes each of the data items in the sequence available to an application as they are received.

Where bit arithmetic or data types are explained, this document uses the notation familiar from the programming language C, except that "***" denotes exponentiation. Similar to the "0x" notation for hexadecimal numbers, numbers in binary notation are prefixed with "0b". Underscores can be added to such a number solely for readability, so 0b00100001 (0x21) might be written 0b001_00001 to

emphasize the desired interpretation of the bits in the byte; in this case, it is split into three bits and five bits. Encoded CBOR data items are sometimes given in the "0x" or "0b" notation; these values are first interpreted as numbers as in C and are then interpreted as byte strings in network byte order, including any leading zero bytes expressed in the notation.

2. CBOR Data Models

CBOR is explicit about its generic data model, which defines the set of all data items that can be represented in CBOR. Its basic generic data model is extensible by the registration of simple type values and tags. Applications can then subset the resulting extended generic data model to build their specific data models.

Within environments that can represent the data items in the generic data model, generic CBOR encoders and decoders can be implemented (which usually involves defining additional implementation data types for those data items that do not already have a natural representation in the environment). The ability to provide generic encoders and decoders is an explicit design goal of CBOR; however many applications will provide their own application-specific encoders and/or decoders.

In the basic (un-extended) generic data model, a data item is one of:

- o an integer in the range $-2^{64}..2^{64}-1$ inclusive
- o a simple value, identified by a number between 0 and 255, but distinct from that number
- o a floating point value, distinct from an integer, out of the set representable by IEEE 754 binary64 (including non-finites) [[IEEE754](#)]
- o a sequence of zero or more bytes ("byte string")
- o a sequence of zero or more Unicode code points ("text string")
- o a sequence of zero or more data items ("array")
- o a mapping (mathematical function) from zero or more data items ("keys") each to a data item ("values"), ("map")
- o a tagged data item, comprising a tag (an integer in the range $0..2^{64}-1$) and a value (a data item)

Note that integer and floating-point values are distinct in this model, even if they have the same numeric value.

Also note that serialization variants, such as number of bytes of the encoded floating value, or the choice of one of the ways in which an integer, the length of a text or byte string, the number of elements in an array or pairs in a map, or a tag value, (collectively "the argument", see [Section 3](#)) can be encoded, are not visible at the generic data model level.

2.1. Extended Generic Data Models

This basic generic data model comes pre-extended by the registration of a number of simple values and tags right in this document, such as:

- o "false", "true", "null", and "undefined" (simple values identified by 20..23)
- o integer and floating point values with a larger range and precision than the above (tags 2 to 5)
- o application data types such as a point in time or an [RFC 3339](#) date/time string (tags 1, 0)

Further elements of the extended generic data model can be (and have been) defined via the IANA registries created for CBOR. Even if such an extension is unknown to a generic encoder or decoder, data items using that extension can be passed to or from the application by representing them at the interface to the application within the basic generic data model, i.e., as generic values of a simple type or generic tagged items.

In other words, the basic generic data model is stable as defined in this document, while the extended generic data model expands by the registration of new simple values or tags, but never shrinks.

While there is a strong expectation that generic encoders and decoders can represent "false", "true", and "null" ("undefined" is intentionally omitted) in the form appropriate for their programming environment, implementation of the data model extensions created by tags is truly optional and a matter of implementation quality.

2.2. Specific Data Models

The specific data model for a CBOR-based protocol usually subsets the extended generic data model and assigns application semantics to the data items within this subset and its components. When documenting

such specific data models, where it is desired to specify the types of data items, it is preferred to identify the types by the names they have in the generic data model ("negative integer", "array") instead of by referring to aspects of their CBOR representation ("major type 1", "major type 4").

Specific data models can also specify what values (including values of different types) are equivalent for the purposes of map keys and encoder freedom. For example, in the generic data model, a valid map MAY have both "0" and "0.0" as keys, and an encoder MUST NOT encode "0.0" as an integer (major type 0, [Section 3.1](#)). However, if a specific data model declares that floating point and integer representations of integral values are equivalent, using both map keys "0" and "0.0" in a single map would be considered duplicates and so invalid, and an encoder could encode integral-valued floats as integers or vice versa, perhaps to save encoded bytes.

3. Specification of the CBOR Encoding

A CBOR data item ([Section 2](#)) is encoded to or decoded from a byte string carrying a well-formed encoded data item as described in this section. The encoding is summarized in Table 5. An encoder MUST produce only well-formed encoded data items. A decoder MUST NOT return a decoded data item when it encounters input that is not a well-formed encoded CBOR data item (this does not detract from the usefulness of diagnostic and recovery tools that might make available some information from a damaged encoded CBOR data item).

The initial byte of each encoded data item contains both information about the major type (the high-order 3 bits, described in [Section 3.1](#)) and additional information (the low-order 5 bits). With a few exceptions, the additional information's value describes how to load an unsigned integer "argument":

- Less than 24: The argument's value is the value of the additional information.
- 24, 25, 26, or 27: The argument's value is held in the following 1, 2, 4, or 8 bytes, respectively, in network byte order. For major type 7 and additional information value 25, 26, 27, these bytes are not used as an integer argument, but as a floating point value (see [Section 3.3](#)).
- 28, 29, 30: These values are reserved for future additions to the CBOR format. In the present version of CBOR, the encoded item is not well-formed.

31: No argument value is derived. If the major type is 0, 1, or 6, the encoded item is not well-formed. For major types 2 to 5, the item's length is indefinite, and for major type 7, the byte does not constitute a data item at all but terminates an indefinite length item; both are described in [Section 3.2](#).

The meaning of this argument depends on the major type. For example, in major type 0, the argument is the value of the data item itself (and in major type 1 the value of the data item is computed from the argument); in major type 2 and 3 it gives the length of the string data in bytes that follows; and in major types 4 and 5 it is used to determine the number of data items enclosed.

If the encoded sequence of bytes ends before the end of a data item, that item is not well-formed. If the encoded sequence of bytes still has bytes remaining after the outermost encoded item is decoded, that encoding is not a single well-formed CBOR item; depending on the application, the decoder may either treat the encoding as not well-formed or just identify the start of the remaining bytes to the application.

A CBOR decoder implementation can be based on a jump table with all 256 defined values for the initial byte (Table 5). A decoder in a constrained implementation can instead use the structure of the initial byte and following bytes for more compact code (see [Appendix C](#) for a rough impression of how this could look).

3.1. Major Types

The following lists the major types and the additional information and other bytes associated with the type.

Major type 0: an integer in the range $0..2^{64}-1$ inclusive. The value of the encoded item is the argument itself. For example, the integer 10 is denoted as the one byte 0b000_01010 (major type 0, additional information 10). The integer 500 would be 0b000_11001 (major type 0, additional information 25) followed by the two bytes 0x01f4, which is 500 in decimal.

Major type 1: a negative integer in the range $-2^{64}..-1$ inclusive. The value of the item is -1 minus the argument. For example, the integer -500 would be 0b001_11001 (major type 1, additional information 25) followed by the two bytes 0x01f3, which is 499 in decimal.

Major type 2: a byte string. The number of bytes in the string is equal to the argument. For example, a byte string whose length is 5 would have an initial byte of 0b010_00101 (major type 2,

additional information 5 for the length), followed by 5 bytes of binary content. A byte string whose length is 500 would have 3 initial bytes of 0b010_11001 (major type 2, additional information 25 to indicate a two-byte length) followed by the two bytes 0x01f4 for a length of 500, followed by 500 bytes of binary content.

Major type 3: a text string ([Section 2](#)), encoded as UTF-8 ([RFC3629](#)). The number of bytes in the string is equal to the argument. A string containing an invalid UTF-8 sequence is well-formed but invalid. This type is provided for systems that need to interpret or display human-readable text, and allows the differentiation between unstructured bytes and text that has a specified repertoire and encoding. In contrast to formats such as JSON, the Unicode characters in this type are never escaped. Thus, a newline character (U+000A) is always represented in a string as the byte 0x0a, and never as the bytes 0x5c6e (the characters "\" and "n") or as 0x5c7530303061 (the characters "\", "u", "0", "0", "0", and "a").

Major type 4: an array of data items. Arrays are also called lists, sequences, or tuples. The argument is the number of data items in the array. Items in an array do not need to all be of the same type. For example, an array that contains 10 items of any type would have an initial byte of 0b100_01010 (major type of 4, additional information of 10 for the length) followed by the 10 remaining items.

Major type 5: a map of pairs of data items. Maps are also called tables, dictionaries, hashes, or objects (in JSON). A map is comprised of pairs of data items, each pair consisting of a key that is immediately followed by a value. The argument is the number of `_pairs_` of data items in the map. For example, a map that contains 9 pairs would have an initial byte of 0b101_01001 (major type of 5, additional information of 9 for the number of pairs) followed by the 18 remaining items. The first item is the first key, the second item is the first value, the third item is the second key, and so on. A map that has duplicate keys may be well-formed, but it is not valid, and thus it causes indeterminate decoding; see also [Section 5.6](#).

Major type 6: a tagged data item whose tag is the argument and whose value is the single following encoded item. See [Section 3.4](#).

Major type 7: floating-point numbers and simple values, as well as the "break" stop code. See [Section 3.3](#).

These eight major types lead to a simple table showing which of the 256 possible values for the initial byte of a data item are used (Table 5).

In major types 6 and 7, many of the possible values are reserved for future specification. See [Section 9](#) for more information on these values.

[3.2.](#) Indefinite Lengths for Some Major Types

Four CBOR items (arrays, maps, byte strings, and text strings) can be encoded with an indefinite length using additional information value 31. This is useful if the encoding of the item needs to begin before the number of items inside the array or map, or the total length of the string, is known. (The application of this is often referred to as "streaming" within a data item.)

Indefinite-length arrays and maps are dealt with differently than indefinite-length byte strings and text strings.

[3.2.1.](#) The "break" Stop Code

The "break" stop code is encoded with major type 7 and additional information value 31 (0b111_11111). It is not itself a data item: it is just a syntactic feature to close an indefinite-length item.

If the "break" stop code appears anywhere where a data item is expected, other than directly inside an indefinite-length string, array, or map -- for example directly inside a definite-length array or map -- the enclosing item is not well-formed.

[3.2.2.](#) Indefinite-Length Arrays and Maps

Indefinite-length arrays and maps are represented using their major type with the additional information value of 31, followed by an arbitrary-length sequence of items for an array or key/value pairs for a map, followed by the "break" stop code ([Section 3.2.1](#)). In other words, indefinite-length arrays and maps look identical to other arrays and maps except for beginning with the additional information value of 31 and ending with the "break" stop code.

If the break stop code appears after a key in a map, in place of that key's value, the map is not well-formed.

There is no restriction against nesting indefinite-length array or map items. A "break" only terminates a single item, so nested indefinite-length items need exactly as many "break" stop codes as there are type bytes starting an indefinite-length item.

