Network Working Group
Internet-Draft
Intended status: Standards Track
Expires: September 3, 2018

C. Bormann
Universitaet Bremen TZI
P. Hoffman
ICANN

March 02, 2018

Concise Binary Object Representation (CBOR)
draft-ietf-cbor-7049bis-02

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.

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 September 3, 2018.

Table of Contents

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 [RFC7159]. It is important to note that this is not a proposal that the grammar in RFC 7159 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).

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", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119, BCP 14 [RFC2119] and indicate requirement levels for compliant CBOR implementations.

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 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 is not followed by extraneous data.

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.

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:

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

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:

- "false", "true", "null", and "undefined" (simple values identified by 20..23)

- integer and floating point values with a larger range and precision than the above (tags 2 to 5)

- application data types such as a point in time (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" 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 their names 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 that 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, map keys "0" and "0.0" 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 as described in this section. The encoding is summarized in Table 5.

The initial byte of each 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). When the value of the additional information is less than 24, it is directly used as a small unsigned integer. When it is 24 to 27, the additional bytes for a variable-length integer immediately follow; the values 24 to 27 of the additional information specify that its length is a 1-, 2-, 4-, or 8-byte unsigned integer, respectively. Additional information value 31 is used for indefinite-length items, described in Section 3.2. Additional information values 28 to 30 are reserved for future expansion.

In all additional information values, the resulting integer is interpreted depending on the major type. It may represent the actual data: for example, in integer types, the resulting integer is used for the value itself. It may instead supply length information: for example, in byte strings it gives the length of the byte string data that follows.

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 unsigned integer. The 5-bit additional information is either the integer itself (for additional information values 0 through 23) or the length of additional data. Additional information 24 means the value is represented in an additional uint8_t, 25 means a uint16_t, 26 means a uint32_t, and 27 means a uint64_t. 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. The encoding follows the rules for unsigned integers (major type 0), except that the value is then -1 minus the encoded unsigned integer. 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 string's length in bytes is represented following the rules for positive integers (major type 0). 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, specifically a string of Unicode characters that is encoded as UTF-8 [RFC3629]. The format of this type is identical to that of byte strings (major type 2), that is, as with major type 2, the length gives the number of bytes. 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 array's length follows the rules for byte strings (major type 2), except that the length denotes the number of data items, not the length in bytes that the array takes up. 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 map's length follows the rules for byte strings (major type 2), except that the length denotes the number of pairs, not the length in bytes that the map takes up. 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 4.7.

Major type 6: optional semantic tagging of other major types. See Section 3.4.

Major type 7: floating-point numbers and simple data types that need no content, 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 8 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. Indefinite-Length Arrays and Maps

Indefinite-length arrays and maps are simply opened without indicating the number of data items that will be included in the array or map, using the additional information value of 31. The initial major type and additional information byte is followed by the elements of the array or map, just as they would be in other arrays or maps. The end of the array or map is indicated by encoding a "break" stop code in a place where the next data item would normally have been included. The "break" is encoded with major type 7 and additional information value 31 (0b111_11111) but is not itself a data item: it is just a syntactic feature to close the array or map. That is, the "break" stop code comes after the last item in the array or map, and it cannot occur anywhere else in place of a data item.

In this way, indefinite-length arrays and maps look identical to other arrays and maps except for beginning with the additional information value 31 and ending with the "break" stop code.

Arrays and maps with indefinite lengths allow any number of items (for arrays) and key/value pairs (for maps) to be given before the "break" stop code. 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"
```

3.2.2. Indefinite-Length Byte Strings and Text Strings

Indefinite-length byte strings and text strings are actually a concatenation of zero or more definite-length byte or text strings ("chunks") that are together treated as one contiguous string. Indefinite-length strings are opened with the major type and additional information value of 31, but what follows are a series of byte or text strings that have definite lengths (the chunks). The end of the series of chunks is indicated by encoding the "break" stop code (0b111_11111) in a place where the next chunk in the series would occur. The contents of the chunks are concatenated together, and the overall length of the indefinite-length string will be the sum of the lengths of all of the chunks. In summary, an indefinite-length string is encoded similarly to how an indefinite-length array of its chunks would be encoded, except that the major type of the indefinite-length string is that of a (text or byte) string and matches the major types of its chunks.

For indefinite-length byte strings, every data item (chunk) between the indefinite-length indicator and the "break" MUST be a definite-length byte string item; if the parser sees any item type other than a byte string before it sees the "break", it is an error.

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.

Text strings with indefinite lengths act the same as byte strings with indefinite lengths, except that all their chunks MUST be definite-length text strings. 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.

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)
31	"break" stop code for indefinite-length items

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

The 5-bit values of 25, 26, and 27 are for 16-bit, 32-bit, and 64-bit IEEE 754 binary floating-point values. 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.)

An encoder MUST NOT encode False as the two-byte sequence of 0xf814, MUST NOT encode True as the two-byte sequence of 0xf815, MUST NOT encode Null as the two-byte sequence of 0xf816, and MUST NOT encode Undefined value as the two-byte sequence of 0xf817. A decoder MUST treat these two-byte sequences as an error. Similar prohibitions apply to the unassigned simple values as well.

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 integer number as indicated by the tag's integer value; 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 can be carried by a tag.

The initial bytes of the tag follow the rules for positive integers (major type 0). The tag is followed by a single data item of any type. For example, assume that a byte string of length 12 is marked with a tag to indicate it is a positive bignum (Section 3.4.2). 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 is directly followed by 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 8.2. Table 3 provides a list of initial values, with definitions in the rest of this section.

Tag	Data Item	Semantics
0	UTF-8 string	Standard date/time string; see Section 3.4.1
1	multiple	Epoch-based date/time; see Section 3.4.1
2	byte string	Positive bignum; see Section 3.4.2
3	byte string	Negative bignum; see Section 3.4.2
4	array	Decimal fraction; see Section 3.4.3
5	array	Bigfloat; see Section 3.4.3
6..20	(Unassigned)	(Unassigned)
21	multiple	Expected conversion to base64url encoding; see Section 3.4.4.2
22	multiple	Expected conversion to base64

		encoding; see Section 3.4.4.2
23	multiple	Expected conversion to base16 encoding; see Section 3.4.4.2
24	byte string	Encoded CBOR data item; see Section 3.4.4.1
25..31	(Unassigned)	(Unassigned)
32	UTF-8 string	URI; see Section 3.4.4.3
33	UTF-8 string	base64url; see Section 3.4.4.3
34	UTF-8 string	base64; see Section 3.4.4.3
35	UTF-8 string	Regular expression; see Section 3.4.4.3
36	UTF-8 string	MIME message; see Section 3.4.4.3
37..55798	(Unassigned)	(Unassigned)
55799	multiple	Self-describe CBOR; see Section 3.4.5
55800+	(Unassigned)	(Unassigned)

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.

Tag value 0 is for date/time strings that follow the standard format described in [RFC3339], as refined by Section 3.3 of [RFC4287].

Tag value 1 is for numerical representation of seconds relative to 1970-01-01T00:00Z in UTC time. (For the non-negative values that the Portable Operating System Interface (POSIX) defines, the number of seconds is counted in the same way as for POSIX "seconds since the epoch" [TIME_T].) The tagged item can be a positive or negative integer (major types 0 and 1), or a floating-point number (major type 7 with additional information 25, 26, or 27). Note that the number can be negative (time before 1970-01-01T00:00Z) and, if a floating-point number, indicate fractional seconds.

3.4.2. Bignums

Protocols using tag values 2 and 3 extend the generic data model (Section 2) with "bignums" representing arbitrary 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.

Bignums are encoded as a byte string data item, which is interpreted as an unsigned integer n in network byte order. For tag value 2, the value of the bignum is n. For tag value 3, the value of the bignum is -1 - n. Decoders that understand these tags MUST be able to decode bignums that have leading zeroes.

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
```

3.4.3. Decimal Fractions and Bigfloats

Protocols using tag value 4 extend the generic data model with data items representing arbitrary-length decimal fractions m*(10*e). Protocols using tag value 5 extend the generic data model with data items representing arbitrary-length binary fractions m*(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*(10**e). Bigfloats (tag 5) use base-2 exponents; the value of a bigfloat data item is m*(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 3.4.2).