For example, assume an encoder wants to represent the abstract array [1, [2, 3], [4, 5]]. The definite-length encoding would be 0x8301820203820405:

```
83      -- Array of length 3
  01      -- 1
  82      -- Array of length 2
    02    -- 2
    03    -- 3
  82      -- Array of length 2
    04    -- 4
    05    -- 5
```

Indefinite-length encoding could be applied independently to each of the three arrays encoded in this data item, as required, leading to representations such as:

0x9f018202039f0405ffff

```
9F      -- Start indefinite-length array
  01      -- 1
  82      -- Array of length 2
    02    -- 2
    03    -- 3
  9F      -- Start indefinite-length array
    04    -- 4
    05    -- 5
    FF    -- "break" (inner array)
  FF      -- "break" (outer array)
```

0x9f01820203820405ff

```
9F      -- Start indefinite-length array
  01      -- 1
  82      -- Array of length 2
    02    -- 2
    03    -- 3
  82      -- Array of length 2
    04    -- 4
    05    -- 5
  FF      -- "break"
```



```

0x83018202039f0405ff
83      -- Array of length 3
  01      -- 1
  82      -- Array of length 2
    02 -- 2
    03 -- 3
  9F      -- Start indefinite-length array
    04 -- 4
    05 -- 5
    FF -- "break"

```

```

0x83019f0203ff820405
83      -- Array of length 3
  01      -- 1
  9F      -- Start indefinite-length array
    02 -- 2
    03 -- 3
    FF -- "break"
  82      -- Array of length 2
    04 -- 4
    05 -- 5

```

An example of an indefinite-length map (that happens to have two key/value pairs) might be:

```

0xbf6346756ef563416d7421ff
BF      -- Start indefinite-length map
  63      -- First key, UTF-8 string length 3
    46756e -- "Fun"
  F5      -- First value, true
  63      -- Second key, UTF-8 string length 3
    416d74 -- "Amt"
  21      -- Second value, -2
  FF      -- "break"

```

3.2.3. Indefinite-Length Byte Strings and Text Strings

Indefinite-length strings are represented by a byte containing the major type and additional information value of 31, followed by a series of byte or text strings ("chunks") that have definite lengths, followed by the "break" stop code ([Section 3.2.1](#)). The data item represented by the indefinite-length string is the concatenation of the chunks.

If any item between the indefinite-length string indicator (0b010_11111 or 0b011_11111) and the "break" stop code is not a definite-length string item of the same major type, the string is not well-formed.

If any definite-length text string inside an indefinite-length text string is invalid, the indefinite-length text string is invalid. Note that this implies that the bytes of a single UTF-8 character cannot be spread between chunks: a new chunk can only be started at a character boundary.

For example, assume the sequence:

```
0b010_11111 0b010_00100 0xaabbccdd 0b010_00011 0xeeff99 0b111_11111
```

```
5F          -- Start indefinite-length byte string
  44        -- Byte string of length 4
    aabbccdd -- Bytes content
  43        -- Byte string of length 3
    eeff99   -- Bytes content
  FF        -- "break"
```

After decoding, this results in a single byte string with seven bytes: 0xaabbccddeeff99.

3.3. Floating-Point Numbers and Values with No Content

Major type 7 is for two types of data: floating-point numbers and "simple values" that do not need any content. Each value of the 5-bit additional information in the initial byte has its own separate meaning, as defined in Table 1. Like the major types for integers, items of this major type do not carry content data; all the information is in the initial bytes.

5-Bit Value	Semantics
0..23	Simple value (value 0..23)
24	Simple value (value 32..255 in following byte)
25	IEEE 754 Half-Precision Float (16 bits follow)
26	IEEE 754 Single-Precision Float (32 bits follow)
27	IEEE 754 Double-Precision Float (64 bits follow)
28-30	Unassigned, not well-formed in the present document
31	"break" stop code for indefinite-length items (Section 3.2.1)

Table 1: Values for Additional Information in Major Type 7

As with all other major types, the 5-bit value 24 signifies a single-byte extension: it is followed by an additional byte to represent the simple value. (To minimize confusion, only the values 32 to 255 are used.) This maintains the structure of the initial bytes: as for the other major types, the length of these always depends on the additional information in the first byte. Table 2 lists the values assigned and available for simple types.

Value	Semantics
0..19	(Unassigned)
20	False
21	True
22	Null
23	Undefined value
24..31	(Reserved)
32..255	(Unassigned)

Table 2: Simple Values

An encoder **MUST NOT** issue two-byte sequences that start with 0xf8 (major type = 7, additional information = 24) and continue with a byte less than 0x20 (32 decimal). Such sequences are not well-formed. (This implies that an encoder cannot encode false, true, null, or undefined in two-byte sequences, only the one-byte variants of these are well-formed.)

The 5-bit values of 25, 26, and 27 are for 16-bit, 32-bit, and 64-bit IEEE 754 binary floating-point values [IEEE754]. These floating-point values are encoded in the additional bytes of the appropriate size. (See [Appendix D](#) for some information about 16-bit floating point.)

3.4. Optional Tagging of Items

In CBOR, a data item can optionally be preceded by a tag to give it additional semantics while retaining its structure. The tag is major type 6, and represents an unsigned integer as indicated by the tag's argument ([Section 3](#)); the (sole) data item is carried as content data. If a tag requires structured data, this structure is encoded into the nested data item. The definition of a tag usually restricts what kinds of nested data item or items are valid for this tag.

For example, assume that a byte string of length 12 is marked with a tag to indicate it is a positive bignum ([Section 3.4.4](#)). This would be marked as 0b110_00010 (major type 6, additional information 2 for the tag) followed by 0b010_01100 (major type 2, additional

information of 12 for the length) followed by the 12 bytes of the bignum.

Decoders do not need to understand tags, and thus tags may be of little value in applications where the implementation creating a particular CBOR data item and the implementation decoding that stream know the semantic meaning of each item in the data flow. Their primary purpose in this specification is to define common data types such as dates. A secondary purpose is to allow optional tagging when the decoder is a generic CBOR decoder that might be able to benefit from hints about the content of items. Understanding the semantic tags is optional for a decoder; it can just jump over the initial bytes of the tag and interpret the tagged data item itself.

A tag always applies to the item that directly follows it. Thus, if tag A is followed by tag B, which is followed by data item C, tag A applies to the result of applying tag B on data item C. That is, a tagged item is a data item consisting of a tag and a value. The content of the tagged item is the data item (the value) that is being tagged.

IANA maintains a registry of tag values as described in [Section 9.2](#). Table 3 provides a list of values that were defined in [\[RFC7049\]](#), with definitions in the rest of this section. Note that many other tags have been defined since the publication of [\[RFC7049\]](#); see the registry described at [Section 9.2](#) for the complete list.

Tag	Data Item	Semantics
0	UTF-8 string	Standard date/time string; see Section 3.4.2
1	multiple	Epoch-based date/time; see Section 3.4.3
2	byte string	Positive bignum; see Section 3.4.4
3	byte string	Negative bignum; see Section 3.4.4
4	array	Decimal fraction; see Section 3.4.5
5	array	Bigfloat; see Section 3.4.5
21	multiple	Expected conversion to base64url encoding; see Section 3.4.6.2

22	multiple	Expected conversion to base64 encoding; see Section 3.4.6.2
23	multiple	Expected conversion to base16 encoding; see Section 3.4.6.2
24	byte string	Encoded CBOR data item; see Section 3.4.6.1
32	UTF-8 string	URI; see Section 3.4.6.3
33	UTF-8 string	base64url; see Section 3.4.6.3
34	UTF-8 string	base64; see Section 3.4.6.3
35	UTF-8 string	Regular expression; see Section 3.4.6.3
36	UTF-8 string	MIME message; see Section 3.4.6.3
55799	multiple	Self-described CBOR; see Section 3.4.7

Table 3: Values for Tags

[3.4.1.](#) Date and Time

Protocols using tag values 0 and 1 extend the generic data model ([Section 2](#)) with data items representing points in time.

[3.4.2.](#) Standard Date/Time String

Tag value 0 contains a text string in the standard format described by the "date-time" production in [[RFC3339](#)], as refined by [Section 3.3 of \[RFC4287\]](#), representing the point in time described there. A nested item of another type or that doesn't match the [[RFC4287](#)] format is invalid.

[3.4.3.](#) Epoch-based Date/Time

Tag value 1 contains a numerical value counting the number of seconds from 1970-01-01T00:00Z in UTC time to the represented point in civil time.

The tagged item **MUST** be an unsigned or negative integer (major types 0 and 1), or a floating-point number (major type 7 with additional information 25, 26, or 27). Other contained types are invalid.

Non-negative values (major type 0 and non-negative floating-point numbers) stand for time values on or after 1970-01-01T00:00Z UTC and are interpreted according to POSIX [[TIME T](#)]. (POSIX time is also known as UNIX Epoch time. Note that leap seconds are handled specially by POSIX time and this results in a 1 second discontinuity several times per decade.) Note that applications that require the expression of times beyond early 2106 cannot leave out support of 64-bit integers for the tagged value.

Negative values (major type 1 and negative floating-point numbers) are interpreted as determined by the application requirements as there is no universal standard for UTC count-of-seconds time before 1970-01-01T00:00Z (this is particularly true for points in time that precede discontinuities in national calendars). The same applies to non-finite values.

To indicate fractional seconds, floating point values can be used within Tag 1 instead of integer values. Note that this generally requires binary64 support, as binary16 and binary32 provide non-zero fractions of seconds only for a short period of time around early 1970. An application that requires Tag 1 support may restrict the tagged value to be an integer (or a floating-point value) only.

[3.4.4.](#) Bignums

Protocols using tag values 2 and 3 extend the generic data model ([Section 2](#)) with "bignums" representing arbitrarily sized integers. In the generic data model, bignum values are not equal to integers from the basic data model, but specific data models can define that equivalence, and preferred encoding never makes use of bignums that also can be expressed as basic integers (see below).

Bignums are encoded as a byte string data item, which is interpreted as an unsigned integer n in network byte order. Contained items of other types are invalid. For tag value 2, the value of the bignum is n . For tag value 3, the value of the bignum is $-1 - n$. The preferred encoding of the byte string is to leave out any leading zeroes (note that this means the preferred encoding for $n = 0$ is the empty byte string, but see below). Decoders that understand these tags **MUST** be able to decode bignums that do have leading zeroes. The preferred encoding of an integer that can be represented using major type 0 or 1 is to encode it this way instead of as a bignum (which means that the empty string never occurs in a bignum when using preferred encoding). Note that this means the non-preferred choice

of a bignum representation instead of a basic integer for encoding a number is not intended to have application semantics (just as the choice of a longer basic integer representation than needed, such as 0x1800 for 0x00 does not).

For example, the number 18446744073709551616 (2^{64}) is represented as 0b110_00010 (major type 6, tag 2), followed by 0b010_01001 (major type 2, length 9), followed by 0x010000000000000000 (one byte 0x01 and eight bytes 0x00). In hexadecimal:

```
C2                -- Tag 2
  49              -- Byte string of length 9
    010000000000000000 -- Bytes content
```

3.4.5. Decimal Fractions and Bigfloats

Protocols using tag value 4 extend the generic data model with data items representing arbitrary-length decimal fractions $m \cdot (10^e)$. Protocols using tag value 5 extend the generic data model with data items representing arbitrary-length binary fractions $m \cdot (2^e)$. As with bignums, values of different types are not equal in the generic data model.

Decimal fractions combine an integer mantissa with a base-10 scaling factor. They are most useful if an application needs the exact representation of a decimal fraction such as 1.1 because there is no exact representation for many decimal fractions in binary floating point.

Bigfloats combine an integer mantissa with a base-2 scaling factor. They are binary floating-point values that can exceed the range or the precision of the three IEEE 754 formats supported by CBOR ([Section 3.3](#)). Bigfloats may also be used by constrained applications that need some basic binary floating-point capability without the need for supporting IEEE 754.

A decimal fraction or a bigfloat is represented as a tagged array that contains exactly two integer numbers: an exponent e and a mantissa m . Decimal fractions (tag 4) use base-10 exponents; the value of a decimal fraction data item is $m \cdot (10^e)$. Bigfloats (tag 5) use base-2 exponents; the value of a bigfloat data item is $m \cdot (2^e)$. The exponent e MUST be represented in an integer of major type 0 or 1, while the mantissa also can be a bignum ([Section 3.4.4](#)). Contained items with other structures are invalid.

An example of a decimal fraction is that the number 273.15 could be represented as 0b110_00100 (major type of 6 for the tag, additional information of 4 for the type of tag), followed by 0b100_00010 (major

type of 4 for the array, additional information of 2 for the length of the array), followed by 0b001_00001 (major type of 1 for the first integer, additional information of 1 for the value of -2), followed by 0b000_11001 (major type of 0 for the second integer, additional information of 25 for a two-byte value), followed by 0b0110101010110011 (27315 in two bytes). In hexadecimal:

```
C4          -- Tag 4
  82        -- Array of length 2
    21      -- -2
    19 6ab3 -- 27315
```

An example of a bigfloat is that the number 1.5 could be represented as 0b110_00101 (major type of 6 for the tag, additional information of 5 for the type of tag), followed by 0b100_00010 (major type of 4 for the array, additional information of 2 for the length of the array), followed by 0b001_00000 (major type of 1 for the first integer, additional information of 0 for the value of -1), followed by 0b000_00011 (major type of 0 for the second integer, additional information of 3 for the value of 3). In hexadecimal:

```
C5          -- Tag 5
  82        -- Array of length 2
    20      -- -1
    03      -- 3
```

Decimal fractions and bigfloats provide no representation of Infinity, -Infinity, or NaN; if these are needed in place of a decimal fraction or bigfloat, the IEEE 754 half-precision representations from [Section 3.3](#) can be used. For constrained applications, where there is a choice between representing a specific number as an integer and as a decimal fraction or bigfloat (such as when the exponent is small and non-negative), there is a quality-of-implementation expectation that the integer representation is used directly.

[3.4.6. Content Hints](#)

The tags in this section are for content hints that might be used by generic CBOR processors. These content hints do not extend the generic data model.

[3.4.6.1. Encoded CBOR Data Item](#)

Sometimes it is beneficial to carry an embedded CBOR data item that is not meant to be decoded immediately at the time the enclosing data item is being decoded. Tag 24 (CBOR data item) can be used to tag the embedded byte string as a data item encoded in CBOR format.

Contained items that aren't byte strings are invalid. Any contained byte string is valid, even if it encodes an invalid or ill-formed CBOR item.

3.4.6.2. Expected Later Encoding for CBOR-to-JSON Converters

Tags 21 to 23 indicate that a byte string might require a specific encoding when interoperating with a text-based representation. These tags are useful when an encoder knows that the byte string data it is writing is likely to be later converted to a particular JSON-based usage. That usage specifies that some strings are encoded as base64, base64url, and so on. The encoder uses byte strings instead of doing the encoding itself to reduce the message size, to reduce the code size of the encoder, or both. The encoder does not know whether or not the converter will be generic, and therefore wants to say what it believes is the proper way to convert binary strings to JSON.

The data item tagged can be a byte string or any other data item. In the latter case, the tag applies to all of the byte string data items contained in the data item, except for those contained in a nested data item tagged with an expected conversion.

These three tag types suggest conversions to three of the base data encodings defined in [[RFC4648](#)]. For base64url encoding (tag 21), padding is not used (see [Section 3.2 of RFC 4648](#)); that is, all trailing equals signs ("=") are removed from the encoded string. For base64 encoding (tag 22), padding is used as defined in [RFC 4648](#). For both base64url and base64, padding bits are set to zero (see [Section 3.5 of RFC 4648](#)), and encoding is performed without the inclusion of any line breaks, whitespace, or other additional characters. Note that, for all three tags, the encoding of the empty byte string is the empty text string.

3.4.6.3. Encoded Text

Some text strings hold data that have formats widely used on the Internet, and sometimes those formats can be validated and presented to the application in appropriate form by the decoder. There are tags for some of these formats. As with tags 21 to 23, if these tags are applied to an item other than a text string, they apply to all text string data items it contains.

- o Tag 32 is for URIs, as defined in [[RFC3986](#)]. If the text string doesn't match the "URI-reference" production, the string is invalid.
- o Tags 33 and 34 are for base64url- and base64-encoded text strings, as defined in [[RFC4648](#)]. If any of:

- * the encoded text string contains non-alphabet characters or only 1 character in the last block of 4, or
- * the padding bits in a 2- or 3-character block are not 0, or
- * the base64 encoding has the wrong number of padding characters, or
- * the base64url encoding has padding characters,

the string is invalid.

- o Tag 35 is for regular expressions that are roughly in Perl Compatible Regular Expressions (PCRE/PCRE2) form [[PCRE](#)] or a version of the JavaScript regular expression syntax [[ECMA262](#)]. (Note that more specific identification may be necessary if the actual version of the specification underlying the regular expression, or more than just the text of the regular expression itself, need to be conveyed.) Any contained string value is valid.
- o Tag 36 is for MIME messages (including all headers), as defined in [[RFC2045](#)]. A text string that isn't a valid MIME message is invalid.

Note that tags 33 and 34 differ from 21 and 22 in that the data is transported in base-encoded form for the former and in raw byte string form for the latter.

[3.4.7.](#) Self-Described CBOR

In many applications, it will be clear from the context that CBOR is being employed for encoding a data item. For instance, a specific protocol might specify the use of CBOR, or a media type is indicated that specifies its use. However, there may be applications where such context information is not available, such as when CBOR data is stored in a file that does not have disambiguating metadata. Here, it may help to have some distinguishing characteristics for the data itself.

Tag 55799 is defined for this purpose. It does not impart any special semantics on the data item that follows; that is, the semantics of a data item tagged with tag 55799 is exactly identical to the semantics of the data item itself.

The serialization of this tag is 0xd9d9f7, which does not appear to be in use as a distinguishing mark for any frequently used file types. In particular, 0xd9d9f7 is not a valid start of a Unicode

text in any Unicode encoding if it is followed by a valid CBOR data item.

For instance, a decoder might be able to decode both CBOR and JSON. Such a decoder would need to mechanically distinguish the two formats. An easy way for an encoder to help the decoder would be to tag the entire CBOR item with tag 55799, the serialization of which will never be found at the beginning of a JSON text.

4. Serialization Considerations

4.1. Preferred Serialization

For some values at the data model level, CBOR provides multiple serializations. For many applications, it is desirable that an encoder always chooses a preferred serialization; however, the present specification does not put the burden of enforcing this preference on either encoder or decoder.

Some constrained decoders may be limited in their ability to decode non-preferred serializations: For example, if only integers below 1_000_000_000 are expected in an application, the decoder may leave out the code that would be needed to decode 64-bit arguments in integers. An encoder that always uses preferred serialization ("preferred encoder") interoperates with this decoder for the numbers that can occur in this application. More generally speaking, it therefore can be said that a preferred encoder is more universally interoperable (and also less wasteful) than one that, say, always uses 64-bit integers.

Similarly, a constrained encoder may be limited in the variety of representation variants it supports in such a way that it does not emit preferred serializations ("variant encoder"): Say, it could be designed to always use the 32-bit variant for an integer that it encodes even if a short representation is available (again, assuming that there is no application need for integers that can only be represented with the 64-bit variant). A decoder that does not rely on only ever receiving preferred serializations ("variation-tolerant decoder") can then be said to be more universally interoperable (it might very well optimize for the case of receiving preferred serializations, though). Full implementations of CBOR decoders are by definition variation-tolerant; the distinction is only relevant if a constrained implementation of a CBOR decoder meets a variant encoder.

The preferred serialization always uses the shortest form of representing the argument ([Section 3](#)); it also uses the shortest floating point encoding that preserves the value being encoded (see

[Section 5.5](#)). Definite length encoding is preferred whenever the length is known at the time the serialization of the item starts.

4.2. Deterministically Encoded CBOR

Some protocols may want encoders to only emit CBOR in a particular deterministic format; those protocols might also have the decoders check that their input is in that deterministic format. Those protocols are free to define what they mean by a "deterministic format" and what encoders and decoders are expected to do. This section defines a set of restrictions that can serve as the base of such a deterministic format.

4.2.1. Core Deterministic Encoding Requirements

A CBOR encoding satisfies the "core deterministic encoding requirements" if it satisfies the following restrictions:

- o Arguments (see [Section 3](#)) for integers, lengths in major types 2 through 5, and tags MUST be as short as possible. In particular:
 - * 0 to 23 and -1 to -24 MUST be expressed in the same byte as the major type;
 - * 24 to 255 and -25 to -256 MUST be expressed only with an additional uint8_t;
 - * 256 to 65535 and -257 to -65536 MUST be expressed only with an additional uint16_t;
 - * 65536 to 4294967295 and -65537 to -4294967296 MUST be expressed only with an additional uint32_t.
- o The keys in every map MUST be sorted in the bitwise lexicographic order of their deterministic encodings. For example, the following keys are sorted correctly:
 1. 10, encoded as 0x0a.
 2. 100, encoded as 0x1864.
 3. -1, encoded as 0x20.
 4. "z", encoded as 0x617a.
 5. "aa", encoded as 0x626161.
 6. [100], encoded as 0x811864.

7. [-1], encoded as 0x8120.
 8. false, encoded as 0xf4.
- o Indefinite-length items MUST NOT appear. They can be encoded as definite-length items instead.

4.2.2. Additional Deterministic Encoding Considerations

If a protocol allows for IEEE floats, then additional deterministic encoding rules might need to be added. One example rule might be to have all floats start as a 64-bit float, then do a test conversion to a 32-bit float; if the result is the same numeric value, use the shorter value and repeat the process with a test conversion to a 16-bit float. (This rule selects 16-bit float for positive and negative Infinity as well.) Also, there are many representations for NaN. If NaN is an allowed value, it must always be represented as 0xf97e00.

CBOR tags present additional considerations for deterministic encoding. The absence or presence of tags in a deterministic format is determined by the optionality of the tags in the protocol. In a CBOR-based protocol that allows optional tagging anywhere, the deterministic format must not allow them. In a protocol that requires tags in certain places, the tag needs to appear in the deterministic format. A CBOR-based protocol that uses deterministic encoding might instead say that all tags that appear in a message must be retained regardless of whether they are optional.