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.

3.4.4. 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.

3.4.4.1. 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 parsed. Tag 24 (CBOR data item) can be used to tag the embedded byte string as a data item encoded in CBOR format.

3.4.4.2. 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, padding is not used (see Section 3.2 of RFC 4648); that is, all trailing equals signs ("=") are removed from the base64url-encoded string. Later tags might be defined for other data encodings of RFC 4648 or for other ways to encode binary data in strings.

3.4.4.3. 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.

- Tag 32 is for URIs, as defined in [RFC3986];

- Tags 33 and 34 are for base64url- and base64-encoded text strings, as defined in [RFC4648];

- Tag 35 is for regular expressions in Perl Compatible Regular Expressions (PCRE) / JavaScript syntax [ECMA262].

- Tag 36 is for MIME messages (including all headers), as defined in [RFC2045];

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.

3.4.5. Self-Describe 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 and disambiguating metadata is not in use. 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 appears not to be in use as a distinguishing mark for frequently used file types. In particular, it is not a valid start of a Unicode text in any Unicode encoding if followed by a valid CBOR data item.

For instance, a decoder might be able to parse 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.

3.5. 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:

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

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:

- "false", "true", "null", and "undefined" (simple values identified by 20..23)
- integer and floating point values with a larger range and precision than the above (tags 2 to 5)
- 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.

A specific data model 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 their names in the generic data model ("negative integer", "array") instead of by referring to aspects of their CBOR representation ("major type 1", "major type 4").

4. 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".

This specification puts no restrictions on CBOR-based protocols. 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. It is advisory only and explicitly excludes any language from RFC 2119 other than words that could be interpreted as "MAY" in the sense of RFC 2119.

4.1. 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.

4.2. Generic Encoders and Decoders

A generic CBOR decoder can decode all well-formed CBOR data and present them to an application. CBOR data is well-formed if it uses the initial bytes, as well as the byte strings and/or data items that are implied by their values, in the manner defined by CBOR, and no extraneous data follows (Appendix C).

Even though CBOR attempts to minimize these cases, not all well-formed CBOR data is valid: for example, the format excludes simple values below 32 that are encoded with an extension byte. Also, specific tags may make semantic constraints that may be violated, such as by including a tag in a bignum tag or by following a byte string within a date tag. Finally, the data may be invalid, such as invalid UTF-8 strings or date strings that do not conform to [RFC3339]. 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 4.5).

Generic decoders provide ways to present well-formed CBOR values, both valid and invalid, to an application. The diagnostic notation (Section 7) 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.

4.3. Syntax Errors

A decoder encountering a CBOR data item that is not well-formed generally can choose to completely fail the decoding (issue an error and/or stop processing altogether), substitute the problematic data and data items using a decoder-specific convention that clearly indicates there has been a problem, or take some other action.

4.3.1. Incomplete CBOR Data Items

The representation of a CBOR data item has a specific length, determined by its initial bytes and by the structure of any data items enclosed in the data items. If less data is available, this can be treated as a syntax error. A decoder may also implement incremental parsing, that is, decode the data item as far as it is available and present the data found so far (such as in an event-based interface), with the option of continuing the decoding once further data is available.

Examples of incomplete data items include:

- A decoder expects a certain number of array or map entries but instead encounters the end of the data.
- A decoder processes what it expects to be the last pair in a map and comes to the end of the data.
- A decoder has just seen a tag and then encounters the end of the data.
- A decoder has seen the beginning of an indefinite-length item but encounters the end of the data before it sees the "break" stop code.

4.3.2. Malformed Indefinite-Length Items

Examples of malformed indefinite-length data items include:

- Within an indefinite-length byte string or text, a decoder finds an item that is not of the appropriate major type before it finds the "break" stop code.

- Within an indefinite-length map, a decoder encounters the "break" stop code immediately after reading a key (the value is missing).

Another error is finding a "break" stop code at a point in the data where there is no immediately enclosing (unclosed) indefinite-length item.

4.3.3. Unknown Additional Information Values

At the time of writing, some additional information values are unassigned and reserved for future versions of this document (see Section 6.2). Since the overall syntax for these additional information values is not yet defined, a decoder that sees an additional information value that it does not understand cannot continue parsing.

4.4. Other Decoding Errors

A CBOR data item may be syntactically well-formed but present a problem with interpreting the data encoded in it in the CBOR data model. Generally speaking, a decoder that finds a data item with such a problem might issue a warning, might stop processing altogether, might handle the error and make the problematic value available to the application as such, or take some other type of action.

Such problems might include:

Duplicate keys in a map: Generic decoders (Section 4.2) 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 4.7).

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.

4.5. 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.

4.6. Numbers

An application or protocol that uses CBOR might restrict the representations of numbers. For instance, a protocol that only deals with integers might say that floating-point numbers may not be used and that decoders of that protocol do not need to be able to handle floating-point numbers. Similarly, a protocol or application that uses CBOR might say that decoders need to be able to handle either type of number.

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.

4.7. 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 objects, a key of integer 1 cannot be distinguished from a key of string "1". This means that, if integer keys are used, the simultaneous use of string keys that look like numbers needs to be avoided. Again, this leads to the conclusion that keys should be of a single CBOR type.

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 should make an intentional decision about what to do when a receiving application does see multiple identical keys in a map. The resulting rule in the protocol should 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 4.10).

The CBOR data model for maps does not allow ascribing semantics to the order of the key/value pairs in the map representation. Thus, it would be a very bad practice to define a CBOR-based protocol in such a way that changing the key/value pair order in a map would change the semantics, apart from trivial aspects (cache usage, etc.). (A CBOR-based protocol can prescribe a specific order of serialization, such as for canonicalization.)

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.

4.7.1. Equivalence of Keys

This notion of equivalence must be used to determine whether keys in maps are duplicates or distinct.

- All numbers are compared by their numeric value.
  - Integer data items with the same value are equal regardless of how many bytes are used to encode them.
  - Floating point data items with the same value are equal regardless of how many bytes are used to encode them.
  - An integer value encoded as a floating point data item is equivalent to the same value encoded as an integer
- Byte strings and text strings are compared by their binary content.
  - A different length encoding has no effect on equivalence.
  - A byte string is equal to a text string if they have the same binary content.