Protocols that include floating, big integer, or other complex values need to define extra requirements on their deterministic encodings. For example:

- o If a protocol includes a field that can express floating values ([Section 3.3](#)), the protocol's deterministic encoding needs to specify whether the integer 1.0 is encoded as 0x01, 0xf93c00, 0xfa3f800000, or 0xfb3ff0000000000000. Three sensible rules for this are:
 1. Encode integral values that fit in 64 bits as values from major types 0 and 1, and other values as the smallest of 16-, 32-, or 64-bit floating point that accurately represents the value,
 2. Encode all values as the smallest of 16-, 32-, or 64-bit floating point that accurately represents the value, even for integral values, or

3. Encode all values as 64-bit floating point.

If NaN is an allowed value, the protocol needs to pick a single representation, for example 0xf97e00.

- o If a protocol includes a field that can express integers larger than 2^{64} using tag 2 ([Section 3.4.4](#)), the protocol's deterministic encoding needs to specify whether small integers are expressed using the tag or major types 0 and 1.
- o A protocol might give encoders the choice of representing a URL as either a text string or, using [Section 3.4.6.3](#), tag 32 containing a text string. This protocol's deterministic encoding needs to either require that the tag is present or require that it's absent, not allow either one.

[4.2.3](#). Length-first map key ordering

The core deterministic encoding requirements sort map keys in a different order from the one suggested by [Section 3.9 of \[RFC7049\]](#) (called "Canonical CBOR" there). Protocols that need to be compatible with [\[RFC7049\]](#)'s order can instead be specified in terms of this specification's "length-first core deterministic encoding requirements":

A CBOR encoding satisfies the "length-first core deterministic encoding requirements" if it satisfies the core deterministic encoding requirements except that the keys in every map MUST be sorted such that:

1. If two keys have different lengths, the shorter one sorts earlier;
2. If two keys have the same length, the one with the lower value in (byte-wise) lexical order sorts earlier.

For example, under the length-first core deterministic encoding requirements, the following keys are sorted correctly:

1. 10, encoded as 0x0a.
2. -1, encoded as 0x20.
3. false, encoded as 0xf4.
4. 100, encoded as 0x1864.
5. "z", encoded as 0x617a.

6. [-1], encoded as 0x8120.
7. "aa", encoded as 0x626161.
8. [100], encoded as 0x811864.

(Although [\[RFC7049\]](#) used the term "Canonical CBOR" for its form of requirements on deterministic encoding, this document avoids this term because "canonicalization" is often associated with specific uses of deterministic encoding only. The terms are essentially exchangeable, however, and the set of core requirements in this document could also be called "Canonical CBOR", while the length-first-ordered version of that could be called "Old Canonical CBOR".)

5. Creating CBOR-Based Protocols

Data formats such as CBOR are often used in environments where there is no format negotiation. A specific design goal of CBOR is to not need any included or assumed schema: a decoder can take a CBOR item and decode it with no other knowledge.

Of course, in real-world implementations, the encoder and the decoder will have a shared view of what should be in a CBOR data item. For example, an agreed-to format might be "the item is an array whose first value is a UTF-8 string, second value is an integer, and subsequent values are zero or more floating-point numbers" or "the item is a map that has byte strings for keys and contains at least one pair whose key is 0xab01".

CBOR-based protocols MUST specify how their decoders handle invalid and other unexpected data. CBOR-based protocols MAY specify that they treat arbitrary valid data as unexpected. Encoders for CBOR-based protocols MUST produce only valid items, that is, the protocol cannot be designed to make use of invalid items. An encoder can be capable of encoding as many or as few types of values as is required by the protocol in which it is used; a decoder can be capable of understanding as many or as few types of values as is required by the protocols in which it is used. This lack of restrictions allows CBOR to be used in extremely constrained environments.

This section discusses some considerations in creating CBOR-based protocols. With few exceptions, it is advisory only and explicitly excludes any language from [BCP 14](#) other than words that could be interpreted as "MAY" in the sense of [BCP 14](#). The exceptions aim at facilitating interoperability of CBOR-based protocols while making use of a wide variety of both generic and application-specific encoders and decoders.

[5.1.](#) CBOR in Streaming Applications

In a streaming application, a data stream may be composed of a sequence of CBOR data items concatenated back-to-back. In such an environment, the decoder immediately begins decoding a new data item if data is found after the end of a previous data item.

Not all of the bytes making up a data item may be immediately available to the decoder; some decoders will buffer additional data until a complete data item can be presented to the application. Other decoders can present partial information about a top-level data item to an application, such as the nested data items that could already be decoded, or even parts of a byte string that hasn't completely arrived yet.

Note that some applications and protocols will not want to use indefinite-length encoding. Using indefinite-length encoding allows an encoder to not need to marshal all the data for counting, but it requires a decoder to allocate increasing amounts of memory while waiting for the end of the item. This might be fine for some applications but not others.

[5.2.](#) Generic Encoders and Decoders

A generic CBOR decoder can decode all well-formed CBOR data and present them to an application. See [Appendix C](#).

Even though CBOR attempts to minimize these cases, not all well-formed CBOR data is valid: for example, the encoded text string "0x62c0ae" does not contain valid UTF-8 and so is not a valid CBOR item. Also, specific tags may make semantic constraints that may be violated, such as a bignum tag containing another tag, or an instance of tag 0 containing a byte string or a text string with contents that do not match [\[RFC3339\]](#)'s "date-time" production. There is no requirement that generic encoders and decoders make unnatural choices for their application interface to enable the processing of invalid data. Generic encoders and decoders are expected to forward simple values and tags even if their specific codepoints are not registered at the time the encoder/decoder is written ([Section 5.4](#)).

Generic decoders provide ways to present well-formed CBOR values, both valid and invalid, to an application. The diagnostic notation ([Section 8](#)) may be used to present well-formed CBOR values to humans.

Generic encoders provide an application interface that allows the application to specify any well-formed value, including simple values and tags unknown to the encoder.

5.3. Invalid Items

A well-formed but invalid CBOR data item presents a problem with interpreting the data encoded in it in the CBOR data model. A CBOR-based protocol could be specified in several layers, in which the lower layers don't process the semantics of some of the CBOR data they forward. These layers can't notice the invalidity in data they don't process and MUST forward that data as-is. The first layer that does process the semantics of an invalid CBOR item MUST take one of two choices:

1. Replace the problematic item with an error marker and continue with the next item, or
2. Issue an error and stop processing altogether.

A CBOR-based protocol MUST specify which of these options its decoders take, for each kind of invalid item they might encounter.

Such problems might include:

Duplicate keys in a map: Generic decoders ([Section 5.2](#)) make data available to applications using the native CBOR data model. That data model includes maps (key-value mappings with unique keys), not multimaps (key-value mappings where multiple entries can have the same key). Thus, a generic decoder that gets a CBOR map item that has duplicate keys will decode to a map with only one instance of that key, or it might stop processing altogether. On the other hand, a "streaming decoder" may not even be able to notice ([Section 5.6](#)).

Inadmissible type on the value following a tag: Tags ([Section 3.4](#)) specify what type of data item is supposed to follow the tag; for example, the tags for positive or negative bignums are supposed to be put on byte strings. A decoder that decodes the tagged data item into a native representation (a native big integer in this example) is expected to check the type of the data item being tagged. Even decoders that don't have such native representations available in their environment may perform the check on those tags known to them and react appropriately.

Invalid UTF-8 string: A decoder might or might not want to verify that the sequence of bytes in a UTF-8 string (major type 3) is actually valid UTF-8 and react appropriately.

5.4. Handling Unknown Simple Values and Tags

A decoder that comes across a simple value ([Section 3.3](#)) that it does not recognize, such as a value that was added to the IANA registry after the decoder was deployed or a value that the decoder chose not to implement, might issue a warning, might stop processing altogether, might handle the error by making the unknown value available to the application as such (as is expected of generic decoders), or take some other type of action.

A decoder that comes across a tag ([Section 3.4](#)) that it does not recognize, such as a tag that was added to the IANA registry after the decoder was deployed or a tag that the decoder chose not to implement, might issue a warning, might stop processing altogether, might handle the error and present the unknown tag value together with the contained data item to the application (as is expected of generic decoders), might ignore the tag and simply present the contained data item only to the application, or take some other type of action.

5.5. Numbers

CBOR-based protocols should take into account that different language environments pose different restrictions on the range and precision of numbers that are representable. For example, the JavaScript number system treats all numbers as floating point, which may result in silent loss of precision in decoding integers with more than 53 significant bits. A protocol that uses numbers should define its expectations on the handling of non-trivial numbers in decoders and receiving applications.

A CBOR-based protocol that includes floating-point numbers can restrict which of the three formats (half-precision, single-precision, and double-precision) are to be supported. For an integer-only application, a protocol may want to completely exclude the use of floating-point values.

A CBOR-based protocol designed for compactness may want to exclude specific integer encodings that are longer than necessary for the application, such as to save the need to implement 64-bit integers. There is an expectation that encoders will use the most compact integer representation that can represent a given value. However, a compact application should accept values that use a longer-than-needed encoding (such as encoding "0" as 0b000_11001 followed by two bytes of 0x00) as long as the application can decode an integer of the given size.

The preferred encoding for a floating point value is the shortest floating point encoding that preserves its value, e.g., 0xf94580 for the number 5.5, and 0xfa45ad9c00 for the number 5555.5, unless the CBOR-based protocol specifically excludes the use of the shorter floating point encodings. For NaN values, a shorter encoding is preferred if zero-padding the shorter significand towards the right reconstitutes the original NaN value (for many applications, the single NaN encoding 0xf97e00 will suffice).

5.6. Specifying Keys for Maps

The encoding and decoding applications need to agree on what types of keys are going to be used in maps. In applications that need to interwork with JSON-based applications, keys probably should be limited to UTF-8 strings only; otherwise, there has to be a specified mapping from the other CBOR types to Unicode characters, and this often leads to implementation errors. In applications where keys are numeric in nature and numeric ordering of keys is important to the application, directly using the numbers for the keys is useful.

If multiple types of keys are to be used, consideration should be given to how these types would be represented in the specific programming environments that are to be used. For example, in JavaScript Maps [[ECMA262](#)], a key of integer 1 cannot be distinguished from a key of floating point 1.0. This means that, if integer keys are used, the protocol needs to avoid use of floating-point keys the values of which happen to be integer numbers in the same map.

Decoders that deliver data items nested within a CBOR data item immediately on decoding them ("streaming decoders") often do not keep the state that is necessary to ascertain uniqueness of a key in a map. Similarly, an encoder that can start encoding data items before the enclosing data item is completely available ("streaming encoder") may want to reduce its overhead significantly by relying on its data source to maintain uniqueness.

A CBOR-based protocol MUST define what to do when a receiving application does see multiple identical keys in a map. The resulting rule in the protocol MUST respect the CBOR data model: it cannot prescribe a specific handling of the entries with the identical keys, except that it might have a rule that having identical keys in a map indicates a malformed map and that the decoder has to stop with an error. Duplicate keys are also prohibited by CBOR decoders that are using strict mode ([Section 5.8](#)).