- Two arrays are equal if all their items are in the same order and equal.

- 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.

- Tags have no effect in determining equality of a data item, if two items are equal then they are equal irrespective of any tags that either or both may have.

- Simple values are equal if they simply have the same value.

Nothing else is equal, a simple value 2 is not equivalent to an integer 2 and an array cannot be equivalent to a map with the same values and sequential integer keys.

4.8. 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.

4.9. Canonical CBOR

Some protocols may want encoders to only emit CBOR in a particular canonical format; those protocols might also have the decoders check that their input is canonical. Those protocols are free to define what they mean by a canonical 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 canonical format.

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

- Integers 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.

- The expression of lengths in major types 2 through 5 MUST be as short as possible. The rules for these lengths follow the above rule for integers.

- The keys in every map MUST be sorted in the bytewise lexicographic order of their canonical 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.

- Indefinite-length items MUST not appear. They can be encoded as definite-length items instead.

If a protocol allows for IEEE floats, then additional canonicalization 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 canonicalization. The absence or presence of tags in a canonical format is determined by the optionality of the tags in the protocol. In a CBOR-based protocol that allows optional tagging anywhere, the canonical format must not allow them. In a protocol that requires tags in certain places, the tag needs to appear in the canonical format. A CBOR-based protocol that uses canonicalization 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 canonical encodings. For example:

- If a protocol includes a field that can express floating values (Section 3.3), the protocol's canonicalization 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.

- If a protocol includes a field that can express integers larger than 2^64 using tag 2 (Section 3.4.2), the protocol's canonicalization needs to specify whether small integers are expressed using the tag or major types 0 and 1.

- A protocol might give encoders the choice of representing a URL as either a text string or, using Section 3.4.4.3, tag 32 containing a text string. This protocol's canonicalization needs to either require that the tag is present or require that it's absent, not allow either one.

4.9.1. Length-first map key ordering

The core canonicalization requirements sort map keys in a different order from the one suggested by [RFC7049]. Protocols that need to be compatible with [RFC7049]'s order can instead be specified in terms of this specification's "length-first core canonicalization requirements":

A CBOR encoding satisfies the "length-first core canonicalization requirements" if it satisfies the core canonicalization 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 canonicalization 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.

4.10. Strict Mode

Some areas of application of CBOR do not require canonicalization (Section 4.9) 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 reliably reject data items with syntax errors (Section 4.3). It will also expend the effort to reliably detect other decoding errors (Section 4.4). In particular, 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:

- a map (major type 5) that has more than one entry with the same key
- a tag that is used on a data item of the incorrect type
- 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:

- It can report an error (and not return data).
- 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.

5. 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.

5.1. 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.

- An integer (major type 0 or 1) becomes a JSON number.
- 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.
- A UTF-8 string (major type 3) becomes a JSON string. Note that JSON requires escaping certain characters (RFC 7159, 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.
- An array (major type 4) becomes a JSON array.
- 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.
- False (major type 7, additional information 20) becomes a JSON false.
- True (major type 7, additional information 21) becomes a JSON true.
- Null (major type 7, additional information 22) becomes a JSON null.

- 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.

- Any other simple value (major type 7, any additional information value not yet discussed) is represented by the substitute value.

- 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.)

- A byte string with an encoding hint (major type 6, tag value 21 through 23) is encoded as described and becomes a JSON string.

- 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.

- Indefinite-length items are made definite before conversion.

5.2. 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:

- 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.)

- 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.

6. 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.

6.1. 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:

- 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 8.1 is the appropriate way to address the extensibility of this codepoint space.

- 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 8.2 is the appropriate way to address the extensibility of this codepoint space.

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

6.2. 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 >= 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.

7. 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 7159, 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.

7.1. 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").

8. 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 [RFC5226]. IANA has also assigned a new MIME media type and an associated Constrained Application Protocol (CoAP) Content-Format entry.

8.1. Simple Values Registry

IANA has created the "Concise Binary Object Representation (CBOR) Simple Values" registry. 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.

8.2. Tags Registry

IANA has created the "Concise Binary Object Representation (CBOR) Tags" registry. The initial values are shown in Table 3.

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:

- Data item
- Semantics (short form)

In addition, First Come First Served requests should include:

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

8.3. Media Type ("MIME Type")

The Internet media type [RFC6838] for CBOR data is application/cbor.

Type name: application

Subtype name: cbor

Required parameters: n/a

Optional parameters: n/a

Encoding considerations: binary

Security considerations: See Section 9 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>

8.4. CoAP Content-Format

Media Type: application/cbor

Encoding: -

Id: 60

Reference: [RFCthis]

8.5. 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 9 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.

9. 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.

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.)

Applications where a CBOR data item is examined by a gatekeeper function and later used by a different application may exhibit vulnerabilities when multiple interpretations of the data item are

possible. For example, an attacker could make use of duplicate keys in maps and precision issues in numbers to make the gatekeeper base its decisions on a different interpretation than the one that will be used by the second application. Protocols that are used in a security context should be defined in such a way that these multiple interpretations are reliably reduced to a single one. To facilitate this, encoder and decoder implementations used in such contexts should provide at least one strict mode of operation (Section 4.10).

10. 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 recent 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, Joe Hildebrand, Keith Moore, Matthew Lepinski, Nico Williams, Phillip Hallam-Baker, Ray Polk, Tim Bray, Tony Finch, Tony Hansen, and Yaron Sheffer.

11. References

11.1. Normative References

[ECMA262] European Computer Manufacturers Association, "ECMAScript Language Specification 5.1 Edition", ECMA Standard ECMA-262, June 2011, <http://www.ecma-international.org/publications/files/ecma-st/ECMA-262.pdf>.

[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>.

[RFC5226] Narten, T. and H. Alvestrand, "Guidelines for Writing an IANA Considerations Section in RFCs", RFC 5226, DOI 10.17487/RFC5226, May 2008, <https://www.rfc-editor.org/info/rfc5226>.

[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/>.

[MessagePack]
Furuhashi, S., "MessagePack", 2013, <http://msgpack.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>.

[RFC7159] Bray, T., Ed., "The JavaScript Object Notation (JSON) Data Interchange Format", RFC 7159, DOI 10.17487/RFC7159, March 2014, <https://www.rfc-editor.org/info/rfc7159>.

[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>.

[UBJSON] The Buzz Media, "Universal Binary JSON Specification", 2013, <http://ubjson.org/>.

[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'010000000000000000')).

Diagnostic	Encoded
0	0x00
1	0x01
10	0x0a
23	0x17
24	0x1818
25	0x1819
100	0x1864
1000	0x1903e8
1000000	0x1a000f4240
1000000000000	0x1b000000e8d4a51000
18446744073709551615	0x1bffffffffffffffff
18446744073709551616	0xc249010000000000000000
-18446744073709551616	0x3bffffffffffffffff

-18446744073709551617	0xc349010000000000000000
-1	0x20
-10	0x29
-100	0x3863
-1000	0x3903e7
0.0	0xf90000
-0.0	0xf98000
1.0	0xf93c00
1.1	0xfb3ff199999999999a
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	0xfbc010666666666666
Infinity	0xf97c00
NaN	0xf97e00
-Infinity	0xf9fc00
Infinity	0xfa7f800000
NaN	0xfa7fc00000
-Infinity	0xfaff800000

Infinity	0xfb7ff0000000000000
NaN	0xfb7ff8000000000000
-Infinity	0xfbfff0000000000000
false	0xf4
true	0xf5
null	0xf6
undefined	0xf7
simple(16)	0xf0
simple(24)	0xf818
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]	0x98190102030405060708090a0b0c0d0e0f101112131415161718181819
{}	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"}	0xa56161614161626142616361436164614461656145
(_ 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]	0x9f0102030405060708090a0b0c0d0e0f101112131415161718181819ff
{_ "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 3.4.1)
0xc1	Epoch-based date/time (data item follows; see Section 3.4.1)
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 3.4.3)
0xc5	Bigfloat (data item "array" follows; see Section 3.4.3)
0xc6..0xd4	(tagged item)
0xd5..0xd7	Expected Conversion (data item follows; see Section 3.4.4.2)
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:

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

The pseudocode has the following prerequisites:

- 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.
- uint() converts a byte string into an unsigned integer by interpreting the byte string in network byte order.
- Arithmetic works as in C.
- 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
  }
  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 parsed 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 ~n (bitwise complement) in C unsigned arithmetic; ~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, 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. UBJSON

[UBJSON] has a design goal to make JSON faster and somewhat smaller, using a binary format that is limited to exactly the data model JSON uses. Thus, there is expressly no intention to support, for example, binary data; however, there is a "high-precision number", expressed as a character string in JSON syntax. UBJSON is not optimized for code compactness, and its type byte coding is optimized for human recognition and not for compact representation of native types such as small integers. Although UBJSON is mostly counted, it provides a reserved "unknown-length" value to support streaming of arrays and maps (JSON objects). Within these containers, UBJSON also has a "Noop" type for padding.

E.5. 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.6. 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	
UBJSON	61 02 42 01 61 02 42 02 42 03	61 ff 42 01 61 02 42 02 42 03 45
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.

- Updated reference for [RFC4267] to [RFC7159] in many places
- Updated reference for [CNN-TERMS] to [RFC7228]
- Added a comment to the last example in Section 2.2.1 (added "Second value")
- Fixed a bug in the example in Section 2.4.2 ("29" -> "49")
- Fixed a bug in the last paragraph of Section 3.6 ("0b000_11101" -> "0b000_11001")

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

Network Working Group
Internet-Draft
Obsoletes: 7049 (if approved)
Intended status: Standards Track
Expires: 3 April 2021

C. Bormann
Universitaet Bremen TZI
P. Hoffman
ICANN

30 September 2020

Concise Binary Object Representation (CBOR)
draft-ietf-cbor-7049bis-16

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 is a revised edition of RFC 7049, with editorial improvements, added detail, and fixed errata. This revision formally obsoletes RFC 7049, while keeping full compatibility of the interchange format from RFC 7049. It does not create a new version of the format.

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 3 April 2021.

Copyright Notice

Table of Contents

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 is a revised edition of [RFC7049], with editorial improvements, added detail, and fixed errata. This revision formally obsoletes RFC 7049, while keeping full compatibility of the interchange format from RFC 7049. It does not create a new version of the format.

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 the implementation complexity limiting how much effort can go into achieving that compactness. 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; the former can be addressed specifically by using "encoded data item".

Decoder: A process that decodes a well-formed encoded 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 (well-formed) 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 (see [RFC8742] for one application). 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 (Section 5.3).

Expected: Besides its normal English meaning, the term "expected" is used to describe requirements beyond CBOR validity that an application has on its input data. Well-formed (processable at all), valid (checked by a validity-checking generic decoder), and expected (checked by the application) form a hierarchy of layers of acceptability.

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.

Terms and concepts for floating-point values such as Infinity, NaN (not a number), negative zero, and subnormal are defined in [IEEE754].

Where bit arithmetic or data types are explained, this document uses the notation familiar from the programming language C [C], except that "**" denotes exponentiation and ".." denotes a range that includes both ends given. Examples and pseudocode assume that signed integers use two's complement representation and that right shifts of signed integers perform sign extension; these assumptions are also specified in Sections 6.8.2 and 7.6.7 of the 2020 version of C++, successor of [Cplusplus17].

Similar to the "0x" notation for hexadecimal numbers, numbers in binary notation are prefixed with "0b". Underscores can be added to 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.

Words may be _italicized_ for emphasis; in the plain text form of this specification this is indicated by surrounding words with underscore characters. Verbatim text (e.g., names from a programming language) may be set in "monospace" type; in plain text this is approximated somewhat ambiguously by surrounding the text in double quotes (which also retain their usual meaning).

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 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 defined in Section 3, a data item is one of:

* an integer in the range -2**64..2**64-1 inclusive
* a simple value, identified by a number between 0 and 255, but distinct from that number itself
* a floating-point value, distinct from an integer, out of the set representable by IEEE 754 binary64 (including non-finites) [IEEE754]
* a sequence of zero or more bytes ("byte string")
* a sequence of zero or more Unicode code points ("text string")
* a sequence of zero or more data items ("array")
* a mapping (mathematical function) from zero or more data items ("keys") each to a data item ("values"), ("map")
* a tagged data item ("tag"), comprising a tag number (an integer in the range 0..2**64-1) and the tag content (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 are not visible at the generic data model level, including the number of bytes of the encoded floating-point 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 number, (collectively "the argument", see Section 3) can be encoded.

2.1. Extended Generic Data Models

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

* "false", "true", "null", and "undefined" (simple values identified by 20..23)

* integer and floating-point values with a larger range and precision than the above (tag numbers 2 to 5)

* application data types such as a point in time or an RFC 3339 date/time string (tag numbers 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 simple values or generic tags.

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 tag numbers, 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, even while encoded as different major types, 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 7 in Appendix B, indexed by the initial byte. 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; all are described in Section 3.2.

The initial byte and any additional bytes consumed to construct the argument are collectively referred to as the "head" of the data item.

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 7). 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 unsigned 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 (Section 1.2). 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 (that of Unicode) and encoding (UTF-8). 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") nor as 0x5c7530303061 (the characters "\", "u", "0", "0", "0", and "a").

Major type 4: an array of data items. In other formats, arrays are also called lists, sequences, or tuples (a "CBOR sequence" is something slightly different, though [RFC8742]). 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 4, additional information 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 5, additional information 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. Because items in a map come in pairs, their total number is always even: A map that contains an odd number of items (no value data present after the last key data item) is not well-formed. 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 ("tag") whose tag number, an

integer in the range 0..2**64-1 inclusive, is the argument and whose enclosed data item ("tag content") is the single encoded data item that follows the head. 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 7).

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.

Table 1 summarizes the major types defined by CBOR, ignoring the next section for now. The number N in this table stands for the argument, mt for the major type.

mt	Meaning	Content
0	unsigned integer N	-
1	negative integer -1-N	-
2	byte string	N bytes
3	text string	N bytes (UTF-8 text)
4	array	N data items (elements)
5	map	2N data items (key/value pairs)
6	tag of number N	1 data item
7	simple/float	-

Table 1: Overview over the definite-length use of CBOR major types (mt = major type, N = argument)

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 ability to start sending a data item before all of it is known is often referred to as "streaming" within that data item.)