The CBOR data model for maps does not allow ascribing semantics to the order of the key/value pairs in the map representation. Thus, a CBOR-based protocol MUST NOT specify that changing the key/value pair

order in a map would change the semantics, except to specify that some, orders are disallowed, for example where they would not meet the requirements of a deterministic encoding ([Section 4.2](#)). (Any secondary effects of map ordering such as on timing, cache usage, and other potential side channels are not considered part of the semantics but may be enough reason on its own for a protocol to require a deterministic encoding format.)

Applications for constrained devices that have maps with 24 or fewer frequently used keys should consider using small integers (and those with up to 48 frequently used keys should consider also using small negative integers) because the keys can then be encoded in a single byte.

[5.6.1](#). Equivalence of Keys

The specific data model applying to a CBOR data item is used to determine whether keys occurring in maps are duplicates or distinct.

At the generic data model level, numerically equivalent integer and floating point values are distinct from each other, as they are from the various big numbers (Tags 2 to 5). Similarly, text strings are distinct from byte strings, even if composed of the same bytes. A tagged value is distinct from an untagged value or from a value tagged with a different tag.

Within each of these groups, numeric values are distinct unless they are numerically equal (specifically, -0.0 is equal to 0.0); for the purpose of map key equivalence, NaN (not a number) values are equivalent if they have the same significand after zero-extending both significands at the right to 64 bits.

(Byte and text) strings are compared byte by byte, arrays element by element, and are equal if they have the same number of bytes/elements and the same values at the same positions. Two maps are equal if they have the same set of pairs regardless of their order; pairs are equal if both the key and value are equal.

Tagged values are equal if both the tag and the value are equal. Simple values are equal if they simply have the same value. Nothing else is equal in the generic data model, a simple value 2 is not equivalent to an integer 2 and an array is never equivalent to a map.

As discussed in [Section 2.2](#), specific data models can make values equivalent for the purpose of comparing map keys that are distinct in the generic data model. Note that this implies that a generic decoder may deliver a decoded map to an application that needs to be checked for duplicate map keys by that application (alternatively,

the decoder may provide a programming interface to perform this service for the application). Specific data models cannot distinguish values for map keys that are equal for this purpose at the generic data model level.

5.7. Undefined Values

In some CBOR-based protocols, the simple value ([Section 3.3](#)) of Undefined might be used by an encoder as a substitute for a data item with an encoding problem, in order to allow the rest of the enclosing data items to be encoded without harm.

5.8. Strict Decoding Mode

Some areas of application of CBOR do not require deterministic encoding ([Section 4.2](#)) but may require that different decoders reach the same (semantically equivalent) results, even in the presence of potentially malicious data. This can be required if one application (such as a firewall or other protecting entity) makes a decision based on the data that another application, which independently decodes the data, relies on.

Normally, it is the responsibility of the sender to avoid ambiguously decodable data. However, the sender might be an attacker specially making up CBOR data such that it will be interpreted differently by different decoders in an attempt to exploit that as a vulnerability. Generic decoders used in applications where this might be a problem need to support a strict mode in which it is also the responsibility of the receiver to reject ambiguously decodable data. It is expected that firewalls and other security systems that decode CBOR will only decode in strict mode.

A decoder in strict mode will reliably reject any data that could be interpreted by other decoders in different ways. It will expend the effort to reliably detect invalid data items ([Section 5.3](#)). For example, a strict decoder needs to have an API that reports an error (and does not return data) for a CBOR data item that contains any of the following:

- o a map (major type 5) that has more than one entry with the same key
- o a tag that is used on a data item of the incorrect type
- o a data item that is incorrectly formatted for the type given to it, such as invalid UTF-8 or data that cannot be interpreted with the specific tag that it has been tagged with

A decoder in strict mode can do one of two things when it encounters a tag or simple value that it does not recognize:

- o It can report an error (and not return data).
- o It can emit the unknown item (type, value, and, for tags, the decoded tagged data item) to the application calling the decoder with an indication that the decoder did not recognize that tag or simple value.

The latter approach, which is also appropriate for non-strict decoders, supports forward compatibility with newly registered tags and simple values without the requirement to update the encoder at the same time as the calling application. (For this, the API for the decoder needs to have a way to mark unknown items so that the calling application can handle them in a manner appropriate for the program.)

Since some of this processing may have an appreciable cost (in particular with duplicate detection for maps), support of strict mode is not a requirement placed on all CBOR decoders.

Some encoders will rely on their applications to provide input data in such a way that unambiguously decodable CBOR results. A generic encoder also may want to provide a strict mode where it reliably limits its output to unambiguously decodable CBOR, independent of whether or not its application is providing API-conformant data.

6. Converting Data between CBOR and JSON

This section gives non-normative advice about converting between CBOR and JSON. Implementations of converters are free to use whichever advice here they want.

It is worth noting that a JSON text is a sequence of characters, not an encoded sequence of bytes, while a CBOR data item consists of bytes, not characters.

6.1. Converting from CBOR to JSON

Most of the types in CBOR have direct analogs in JSON. However, some do not, and someone implementing a CBOR-to-JSON converter has to consider what to do in those cases. The following non-normative advice deals with these by converting them to a single substitute value, such as a JSON null.

- o An integer (major type 0 or 1) becomes a JSON number.

- o A byte string (major type 2) that is not embedded in a tag that specifies a proposed encoding is encoded in base64url without padding and becomes a JSON string.
- o A UTF-8 string (major type 3) becomes a JSON string. Note that JSON requires escaping certain characters ([\[RFC8259\]](#), [Section 7](#)): quotation mark (U+0022), reverse solidus (U+005C), and the "C0 control characters" (U+0000 through U+001F). All other characters are copied unchanged into the JSON UTF-8 string.
- o An array (major type 4) becomes a JSON array.
- o A map (major type 5) becomes a JSON object. This is possible directly only if all keys are UTF-8 strings. A converter might also convert other keys into UTF-8 strings (such as by converting integers into strings containing their decimal representation); however, doing so introduces a danger of key collision. Note also that, if tags on UTF-8 strings are ignored as proposed below, this will cause a key collision if the tags are different but the strings are the same.
- o False (major type 7, additional information 20) becomes a JSON false.
- o True (major type 7, additional information 21) becomes a JSON true.
- o Null (major type 7, additional information 22) becomes a JSON null.
- o A floating-point value (major type 7, additional information 25 through 27) becomes a JSON number if it is finite (that is, it can be represented in a JSON number); if the value is non-finite (NaN, or positive or negative Infinity), it is represented by the substitute value.
- o Any other simple value (major type 7, any additional information value not yet discussed) is represented by the substitute value.
- o A bignum (major type 6, tag value 2 or 3) is represented by encoding its byte string in base64url without padding and becomes a JSON string. For tag value 3 (negative bignum), a "~" (ASCII tilde) is inserted before the base-encoded value. (The conversion to a binary blob instead of a number is to prevent a likely numeric overflow for the JSON decoder.)
- o A byte string with an encoding hint (major type 6, tag value 21 through 23) is encoded as described and becomes a JSON string.

- o For all other tags (major type 6, any other tag value), the embedded CBOR item is represented as a JSON value; the tag value is ignored.
- o Indefinite-length items are made definite before conversion.

6.2. Converting from JSON to CBOR

All JSON values, once decoded, directly map into one or more CBOR values. As with any kind of CBOR generation, decisions have to be made with respect to number representation. In a suggested conversion:

- o JSON numbers without fractional parts (integer numbers) are represented as integers (major types 0 and 1, possibly major type 6 tag value 2 and 3), choosing the shortest form; integers longer than an implementation-defined threshold (which is usually either 32 or 64 bits) may instead be represented as floating-point values. (If the JSON was generated from a JavaScript implementation, its precision is already limited to 53 bits maximum.)
- o Numbers with fractional parts are represented as floating-point values. Preferably, the shortest exact floating-point representation is used; for instance, 1.5 is represented in a 16-bit floating-point value (not all implementations will be capable of efficiently finding the minimum form, though). There may be an implementation-defined limit to the precision that will affect the precision of the represented values. Decimal representation should only be used if that is specified in a protocol.

CBOR has been designed to generally provide a more compact encoding than JSON. One implementation strategy that might come to mind is to perform a JSON-to-CBOR encoding in place in a single buffer. This strategy would need to carefully consider a number of pathological cases, such as that some strings represented with no or very few escapes and longer (or much longer) than 255 bytes may expand when encoded as UTF-8 strings in CBOR. Similarly, a few of the binary floating-point representations might cause expansion from some short decimal representations (1.1, 1e9) in JSON. This may be hard to get right, and any ensuing vulnerabilities may be exploited by an attacker.

7. Future Evolution of CBOR

Successful protocols evolve over time. New ideas appear, implementation platforms improve, related protocols are developed and evolve, and new requirements from applications and protocols are added. Facilitating protocol evolution is therefore an important design consideration for any protocol development.

For protocols that will use CBOR, CBOR provides some useful mechanisms to facilitate their evolution. Best practices for this are well known, particularly from JSON format development of JSON-based protocols. Therefore, such best practices are outside the scope of this specification.

However, facilitating the evolution of CBOR itself is very well within its scope. CBOR is designed to both provide a stable basis for development of CBOR-based protocols and to be able to evolve. Since a successful protocol may live for decades, CBOR needs to be designed for decades of use and evolution. This section provides some guidance for the evolution of CBOR. It is necessarily more subjective than other parts of this document. It is also necessarily incomplete, lest it turn into a textbook on protocol development.

7.1. Extension Points

In a protocol design, opportunities for evolution are often included in the form of extension points. For example, there may be a codepoint space that is not fully allocated from the outset, and the protocol is designed to tolerate and embrace implementations that start using more codepoints than initially allocated.

Sizing the codepoint space may be difficult because the range required may be hard to predict. An attempt should be made to make the codepoint space large enough so that it can slowly be filled over the intended lifetime of the protocol.

CBOR has three major extension points:

- o the "simple" space (values in major type 7). Of the 24 efficient (and 224 slightly less efficient) values, only a small number have been allocated. Implementations receiving an unknown simple data item may be able to process it as such, given that the structure of the value is indeed simple. The IANA registry in [Section 9.1](#) is the appropriate way to address the extensibility of this codepoint space.
- o the "tag" space (values in major type 6). Again, only a small part of the codepoint space has been allocated, and the space is

abundant (although the early numbers are more efficient than the later ones). Implementations receiving an unknown tag can choose to simply ignore it or to process it as an unknown tag wrapping the following data item. The IANA registry in [Section 9.2](#) is the appropriate way to address the extensibility of this codepoint space.

- o the "additional information" space. An implementation receiving an unknown additional information value has no way to continue decoding, so allocating codepoints to this space is a major step. There are also very few codepoints left.

[7.2.](#) Curating the Additional Information Space

The human mind is sometimes drawn to filling in little perceived gaps to make something neat. We expect the remaining gaps in the codepoint space for the additional information values to be an attractor for new ideas, just because they are there.