Indefinite-length arrays and maps are dealt with differently than indefinite-length strings (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 zero or more 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 for byte string or text string with an additional information value of 31, followed by a series of zero or more strings of the specified type ("chunks") that have definite lengths, and finished 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 no chunks are present, the data item is an empty string of the specified type. Zero-length chunks, while not particularly useful, are permitted.

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.

The design does not allow nesting indefinite-length strings as chunks into indefinite-length strings. If it were allowed, it would require decoder implementations to keep a stack, or at least a count, of nesting levels. It is unnecessary on the encoder side because the inner indefinite-length string would consist of chunks, and these could instead be put directly into the outer indefinite-length string.

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 UTF-8 bytes of a single Unicode code point (scalar value) cannot be spread between chunks: a new chunk of a text string can only be started at a code point boundary.

For example, assume an encoded data item consisting of the bytes:

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.

3.2.4. Summary of indefinite-length use of major types

Table 2 summarizes the major types defined by CBOR as used for indefinite length encoding (with additional information set to 31). mt stands for the major type.

mt	Meaning	enclosed up to "break" stop code
0	(not well-formed)	-
1	(not well-formed)	-
2	byte string	definite-length byte strings
3	text string	definite-length text strings
4	array	data items (elements)
5	map	data items (key/value pairs)
6	(not well-formed)	-
7	"break" stop code	-

Table 2: Overview over the indefinite-length use of CBOR major types (mt = major type, additional information = 31)

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 3. Like the major types for integers, items of this major type do not carry content data; all the information is in the initial bytes (the head).

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	Reserved, not well-formed in the present document
31	"break" stop code for indefinite-length items (Section 3.2.1)

Table 3: 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 4 lists the numeric values assigned and available for simple values.

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

Table 4: 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, and that only the one-byte variants of these are well-formed; more generally speaking, each simple value only has a single representation variant).

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 numbers.)

3.4. Tagging of Items

In CBOR, a data item can be enclosed by a tag to give it some additional semantics, as uniquely identified by a "tag number". The tag is major type 6, its argument (Section 3) indicates the tag number, and it contains a single enclosed data item, the "tag content". (If a tag requires further structure to its content, this structure is provided by the enclosed data item.) We use the term "tag" for the entire data item consisting of both a tag number and the tag content: the tag content is the data item that is being tagged.

For example, assume that a byte string of length 12 is marked with a tag of number 2 to indicate it is a positive "bignum" (Section 3.4.3). The encoded data item would start with a byte 0b110_00010 (major type 6, additional information 2 for the tag number) followed by the encoded tag content: 0b010_01100 (major type 2, additional information of 12 for the length) followed by the 12 bytes of the bignum.

The definition of a tag number describes the additional semantics conveyed for tags with this tag number in the extended generic data model. These semantics may include equivalence of some tagged data items with other data items, including some that can already be represented in the basic generic data model. For instance, 0xc24101, a bignum the tag content of which is the byte string with the single byte 0x01, is equivalent to an integer 1, which could also be encoded for instance as 0x01, 0x1801, or 0x190001. The tag definition may include the definition of a preferred serialization (Section 4.1) that is recommended for generic encoders; this may prefer basic generic data model representations over ones that employ a tag.

The tag definition usually restricts what kinds of nested data item or items are valid for such tags. Tag definitions may restrict their content to a very specific syntactic structure, as the tags defined in this document do, or they may aim at a more semantically defined definition of their content, as for instance tags 40 and 1040 do [RFC8746]: These accept a number of different ways of representing arrays.

As a matter of convention, many tags do not accept null or undefined values as tag content; instead, the expectation is that a null or undefined value can be used in place of the entire tag; Section 3.4.2 provides some further considerations for one specific tag about the handling of this convention in application protocols and in mapping to platform types.

Decoders do not need to understand tags of every tag number, and 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 provide conversion hints when it is foreseen that the CBOR data item needs to be translated into a different format, requiring hints about the content of items. Understanding the semantics of tags is optional for a decoder; it can simply present both the tag number and the tag content to the application, without interpreting the additional semantics of the tag.

A tag applies semantics to the data item it encloses. Tags can nest: If tag A encloses tag B, which encloses data item C, tag A applies to the result of applying tag B on data item C.

IANA maintains a registry of tag numbers as described in Section 9.2. Table 5 provides a list of tag numbers that were defined in [RFC7049], with definitions in the rest of this section. (Tag number 35 was also defined in [RFC7049]; a discussion of this tag number follows in Section 3.4.5.3.) Note that many other tag numbers have been defined since the publication of [RFC7049]; see the registry described at Section 9.2 for the complete list.

Tag Number	Data Item	Tag Content Semantics
0	text string	Standard date/time string; see Section 3.4.1
1	integer or float	Epoch-based date/time; see Section 3.4.2
2	byte string	Positive bignum; see Section 3.4.3
3	byte string	Negative bignum; see Section 3.4.3
4	array	Decimal fraction; see Section 3.4.4
5	array	Bigfloat; see Section 3.4.4
21	(any)	Expected conversion to base64url encoding; see Section 3.4.5.2
22	(any)	Expected conversion to base64 encoding; see Section 3.4.5.2
23	(any)	Expected conversion to base16 encoding; see Section 3.4.5.2
24	byte string	Encoded CBOR data item; see Section 3.4.5.1
32	text string	URI; see Section 3.4.5.3
33	text string	base64url; see Section 3.4.5.3
34	text string	base64; see Section 3.4.5.3
36	text string	MIME message; see Section 3.4.5.3
55799	(any)	Self-described CBOR; see Section 3.4.6

Table 5: Tag numbers defined in RFC 7049

Conceptually, tags are interpreted in the generic data model, not at (de-)serialization time. A small number of tags (at this time, tag number 25 and tag number 29 [IANA.cbor-tags]) have been registered with semantics that may require processing at (de-)serialization time: The decoder needs to be aware and the encoder needs to be in control of the exact sequence in which data items are encoded into the CBOR data item. This means these tags cannot be implemented on top of an arbitrary generic CBOR encoder/decoder (which might not reflect the serialization order for entries in a map at the data model level and vice versa); their implementation therefore typically needs to be integrated into the generic encoder/decoder. The definition of new tags with this property is NOT RECOMMENDED.

IANA allocated tag numbers 65535, 4294967295, and 18446744073709551615 (binary all-ones in 16-bit, 32-bit, and 64-bit). These can be used as a convenience for implementers that want a single integer data structure to indicate either that a specific tag is present, or the absence of a tag. That allocation is described in Section 10 of [I-D.bormann-cbor-notable-tags]. These tags are not intended to occur in actual CBOR data items; implementations MAY flag such an occurrence as an error.

Protocols using tag numbers 0 and 1 extend the generic data model (Section 2) with data items representing points in time; tag numbers 2 and 3, with arbitrarily sized integers; and tag numbers 4 and 5, with floating-point values of arbitrary size and precision.

3.4.1. Standard Date/Time String

Tag number 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 a text string that doesn't match the [RFC4287] format is invalid.

3.4.2. Epoch-based Date/Time

Tag number 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 tag content 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".) 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 tag content.

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 number 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 number 1 support may restrict the tag content to be an integer (or a floating-point value) only.

Note that platform types for date/time may include null or undefined values, which may also be desirable at an application protocol level. While emitting tag number 1 values with non-finite tag content values (e.g., with NaN for undefined date/time values or with Infinite for an expiry date that is not set) may seem an obvious way to handle this, using untagged null or undefined avoids the use of non-finites and results in a shorter encoding. Application protocol designers are encouraged to consider these cases and include clear guidelines for handling them.

3.4.3. Bignums

Protocols using tag numbers 2 and 3 extend the generic data model (Section 2) with "bignums" representing arbitrarily sized integers. In the basic generic data model, bignum values are not equal to integers from the same model, but the extended generic data model created by this tag definition defines equivalence based on numeric value, and preferred serialization (Section 4.1) 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 number 2, the value of the bignum is n. For tag number 3, the value of the bignum is -1 - n. The preferred serialization of the byte string is to leave out any leading zeroes (note that this means the preferred serialization 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 serialization 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 serialization). 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 number 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
```

3.4.4. Decimal Fractions and Bigfloats

Protocols using tag number 4 extend the generic data model with data items representing arbitrary-length decimal fractions of the form m*(10**e). Protocols using tag number 5 extend the generic data model with data items representing arbitrary-length binary fractions of the form m*(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 representations.

"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 number 4) use base-10 exponents; the value of a decimal fraction data item is m*(10**e). Bigfloats (tag number 5) use base-2 exponents; the value of a bigfloat data item is m*(2**e). The exponent e MUST be represented in an integer of major type 0 or 1, while the mantissa can also be a bignum (Section 3.4.3). 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 6 for tag, additional information 4 for the tag number), followed by 0b100_00010 (major type 4 for the array, additional information 2 for the length of the array), followed by 0b001_00001 (major type 1 for the first integer, additional information 1 for the value of -2), followed by 0b000_11001 (major type 0 for the second integer, additional information 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 6 for tag, additional information 5 for the tag number), followed by 0b100_00010 (major type 4 for the array, additional information 2 for the length of the array), followed by 0b001_00000 (major type 1 for the first integer, additional information 0 for the value of -1), followed by 0b000_00011 (major type 0 for the second integer, additional information 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.

3.4.5. 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.

3.4.5.1. 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 number 24 (CBOR data item) can be used to tag the embedded byte string as a single data item encoded in CBOR format. Contained items that aren't byte strings are invalid. A contained byte string is valid if it encodes a well-formed CBOR data item; validity checking of the decoded CBOR item is not required for tag validity (but could be offered by a generic decoder as a special option).

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

Tag numbers 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 numbers suggest conversions to three of the base data encodings defined in [RFC4648]. Tag number 21 suggests conversion to base64url encoding (Section 5 of RFC 4648), where padding is not used (see Section 3.2 of RFC 4648); that is, all trailing equals signs ("=") are removed from the encoded string. Tag number 22 suggests conversion to classical base64 encoding (Section 4 of RFC 4648), with padding as defined in RFC 4648. For both base64url and base64, padding bits are set to zero (see Section 3.5 of RFC 4648), and the conversion to alternate encoding is performed on the contents of the byte string (that is, without adding any line breaks, whitespace, or other additional characters). Tag number 23 suggests conversion to base16 (hex) encoding, with uppercase alphabetics (see Section 8 of RFC 4648). Note that, for all three tag numbers, the encoding of the empty byte string is the empty text string.

3.4.5.3. 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.

* Tag number 32 is for URIs, as defined in [RFC3986]. If the text string doesn't match the "URI-reference" production, the string is invalid.

* Tag numbers 33 and 34 are for base64url- and base64-encoded text strings, respectively, as defined in [RFC4648]. If any of:

  - the encoded text string contains non-alphabet characters or only 1 alphabet character in the last block of 4 (where alphabet is defined by Section 5 of [RFC4648] for tag number 33 and Section 4 of [RFC4648] for tag number 34), 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.

* Tag number 36 is for MIME messages (including all headers), as defined in [RFC2045]. A text string that isn't a valid MIME message is invalid. (For this tag, validity checking may be particularly onerous for a generic decoder and might therefore not be offered. Note that many MIME messages are general binary data and can therefore not be represented in a text string; [IANA.cbor-tags] lists a registration for tag number 257 that is similar to tag number 36 but uses a byte string as its tag content.)

Note that tag numbers 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.

[RFC7049] also defined a tag number 35, for regular expressions that are in Perl Compatible Regular Expressions (PCRE/PCRE2) form [PCRE] or in JavaScript regular expression syntax [ECMA262]. The state of the art in these regular expression specifications has since advanced and is continually advancing, so the present specification does not attempt to update the references to a snapshot that is current at the

time of writing. Instead, this tag remains available (as registered in [RFC7049]) for applications that specify the particular regular expression variant they use out-of-band (possibly by limiting the usage to a defined common subset of both PCRE and ECMA262). As the present specification clarifies tag validity beyond [RFC7049], we note that due to the open way the tag was defined in [RFC7049], any contained string value needs to be valid at the CBOR tag level (but may then not be "expected" at the application level).

3.4.6. 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 number 55799 is defined for this purpose, specifically for use at the start of a stored encoded CBOR data item as specified by an application. It does not impart any special semantics on the data item that it encloses; that is, the semantics of the tag content enclosed in tag number 55799 is exactly identical to the semantics of the tag content itself.

The serialization of this tag's head 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 number 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 (preferred encoding); 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 (one billion) 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 therefore 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.

The preferred serialization 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. 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).

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:

* Preferred serialization MUST be used. In particular, this means that arguments (see Section 3) for integers, lengths in major types 2 through 5, and tags MUST be as short as possible, for instance:

  - 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.

  Floating-point values also MUST use the shortest form that preserves the value, e.g. 1.5 is encoded as 0xf93e00 (binary16) and 1000000.5 as 0xfa49742408 (binary32). (One implementation of this is 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 form and repeat the process with a test conversion to a 16-bit float. This also works to select 16-bit float for positive and negative Infinity as well.)

* Indefinite-length items MUST NOT appear. They can be encoded as definite-length items instead.

* The keys in every map MUST be sorted in the bytewise 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.

(Implementation note: the self-delimiting nature of the CBOR encoding means that there are no two well-formed CBOR encoded data items where one is a prefix of the other. The bytewise lexicographic comparison of deterministic encodings of different map keys therefore always ends in a position where the byte differs between the keys, before the end of a key is reached.)

4.2.2. Additional Deterministic Encoding Considerations

CBOR tags present additional considerations for deterministic encoding. If a CBOR-based protocol were to provide the same semantics for the presence and absence of a specific tag (e.g., by allowing both tag 1 data items and raw numbers in a date/time position, treating the latter as if they were tagged), the deterministic format would not allow the presence of the tag, based on the "shortest form" principle. For example, a protocol might give encoders the choice of representing a URL as either a text string or, using Section 3.4.5.3, tag number 32 containing a text string. This protocol's deterministic encoding needs to either require that the tag is present or require that it is absent, not allow either one.