The present specification does not manage the additional information codepoint space by an IANA registry. Instead, allocations out of this space can only be done by updating this specification.

For an additional information value of $n \geq 24$, the size of the additional data typically is $2^{(n-24)}$ bytes. Therefore, additional information values 28 and 29 should be viewed as candidates for 128-bit and 256-bit quantities, in case a need arises to add them to the protocol. Additional information value 30 is then the only additional information value available for general allocation, and there should be a very good reason for allocating it before assigning it through an update of this protocol.

[8.](#) Diagnostic Notation

CBOR is a binary interchange format. To facilitate documentation and debugging, and in particular to facilitate communication between entities cooperating in debugging, this section defines a simple human-readable diagnostic notation. All actual interchange always happens in the binary format.

Note that this truly is a diagnostic format; it is not meant to be parsed. Therefore, no formal definition (as in ABNF) is given in this document. (Implementers looking for a text-based format for representing CBOR data items in configuration files may also want to consider YAML [[YAML](#)].)

The diagnostic notation is loosely based on JSON as it is defined in [RFC 8259](#), extending it where needed.

The notation borrows the JSON syntax for numbers (integer and floating point), True (`>true<`), False (`>false<`), Null (`>null<`), UTF-8 strings, arrays, and maps (maps are called objects in JSON; the diagnostic notation extends JSON here by allowing any data item in the key position). Undefined is written `>undefined<` as in JavaScript. The non-finite floating-point numbers Infinity, -Infinity, and NaN are written exactly as in this sentence (this is also a way they can be written in JavaScript, although JSON does not allow them). A tagged item is written as an integer number for the tag followed by the item in parentheses; for instance, an [RFC 3339](#) (ISO 8601) date could be notated as:

```
0("2013-03-21T20:04:00Z")
```

or the equivalent relative time as

```
1(1363896240)
```

Byte strings are notated in one of the base encodings, without padding, enclosed in single quotes, prefixed by `>h<` for base16, `>b32<` for base32, `>h32<` for base32hex, `>b64<` for base64 or base64url (the actual encodings do not overlap, so the string remains unambiguous). For example, the byte string `0x12345678` could be written `h'12345678'`, `b32'CI2FM6A'`, or `b64'EjRWeA'`.

Unassigned simple values are given as `"simple()"` with the appropriate integer in the parentheses. For example, `"simple(42)"` indicates major type 7, value 42.

[8.1.](#) Encoding Indicators

Sometimes it is useful to indicate in the diagnostic notation which of several alternative representations were actually used; for example, a data item written `>1.5<` by a diagnostic decoder might have been encoded as a half-, single-, or double-precision float.

The convention for encoding indicators is that anything starting with an underscore and all following characters that are alphanumeric or underscore, is an encoding indicator, and can be ignored by anyone not interested in this information. Encoding indicators are always optional.

A single underscore can be written after the opening brace of a map or the opening bracket of an array to indicate that the data item was represented in indefinite-length format. For example, `[_ 1, 2]` contains an indicator that an indefinite-length representation was used to represent the data item `[1, 2]`.

An underscore followed by a decimal digit *n* indicates that the preceding item (or, for arrays and maps, the item starting with the preceding bracket or brace) was encoded with an additional information value of $24+n$. For example, `1.5_1` is a half-precision floating-point number, while `1.5_3` is encoded as double precision. This encoding indicator is not shown in [Appendix A](#). (Note that the encoding indicator `"_"` is thus an abbreviation of the full form `"_7"`, which is not used.)

As a special case, byte and text strings of indefinite length can be notated in the form `(_ h'0123', h'4567')` and `(_ "foo", "bar")`.

9. IANA Considerations

IANA has created two registries for new CBOR values. The registries are separate, that is, not under an umbrella registry, and follow the rules in [\[RFC8126\]](#). IANA has also assigned a new MIME media type and an associated Constrained Application Protocol (CoAP) Content-Format entry.

9.1. Simple Values Registry

IANA has created the "Concise Binary Object Representation (CBOR) Simple Values" registry at [\[IANA.cbor-simple-values\]](#). The initial values are shown in Table 2.

New entries in the range 0 to 19 are assigned by Standards Action. It is suggested that these Standards Actions allocate values starting with the number 16 in order to reserve the lower numbers for contiguous blocks (if any).

New entries in the range 32 to 255 are assigned by Specification Required.

9.2. Tags Registry

IANA has created the "Concise Binary Object Representation (CBOR) Tags" registry at [\[IANA.cbor-tags\]](#). The tags that were defined in [\[RFC7049\]](#) are described in detail in [Section 3.4](#), but other tags have already been defined.

New entries in the range 0 to 23 are assigned by Standards Action. New entries in the range 24 to 255 are assigned by Specification Required. New entries in the range 256 to 18446744073709551615 are assigned by First Come First Served. The template for registration requests is:

- o Data item

- o Semantics (short form)

In addition, First Come First Served requests should include:

- o Point of contact
- o Description of semantics (URL) - This description is optional; the URL can point to something like an Internet-Draft or a web page.

9.3. Media Type ("MIME Type")

The Internet media type [[RFC6838](#)] for a single encoded CBOR data item is application/cbor.

Type name: application

Subtype name: cbor

Required parameters: n/a

Optional parameters: n/a

Encoding considerations: binary

Security considerations: See [Section 10](#) of this document

Interoperability considerations: n/a

Published specification: This document

Applications that use this media type: None yet, but it is expected that this format will be deployed in protocols and applications.

Additional information:

Magic number(s): n/a

File extension(s): .cbor

Macintosh file type code(s): n/a

Person & email address to contact for further information:

Carsten Bormann

cabo@tzi.org

Intended usage: COMMON

Restrictions on usage: none

Author:

Carsten Bormann <cabo@tzi.org>

Change controller:

The IESG <iesg@ietf.org>

9.4. CoAP Content-Format

Media Type: application/cbor

Encoding: -

Id: 60

Reference: [RFCthis]

9.5. The +cbor Structured Syntax Suffix Registration

Name: Concise Binary Object Representation (CBOR)

+suffix: +cbor

References: [RFCthis]

Encoding Considerations: CBOR is a binary format.

Interoperability Considerations: n/a

Fragment Identifier Considerations:

The syntax and semantics of fragment identifiers specified for +cbor SHOULD be as specified for "application/cbor". (At publication of this document, there is no fragment identification syntax defined for "application/cbor".)

The syntax and semantics for fragment identifiers for a specific "xxx/yyy+cbor" SHOULD be processed as follows:

For cases defined in +cbor, where the fragment identifier resolves per the +cbor rules, then process as specified in +cbor.

For cases defined in +cbor, where the fragment identifier does not resolve per the +cbor rules, then process as specified in "xxx/yyy+cbor".

For cases not defined in +cbor, then process as specified in "xxx/yyy+cbor".

Security Considerations: See [Section 10](#) of this document

Contact:

Apps Area Working Group (apps-discuss@ietf.org)

Author/Change Controller:

The Apps Area Working Group.

The IESG has change control over this registration.

[10.](#) Security Considerations

A network-facing application can exhibit vulnerabilities in its processing logic for incoming data. Complex parsers are well known as a likely source of such vulnerabilities, such as the ability to remotely crash a node, or even remotely execute arbitrary code on it. CBOR attempts to narrow the opportunities for introducing such vulnerabilities by reducing parser complexity, by giving the entire range of encodable values a meaning where possible.

Because CBOR decoders are often used as a first step in processing unvalidated input, they need to be fully prepared for all types of hostile input that may be designed to corrupt, overrun, or achieve control of the system decoding the CBOR data item. A CBOR decoder needs to assume that all input may be hostile even if it has been checked by a firewall, has come over a TLS-secured channel, is encrypted or signed, or has come from some other source that is presumed trusted.

Hostile input may be constructed to overrun buffers, overflow or underflow integer arithmetic, or cause other decoding disruption. CBOR data items might have lengths or sizes that are intentionally extremely large or too short. Resource exhaustion attacks might attempt to lure a decoder into allocating very big data items (strings, arrays, maps) or exhaust the stack depth by setting up deeply nested items. Decoders need to have appropriate resource management to mitigate these attacks. (Items for which very large sizes are given can also attempt to exploit integer overflow vulnerabilities.)

A CBOR decoder, by definition, only accepts well-formed CBOR; this is the first step to its robustness. Input that is not well-formed CBOR causes no further processing from the point where the lack of well-formedness was detected. If possible, any data decoded up to this point should have no impact on the application using the CBOR decoder.

In addition to ascertaining well-formedness, a CBOR decoder might also perform validity checks on the CBOR data. Alternatively, it can leave those checks to the application using the decoder. This choice needs to be clearly documented in the decoder. Beyond the validity at the CBOR level, an application also needs to ascertain that the input is in alignment with the application protocol that is serialized in CBOR.

CBOR encoders do not receive input directly from the network and are thus not directly attackable in the same way as CBOR decoders. However, CBOR encoders often have an API that takes input from another level in the implementation and can be attacked through that API. The design and implementation of that API should assume the behavior of its caller may be based on hostile input or on coding mistakes. It should check inputs for buffer overruns, overflow and underflow of integer arithmetic, and other such errors that are aimed to disrupt the encoder.

Protocols that are used in a security context should be defined in such a way that potential multiple interpretations are reliably reduced to a single interpretation. For example, an attacker could make use of invalid input such as duplicate keys in maps, or exploit different precision in processing numbers to make one application base its decisions on a different interpretation than the one that will be used by a second application. To facilitate consistent interpretation, encoder and decoder implementations used in such contexts should provide at least one strict mode of operation ([Section 5.8](#)).

11. References

11.1. Normative References

- [ECMA262] Ecma International, "ECMAScript 2018 Language Specification", ECMA Standard ECMA-262, 9th Edition, June 2018, <<https://www.ecma-international.org/publications/files/ECMA-ST/Ecma-262.pdf>>.
- [IEEE754] IEEE, "IEEE Standard for Floating-Point Arithmetic", IEEE Std 754-2008.
- [RFC2045] Freed, N. and N. Borenstein, "Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies", [RFC 2045](#), DOI 10.17487/RFC2045, November 1996, <<https://www.rfc-editor.org/info/rfc2045>>.
- [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>>.
- [RFC3339] Klyne, G. and C. Newman, "Date and Time on the Internet: Timestamps", [RFC 3339](#), DOI 10.17487/RFC3339, July 2002, <<https://www.rfc-editor.org/info/rfc3339>>.
- [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>>.
- [RFC3986] Berners-Lee, T., Fielding, R., and L. Masinter, "Uniform Resource Identifier (URI): Generic Syntax", STD 66, [RFC 3986](#), DOI 10.17487/RFC3986, January 2005, <<https://www.rfc-editor.org/info/rfc3986>>.
- [RFC4287] Nottingham, M., Ed. and R. Sayre, Ed., "The Atom Syndication Format", [RFC 4287](#), DOI 10.17487/RFC4287, December 2005, <<https://www.rfc-editor.org/info/rfc4287>>.
- [RFC4648] Josefsson, S., "The Base16, Base32, and Base64 Data Encodings", [RFC 4648](#), DOI 10.17487/RFC4648, October 2006, <<https://www.rfc-editor.org/info/rfc4648>>.
- [RFC8126] Cotton, M., Leiba, B., and T. Narten, "Guidelines for Writing an IANA Considerations Section in RFCs", [BCP 26](#), [RFC 8126](#), DOI 10.17487/RFC8126, June 2017, <<https://www.rfc-editor.org/info/rfc8126>>.