In a protocol that does require tags in certain places to obtain specific semantics, the tag needs to appear in the deterministic format as well. Deterministic encoding considerations also apply to the content of tags.

If a protocol includes a field that can express integers with an absolute value of 2^64 or larger using tag numbers 2 or 3 (Section 3.4.3), the protocol's deterministic encoding needs to specify whether smaller integers are also expressed using these tags or using major types 0 and 1. Preferred serialization uses the latter choice, which is therefore recommended.

Protocols that include floating-point values, whether represented using basic floating-point values (Section 3.3) or using tags (or both), may need to define extra requirements on their deterministic encodings, such as:

* Although IEEE floating-point values can represent both positive and negative zero as distinct values, the application might not distinguish these and might decide to represent all zero values with a positive sign, disallowing negative zero. (The application may also want to restrict the precision of floating-point values in such a way that there is never a need to represent 64-bit -- or even 32-bit -- floating-point values.)

* If a protocol includes a field that can express floating-point values, with a specific data model that declares integer and floating-point values to be interchangeable, the protocol's deterministic encoding needs to specify whether (for example) the integer 1.0 is encoded as 0x01 (unsigned integer), 0xf93c00 (binary16), 0xfa3f800000 (binary32), or 0xfb3ff0000000000000 (binary64). Example 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 preferred (smallest of 16-, 32-, or 64-bit) floating-point representation that accurately represents the value,

  2. Encode all values as the preferred floating-point representation that accurately represents the value, even for integral values, or

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

  Rule 1 straddles the boundaries between integers and floating-point values, and Rule 3 does not use preferred serialization, so Rule 2 may be a good choice in many cases.

* If NaN is an allowed value and there is no intent to support NaN payloads or signaling NaNs, the protocol needs to pick a single representation, typically 0xf97e00. If that simple choice is not possible, specific attention will be needed for NaN handling.

* Subnormal numbers (nonzero numbers with the lowest possible exponent of a given IEEE 754 number format) may be flushed to zero outputs or be treated as zero inputs in some floating-point implementations. A protocol's deterministic encoding may want to specifically accommodate such implementations while creating an onus on other implementations, by excluding subnormal numbers from interchange, interchanging zero instead.

* The same number can be represented by different decimal fractions, by different bigfloats, and by different forms under other tags that may be defined to express numeric values. Depending on the implementation, it may not always be practical to determine whether any of these forms (or forms in the basic generic data model) are equivalent. An application protocol that presents choices of this kind for the representation format of numbers needs to be explicit in how the formats are to be chosen for deterministic encoding.

4.2.3. Length-first Map Key Ordering

The core deterministic encoding requirements (Section 4.2.1) 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 interchangeable, 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 a 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.

The rest of 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. Such an application also MUST have a matching streaming security mechanism, where the desired protection is available for incremental data presented to the application.

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 encoded CBOR data items and present the data items to an application. See Appendix C. (The diagnostic notation, Section 8, may be used to present well-formed CBOR values to humans.)

Generic CBOR encoders provide an application interface that allows the application to specify any well-formed value to be encoded as a CBOR data item, including simple values and tags unknown to the encoder.

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 (because [RFC3629] requires always using the shortest form) and so is not a valid CBOR item. Also, specific tags may make semantic constraints that may be violated, for instance by a bignum tag enclosing another tag, or by an instance of tag number 0 containing a byte string, or containing 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).

5.3. Validity of Items

A well-formed but invalid CBOR data item (Section 1.2) 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 any validity errors 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 occur at the basic validity level of CBOR or in the context of tags (tag validity).

5.3.1. Basic validity

Two kinds of validity errors can occur in the basic generic data model:

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. See Section 5.6 for more discussion of keys in maps.

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.

5.3.2. Tag validity

Two additional kinds of validity errors are introduced by adding tags to the basic generic data model:

Inadmissible type for tag content: Tag numbers (Section 3.4) specify

what type of data item is supposed to be used as their tag content; for example, the tag numbers 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.

Inadmissible value for tag content: The type of data item may be admissible for a tag's content, but the specific value may not be; e.g., a value of "yesterday" is not acceptable for the content of tag 0, even though it properly is a text string. A decoder that normally ingests such tags into equivalent platform types might present this tag to the application in a similar way to how it would present a tag with an unknown tag number (Section 5.4).

5.4. Validity and Evolution

A decoder with validity checking will expend the effort to reliably detect data items with validity errors. For example, such a 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 validity errors listed in the previous subsection.

The set of tags defined in the tag registry (Section 9.2), as well as the set of simple values defined in the simple values registry (Section 9.1), can grow at any time beyond the set understood by a generic decoder. A validity-checking decoder can do one of two things when it encounters such a case that it does not recognize:

* It can report an error (and not return data). Note that treating this case as an error can cause ossification, and is thus not encouraged. This error is not a validity error per se. This kind of error is more likely to be raised by a decoder that would be performing validity checking if this were a known case.

* 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 number or simple value.

The latter approach, which is also appropriate for decoders that do not support validity checking, provides 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 the processing needed for validity checking may have an appreciable cost (in particular with duplicate detection for maps), support of validity checking 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 valid CBOR results from the encoder. A generic encoder may also want to provide a validity-checking mode where it reliably limits its output to valid CBOR, independent of whether or not its application is indeed providing API-conformant data.

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 basic JavaScript number system treats all numbers as floating-point values, which may result in silent loss of precision in decoding integers with more than 53 significant bits. Another example is that, since CBOR keeps the sign bit for its integer representation in the major type, it has one bit more for signed numbers of a certain length (e.g., -2**64..2**64-1 for 1+8-byte integers) than the typical platform signed integer representation of the same length (-2**63..2**63-1 for 8-byte int64_t). 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 that does not require deterministic encoding 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. Similar considerations apply to floating-point values; decoding both preferred serializations and longer-than-needed ones is recommended.

CBOR-based protocols for constrained applications that provide 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), might express a quality-of-implementation expectation that the integer representation is used directly.

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, conversion is simplified by limiting keys to text strings only; otherwise, there has to be a specified mapping from the other CBOR types to text strings, 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. When processing maps that exhibit entries with duplicate keys, a generic decoder might do one of the following:

* Not accept maps with duplicate keys (that is, enforce validity for maps, see also Section 5.4). These generic decoders are universally useful. An application may still need to do perform its own duplicate checking based on application rules (for instance if the application equates integers and floating-point values in map key positions for specific maps).

* Pass all map entries to the application, including ones with duplicate keys. This requires the application to handle (check against) duplicate keys, even if the application rules are identical to the generic data model rules.

* Lose some entries with duplicate keys, e.g. by only delivering the final (or first) entry out of the entries with the same key. With such a generic decoder, applications may get different results for a specific key on different runs and with different generic decoders as which value is returned is based on generic decoder implementation and the actual order of keys in the map. In particular, applications cannot validate key uniqueness on their own as they do not necessarily see all entries; they may not be able to use such a generic decoder if they do need to validate key uniqueness. These generic decoders can only be used in situations where the data source and transfer can be relied upon to always provide valid maps; this is not possible if the data source and transfer can be attacked.