- [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>>.
- [TIME_T] The Open Group Base Specifications, "Vol. 1: Base Definitions, Issue 7", [Section 4.15](#) 'Seconds Since the Epoch', IEEE Std 1003.1, 2013 Edition, 2013, <http://pubs.opengroup.org/onlinepubs/9699919799/basedefs/V1_chap04.html#tag_04_15>.

[11.2](#). Informative References

- [ASN.1] International Telecommunication Union, "Information Technology -- ASN.1 encoding rules: Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER) and Distinguished Encoding Rules (DER)", ITU-T Recommendation X.690, 1994.
- [BSON] Various, "BSON - Binary JSON", 2013, <<http://bsonspec.org/>>.
- [IANA.cbor-simple-values] IANA, "Concise Binary Object Representation (CBOR) Simple Values", <<http://www.iana.org/assignments/cbor-simple-values>>.
- [IANA.cbor-tags] IANA, "Concise Binary Object Representation (CBOR) Tags", <<http://www.iana.org/assignments/cbor-tags>>.
- [MessagePack] Furuhashi, S., "MessagePack", 2013, <<http://msgpack.org/>>.
- [PCRE] Ho, A., "PCRE - Perl Compatible Regular Expressions", 2018, <<http://www.pcre.org/>>.
- [RFC0713] Haverty, J., "MSDTP-Message Services Data Transmission Protocol", [RFC 713](#), DOI 10.17487/RFC0713, April 1976, <<https://www.rfc-editor.org/info/rfc713>>.
- [RFC6838] Freed, N., Klensin, J., and T. Hansen, "Media Type Specifications and Registration Procedures", [BCP 13](#), [RFC 6838](#), DOI 10.17487/RFC6838, January 2013, <<https://www.rfc-editor.org/info/rfc6838>>.
- [RFC7049] Bormann, C. and P. Hoffman, "Concise Binary Object Representation (CBOR)", [RFC 7049](#), DOI 10.17487/RFC7049, October 2013, <<https://www.rfc-editor.org/info/rfc7049>>.

- [RFC7228] Bormann, C., Ersue, M., and A. Keranen, "Terminology for Constrained-Node Networks", [RFC 7228](#), DOI 10.17487/RFC7228, May 2014, <<https://www.rfc-editor.org/info/rfc7228>>.
- [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>>.
- [YAML] Ben-Kiki, O., Evans, C., and I. Net, "YAML Ain't Markup Language (YAML[TM]) Version 1.2", 3rd Edition, October 2009, <<http://www.yaml.org/spec/1.2/spec.html>>.

Appendix A. Examples

The following table provides some CBOR-encoded values in hexadecimal (right column), together with diagnostic notation for these values (left column). Note that the string `"\u00fc"` is one form of diagnostic notation for a UTF-8 string containing the single Unicode character U+00FC, LATIN SMALL LETTER U WITH DIAERESIS (u umlaut). Similarly, `"\u6c34"` is a UTF-8 string in diagnostic notation with a single character U+6C34 (CJK UNIFIED IDEOGRAPH-6C34, often representing "water"), and `"\ud800\udd51"` is a UTF-8 string in diagnostic notation with a single character U+10151 (GREEK ACROPHONIC ATTIC FIFTY STATERS). (Note that all these single-character strings could also be represented in native UTF-8 in diagnostic notation, just not in an ASCII-only specification like the present one.) In the diagnostic notation provided for bignums, their intended numeric value is shown as a decimal number (such as 18446744073709551616) instead of showing a tagged byte string (such as `2(h'01000000000000000000')`).

Diagnostic	Encoded
0	0x00
1	0x01
10	0x0a
23	0x17
24	0x1818
25	0x1819
100	0x1864
1000	0x1903e8
1000000	0x1a000f4240
10000000000000	0x1b000000e8d4a51000
18446744073709551615	0x1bffffffffffffffffffff
18446744073709551616	0xc24901000000000000000000
-18446744073709551616	0x3bffffffffffffffffffff

-18446744073709551617	0xc34901000000000000000000	
-1	0x20	
-10	0x29	
-100	0x3863	
-1000	0x3903e7	
0.0	0xf90000	
-0.0	0xf98000	
1.0	0xf93c00	
1.1	0xfb3ff1999999999999a	
1.5	0xf93e00	
65504.0	0xf97bff	
100000.0	0xfa47c35000	
3.4028234663852886e+38	0xfa7f7fffff	
1.0e+300	0xfb7e37e43c8800759c	
5.960464477539063e-8	0xf90001	
0.00006103515625	0xf90400	
-4.0	0xf9c400	
-4.1	0xfbc01066666666666666	
Infinity	0xf97c00	
NaN	0xf97e00	
-Infinity	0xf9fc00	
Infinity	0xfa7f800000	
NaN	0xfa7fc00000	
-Infinity	0xfaff800000	

Infinity	0xfb7fff0000000000000000	
NaN	0xfb7fff8000000000000000	
-Infinity	0xbffff00000000000000000	
false	0xf4	
true	0xf5	
null	0xf6	
undefined	0xf7	
simple(16)	0xf0	
simple(255)	0xf8ff	
0("2013-03-21T20:04:00Z")	0xc074323031332d30332d32315432303a30343a30305a	
1(1363896240)	0xc11a514b67b0	
1(1363896240.5)	0xc1fb41d452d9ec200000	
23(h'01020304')	0xd74401020304	
24(h'6449455446')	0xd818456449455446	
32("http://www.example.com")	0xd82076687474703a2f2f7777772e6578616d706c652e636f6d	
h''	0x40	
h'01020304'	0x4401020304	
""	0x60	
"a"	0x6161	
"IETF"	0x6449455446	
"\\\""	0x62225c	
"\u00fc"	0x62c3bc	
"\u6c34"	0x63e6b0b4	

"\ud800\udd51"	0x64f0908591	
[]	0x80	
[1, 2, 3]	0x83010203	
[1, [2, 3], [4, 5]]	0x8301820203820405	
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25]	0x98190102030405060708090a0b0c0d0e 0f101112131415161718181819	
{}	0xa0	
{1: 2, 3: 4}	0xa201020304	
{"a": 1, "b": [2, 3]}	0xa26161016162820203	
["a", {"b": "c"}]	0x826161a161626163	
{"a": "A", "b": "B", "c": "C", "d": "D", "e": "E"}	0xa5616161416162614261636143616461 4461656145	
(_ h'0102', h'030405')	0x5f42010243030405ff	
(_ "strea", "ming")	0x7f657374726561646d696e67ff	
[_]	0x9fff	
[_ 1, [2, 3], [_ 4, 5]]	0x9f018202039f0405ffff	
[_ 1, [2, 3], [4, 5]]	0x9f01820203820405ff	
[1, [2, 3], [_ 4, 5]]	0x83018202039f0405ff	
[1, [_ 2, 3], [4, 5]]	0x83019f0203ff820405	
[_ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25]	0x9f0102030405060708090a0b0c0d0e0f 101112131415161718181819ff	
{_ "a": 1, "b": [_ 2, 3]}	0xbf61610161629f0203ffff	
["a", {_ "b": "c"}]	0x826161bf61626163ff	
{_ "Fun": true, "Amt": -2}	0xbf6346756ef563416d7421ff	

Table 4: Examples of Encoded CBOR Data Items

Appendix B. Jump Table

For brevity, this jump table does not show initial bytes that are reserved for future extension. It also only shows a selection of the initial bytes that can be used for optional features. (All unsigned integers are in network byte order.)

Byte	Structure/Semantics
0x00..0x17	Integer 0x00..0x17 (0..23)
0x18	Unsigned integer (one-byte uint8_t follows)
0x19	Unsigned integer (two-byte uint16_t follows)
0x1a	Unsigned integer (four-byte uint32_t follows)
0x1b	Unsigned integer (eight-byte uint64_t follows)
0x20..0x37	Negative integer -1-0x00..-1-0x17 (-1..-24)
0x38	Negative integer -1-n (one-byte uint8_t for n follows)
0x39	Negative integer -1-n (two-byte uint16_t for n follows)
0x3a	Negative integer -1-n (four-byte uint32_t for n follows)
0x3b	Negative integer -1-n (eight-byte uint64_t for n follows)
0x40..0x57	byte string (0x00..0x17 bytes follow)
0x58	byte string (one-byte uint8_t for n, and then n bytes follow)
0x59	byte string (two-byte uint16_t for n, and then n bytes follow)
0x5a	byte string (four-byte uint32_t for n, and then n bytes follow)

0x5b	byte string (eight-byte uint64_t for n, and then n bytes follow)
0x5f	byte string, byte strings follow, terminated by "break"
0x60..0x77	UTF-8 string (0x00..0x17 bytes follow)
0x78	UTF-8 string (one-byte uint8_t for n, and then n bytes follow)
0x79	UTF-8 string (two-byte uint16_t for n, and then n bytes follow)
0x7a	UTF-8 string (four-byte uint32_t for n, and then n bytes follow)
0x7b	UTF-8 string (eight-byte uint64_t for n, and then n bytes follow)
0x7f	UTF-8 string, UTF-8 strings follow, terminated by "break"
0x80..0x97	array (0x00..0x17 data items follow)
0x98	array (one-byte uint8_t for n, and then n data items follow)
0x99	array (two-byte uint16_t for n, and then n data items follow)
0x9a	array (four-byte uint32_t for n, and then n data items follow)
0x9b	array (eight-byte uint64_t for n, and then n data items follow)
0x9f	array, data items follow, terminated by "break"
0xa0..0xb7	map (0x00..0x17 pairs of data items follow)
0xb8	map (one-byte uint8_t for n, and then n pairs of data items follow)
0xb9	map (two-byte uint16_t for n, and then n pairs of data items follow)

0xba	map (four-byte uint32_t for n, and then n pairs of data items follow)	
0xbb	map (eight-byte uint64_t for n, and then n pairs of data items follow)	
0xbf	map, pairs of data items follow, terminated by "break"	
0xc0	Text-based date/time (data item follows; see Section 3.4.2)	
0xc1	Epoch-based date/time (data item follows; see Section 3.4.3)	
0xc2	Positive bignum (data item "byte string" follows)	
0xc3	Negative bignum (data item "byte string" follows)	
0xc4	Decimal Fraction (data item "array" follows; see Section 3.4.5)	
0xc5	Bigfloat (data item "array" follows; see Section 3.4.5)	
0xc6..0xd4	(tagged item)	
0xd5..0xd7	Expected Conversion (data item follows; see Section 3.4.6.2)	
0xd8..0xdb	(more tagged items, 1/2/4/8 bytes and then a data item follow)	
0xe0..0xf3	(simple value)	
0xf4	False	
0xf5	True	
0xf6	Null	
0xf7	Undefined	
0xf8	(simple value, one byte follows)	
0xf9	Half-Precision Float (two-byte IEEE 754)	
0xfa	Single-Precision Float (four-byte IEEE 754)	

0xfb	Double-Precision Float (eight-byte IEEE 754)	
0xff	"break" stop code	
+-----+	+-----+	+-----+

Table 5: Jump Table for Initial Byte

[Appendix C](#). Pseudocode

The well-formedness of a CBOR item can be checked by the pseudocode in Figure 1. The data is well-formed if and only if:

- o the pseudocode does not "fail";
- o after execution of the pseudocode, no bytes are left in the input (except in streaming applications)

The pseudocode has the following prerequisites:

- o take(n) reads n bytes from the input data and returns them as a byte string. If n bytes are no longer available, take(n) fails.
- o uint() converts a byte string into an unsigned integer by interpreting the byte string in network byte order.
- o Arithmetic works as in C.
- o All variables are unsigned integers of sufficient range.


```
well_formed (breakable = false) {
    // process initial bytes
    ib = uint(take(1));
    mt = ib >> 5;
    val = ai = ib & 0x1f;
    switch (ai) {
        case 24: val = uint(take(1)); break;
        case 25: val = uint(take(2)); break;
        case 26: val = uint(take(4)); break;
        case 27: val = uint(take(8)); break;
        case 28: case 29: case 30: fail();
        case 31:
            return well_formed_indefinite(mt, breakable);
    }
    // process content
    switch (mt) {
        // case 0, 1, 7 do not have content; just use val
        case 2: case 3: take(val); break; // bytes/UTF-8
        case 4: for (i = 0; i < val; i++) well_formed(); break;
        case 5: for (i = 0; i < val*2; i++) well_formed(); break;
        case 6: well_formed(); break;      // 1 embedded data item
        case 7: if (ai == 24 && val < 32) fail(); // bad simple
    }
    return mt;                          // finite data item
}

well_formed_indefinite(mt, breakable) {
    switch (mt) {
        case 2: case 3:
            while ((it = well_formed(true)) != -1)
                if (it != mt)          // need finite embedded
                    fail();             //   of same type
            break;
        case 4: while (well_formed(true) != -1); break;
        case 5: while (well_formed(true) != -1) well_formed(); break;
        case 7:
            if (breakable)
                return -1;             // signal break out
            else fail();                // no enclosing indefinite
        default: fail();                // wrong mt
    }
    return 0;                          // no break out
}
```

Figure 1: Pseudocode for Well-Formedness Check

Note that the remaining complexity of a complete CBOR decoder is about presenting data that has been decoded to the application in an appropriate form.

Major types 0 and 1 are designed in such a way that they can be encoded in C from a signed integer without actually doing an if-then-else for positive/negative (Figure 2). This uses the fact that $(-1-n)$, the transformation for major type 1, is the same as $\sim n$ (bitwise complement) in C unsigned arithmetic; $\sim n$ can then be expressed as $(-1)^n$ for the negative case, while 0^n leaves n unchanged for non-negative. The sign of a number can be converted to -1 for negative and 0 for non-negative (0 or positive) by arithmetic-shifting the number by one bit less than the bit length of the number (for example, by 63 for 64-bit numbers).

```
void encode_sint(int64_t n) {
    uint64t ui = n >> 63;    // extend sign to whole length
    mt = ui & 0x20;           // extract major type
    ui ^= n;                   // complement negatives
    if (ui < 24)
        *p++ = mt + ui;
    else if (ui < 256) {
        *p++ = mt + 24;
        *p++ = ui;
    } else
        ...
}
```

Figure 2: Pseudocode for Encoding a Signed Integer

[Appendix D](#). Half-Precision

As half-precision floating-point numbers were only added to IEEE 754 in 2008 [[IEEE754](#)], today's programming platforms often still only have limited support for them. It is very easy to include at least decoding support for them even without such support. An example of a small decoder for half-precision floating-point numbers in the C language is shown in Figure 3. A similar program for Python is in Figure 4; this code assumes that the 2-byte value has already been decoded as an (unsigned short) integer in network byte order (as would be done by the pseudocode in [Appendix C](#)).


```
#include <math.h>

double decode_half(unsigned char *halfp) {
    int half = (halfp[0] << 8) + halfp[1];
    int exp = (half >> 10) & 0x1f;
    int mant = half & 0x3ff;
    double val;
    if (exp == 0) val = ldexp(mant, -24);
    else if (exp != 31) val = ldexp(mant + 1024, exp - 25);
    else val = mant == 0 ? INFINITY : NAN;
    return half & 0x8000 ? -val : val;
}
```

Figure 3: C Code for a Half-Precision Decoder

```
import struct
from math import ldexp

def decode_single(single):
    return struct.unpack("!f", struct.pack("!I", single))[0]

def decode_half(half):
    valu = (half & 0x7fff) << 13 | (half & 0x8000) << 16
    if ((half & 0x7c00) != 0x7c00):
        return ldexp(decode_single(valu), 112)
    return decode_single(valu | 0x7f800000)
```

Figure 4: Python Code for a Half-Precision Decoder

[Appendix E](#). Comparison of Other Binary Formats to CBOR's Design Objectives

The proposal for CBOR follows a history of binary formats that is as long as the history of computers themselves. Different formats have had different objectives. In most cases, the objectives of the format were never stated, although they can sometimes be implied by the context where the format was first used. Some formats were meant to be universally usable, although history has proven that no binary format meets the needs of all protocols and applications.

CBOR differs from many of these formats due to it starting with a set of objectives and attempting to meet just those. This section compares a few of the dozens of formats with CBOR's objectives in order to help the reader decide if they want to use CBOR or a different format for a particular protocol or application.

Note that the discussion here is not meant to be a criticism of any format: to the best of our knowledge, no format before CBOR was meant

to cover CBOR's objectives in the priority we have assigned them. A brief recap of the objectives from [Section 1.1](#) is:

1. unambiguous encoding of most common data formats from Internet standards
2. code compactness for encoder or decoder
3. no schema description needed
4. reasonably compact serialization
5. applicability to constrained and unconstrained applications
6. good JSON conversion
7. extensibility

[E.1.](#) ASN.1 DER, BER, and PER

[ASN.1] has many serializations. In the IETF, DER and BER are the most common. The serialized output is not particularly compact for many items, and the code needed to decode numeric items can be complex on a constrained device.

Few (if any) IETF protocols have adopted one of the several variants of Packed Encoding Rules (PER). There could be many reasons for this, but one that is commonly stated is that PER makes use of the schema even for parsing the surface structure of the data stream, requiring significant tool support. There are different versions of the ASN.1 schema language in use, which has also hampered adoption.

[E.2.](#) MessagePack

[MessagePack] is a concise, widely implemented counted binary serialization format, similar in many properties to CBOR, although somewhat less regular. While the data model can be used to represent JSON data, MessagePack has also been used in many remote procedure call (RPC) applications and for long-term storage of data.

MessagePack has been essentially stable since it was first published around 2011; it has not yet had a transition. The evolution of MessagePack is impeded by an imperative to maintain complete backwards compatibility with existing stored data, while only few bytecodes are still available for extension. Repeated requests over the years from the MessagePack user community to separate out binary and text strings in the encoding recently have led to an extension proposal that would leave MessagePack's "raw" data ambiguous between

its usages for binary and text data. The extension mechanism for MessagePack remains unclear.

[E.3.](#) BSON

[BSON] is a data format that was developed for the storage of JSON-like maps (JSON objects) in the MongoDB database. Its major distinguishing feature is the capability for in-place update, foregoing a compact representation. BSON uses a counted representation except for map keys, which are null-byte terminated. While BSON can be used for the representation of JSON-like objects on the wire, its specification is dominated by the requirements of the database application and has become somewhat baroque. The status of how BSON extensions will be implemented remains unclear.

[E.4.](#) MSDTP: [RFC 713](#)

Message Services Data Transmission (MSDTP) is a very early example of a compact message format; it is described in [[RFC0713](#)], written in 1976. It is included here for its historical value, not because it was ever widely used.

[E.5.](#) Conciseness on the Wire

While CBOR's design objective of code compactness for encoders and decoders is a higher priority than its objective of conciseness on the wire, many people focus on the wire size. Table 6 shows some encoding examples for the simple nested array [1, [2, 3]]; where some form of indefinite-length encoding is supported by the encoding, [_ 1, [2, 3]] (indefinite length on the outer array) is also shown.

Format	[1, [2, 3]]	[_ 1, [2, 3]]
RFC 713	c2 05 81 c2 02 82 83	
ASN.1 BER	30 0b 02 01 01 30 06 02 01 02 02 01 03	30 80 02 01 01 30 06 02 01 02 02 01 03 00 00
MessagePack	92 01 92 02 03	
BSON	22 00 00 00 10 30 00 01 00 00 00 04 31 00 13 00 00 00 10 30 00 02 00 00 00 10 31 00 03 00 00 00 00 00	
CBOR	82 01 82 02 03	9f 01 82 02 03 ff

Table 6: Examples for Different Levels of Conciseness

Appendix F. Changes from [RFC 7049](#)

The following is a list of known changes from [RFC 7049](#). This list is non-authoritative. It is meant to help reviewers see the significant differences.

- o Updated reference for [\[RFC4627\]](#) to [\[RFC8259\]](#) in many places
- o Updated reference for [\[CNN-TERMS\]](#) to [\[RFC7228\]](#)
- o Added a comment to the last example in [Section 2.2.1](#) (added "Second value")
- o Fixed a bug in the example in [Section 2.4.2](#) ("29" -> "49")
- o Fixed a bug in the last paragraph of [Section 3.6](#) ("0b000_11101" -> "0b000_11001")

Acknowledgements

CBOR was inspired by MessagePack. MessagePack was developed and promoted by Sadayuki Furuhashi ("frsyuki"). This reference to MessagePack is solely for attribution; CBOR is not intended as a version of or replacement for MessagePack, as it has different design goals and requirements.

The need for functionality beyond the original MessagePack Specification became obvious to many people at about the same time around the year 2012. BinaryPack is a minor derivation of MessagePack that was developed by Eric Zhang for the binaryjs project. A similar, but different, extension was made by Tim Caswell for his msgpack-js and msgpack-js-browser projects. Many people have contributed to the discussion about extending MessagePack to separate text string representation from byte string representation.

The encoding of the additional information in CBOR was inspired by the encoding of length information designed by Klaus Hartke for CoAP.

This document also incorporates suggestions made by many people, notably Dan Frost, James Manger, Jeffrey Yaskin, Joe Hildebrand, Keith Moore, Laurence Lundblade, Matthew Lepinski, Michael Richardson, Nico Williams, Phillip Hallam-Baker, Ray Polk, Tim Bray, Tony Finch, Tony Hansen, and Yaron Sheffer.

Authors' Addresses

Carsten Bormann
Universitaet Bremen TZI
Postfach 330440
D-28359 Bremen
Germany

Phone: +49-421-218-63921
EMail: cabo@tzi.org

Paul Hoffman
ICANN

EMail: paul.hoffman@icann.org