Generic decoders need to document which of these three approaches they implement.

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 their own for a protocol to require a deterministic encoding format.)

Applications for constrained devices that have maps where a small number of frequently used keys can be identified should consider using small integers as keys; for instance, a set of 24 or fewer frequent keys can be encoded in a single byte as unsigned integers, up to 48 if negative integers are also used. Less frequently occurring keys can then use integers with longer encodings.

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 number.

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 number and the tag content are equal. (Note that a generic decoder that provides processing for a specific tag may not be able to distinguish some semantically equivalent values, e.g. if leading zeroes occur in the content of tag 2/3 (Section 3.4.3).) 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 are not able to 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.

6. Converting Data between CBOR and JSON

This section gives non-normative advice about converting between CBOR and JSON. Implementations of converters MAY 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.

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

* 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.

* 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.

* An array (major type 4) becomes a JSON array.

* 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.

* False (major type 7, additional information 20) becomes a JSON false.

* True (major type 7, additional information 21) becomes a JSON true.

* Null (major type 7, additional information 22) becomes a JSON null.

* 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.

* Any other simple value (major type 7, any additional information value not yet discussed) is represented by the substitute value.

* A bignum (major type 6, tag number 2 or 3) is represented by encoding its byte string in base64url without padding and becomes a JSON string. For tag number 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.)

* A byte string with an encoding hint (major type 6, tag number 21 through 23) is encoded as described by the hint and becomes a JSON string.

* For all other tags (major type 6, any other tag number), the tag content is represented as a JSON value; the tag number is ignored.

* Indefinite-length items are made definite before conversion.

A CBOR-to-JSON converter may want to keep to the JSON profile I-JSON [RFC7493], to maximize interoperability and increase confidence that the JSON output can be processed with predictable results. For example, this has implications on the range of integers that can be represented reliably, as well as on the top-level items that may be supported by older JSON implementations.

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:

* JSON numbers without fractional parts (integer numbers) are represented as integers (major types 0 and 1, possibly major type 6 tag number 2 and 3), choosing the shortest form; integers longer than an implementation-defined threshold may instead be represented as floating-point values. The default range that is represented as integer is -2**53+1..2**53-1 (fully exploiting the range for exact integers in the binary64 representation often used for decoding JSON [RFC7493]). A CBOR-based protocol, or a generic converter implementation, may choose -2**32..2**32-1 or -2**64..2**64-1 (fully using the integer ranges available in CBOR with uint32_t or uint64_t, respectively) or even -2**31..2**31-1 or -2**63..2**63-1 (using popular ranges for two's complement signed integers). (If the JSON was generated from a JavaScript implementation, its precision is already limited to 53 bits maximum.)

* Numbers with fractional parts are represented as floating-point values, performing the decimal-to-binary conversion based on the precision provided by IEEE 754 binary64. The mathematical value of the JSON number is converted to binary64 using the roundTiesToEven procedure in Section 4.3.1 of [IEEE754]. Then, when encoding in CBOR, the preferred serialization uses the shortest floating-point representation exactly representing this conversion result; 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). Instead of using the default binary64 precision, there may be an implementation-defined limit to the precision of the conversion that will affect the precision of the represented values. Decimal representation should only be used on the CBOR side 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. Protocol designs should attempt 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:

* 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 easily 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.

* the "tag" space (values in major type 6). The total codepoint space is abundant; only a tiny part of it has been allocated. However, not all of these codepoints are equally efficient: the first 24 only consume a single ("1+0") byte, and half of them have already been allocated. The next 232 values only consume two ("1+1") bytes, with nearly a quarter already allocated. These subspaces need some curation to last for a few more decades. Implementations receiving an unknown tag number can choose to process just the enclosed tag content or, preferably, to process the tag as an unknown tag number wrapping the tag content. The IANA registry in Section 9.2 is the appropriate way to address the extensibility of this codepoint space.

* the "additional information" space. An implementation receiving an unknown additional information value has no way to continue decoding, so allocating codepoints in this space is a major step beyond just exercising an extension point. There are also very few codepoints left. See also Section 7.2.

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 >= 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 the present specification.

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 tag is written as an integer number for the tag number, followed by the tag content 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.

A number of useful extensions to the diagnostic notation defined here are provided in Appendix G of [RFC8610], "Extended Diagnostic Notation" (EDN). Similarly, an extension of this notation could be provided in a separate document to provide for the documentation of NaN payloads, which are not covered in the present document.

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. For example, "_" or "_3". 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.)

The detailed chunk structure of byte and text strings of indefinite length can be notated in the form (_ h'0123', h'4567') and (_ "foo", "bar"). However, for an indefinite length string with no chunks inside, (_) would be ambiguous whether a byte string (0x5fff) or a text string (0x7fff) is meant and is therefore not used. The basic forms ''_ and ""_ can be used instead and are reserved for the case with no chunks only -- not as short forms for the (permitted, but not really useful) encodings with only empty chunks, which to preserve the chunk structure need to be notated as (_ ''), (_ ""), etc.

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 4.

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, and other tags have already been defined since then.

New entries in the range 0 to 23 ("1+0") are assigned by Standards Action. New entries in the ranges 24 to 255 ("1+1") and 256 to 32767 (lower half of "1+2") are assigned by Specification Required. New entries in the range 32768 to 18446744073709551615 (upper half of "1+2", "1+4", and "1+8") are assigned by First Come First Served. The template for registration requests is:

* Data item

* Semantics (short form)

In addition, First Come First Served requests should include:

* Point of contact

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

Applicants exercising the First Come First Served range and making a suggestion for a tag number that is not representable in 32 bits (i.e., larger than 4294967295) should be aware that this could reduce interoperability with implementations that do not support 64-bit numbers.

9.3. Media Type ("MIME Type")

The Internet media type [RFC6838] for a single encoded CBOR data item is application/cbor, as defined in [IANA.media-types]:

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: Many

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: IETF CBOR Working Group cbor@ietf.org (mailto:cbor@ietf.org) or IETF Applications and Real-Time Area art@ietf.org (mailto:art@ietf.org)

Intended usage: COMMON

Restrictions on usage: none

Author: IETF CBOR Working Group cbor@ietf.org (mailto:cbor@ietf.org)

Change controller: The IESG iesg@ietf.org (mailto:iesg@ietf.org)

9.4. CoAP Content-Format

The CoAP Content-Format for CBOR is registered in [IANA.core-parameters]:

Media Type: application/cbor

Encoding: -

Id: 60

Reference: [RFCthis]

9.5. The +cbor Structured Syntax Suffix Registration

The Structured Syntax Suffix [RFC6838] for media types based on a single encoded CBOR data item is +cbor, as defined in [IANA.media-type-structured-suffix]:

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: IETF CBOR Working Group cbor@ietf.org (mailto:cbor@ietf.org) or IETF Applications and Real-Time Area art@ietf.org (mailto:art@ietf.org)

Author/Change Controller: The IESG iesg@ietf.org
(mailto:iesg@ietf.org)

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 secure channel such as TLS, is encrypted or signed, or has come from some other source that is presumed trusted.

Section 4.1 gives examples of limitations in interoperability when using a constrained CBOR decoder with input from a CBOR encoder that uses a non-preferred serialization. When a single data item is consumed both by such a constrained decoder and a full decoder, it can lead to security issues that can be exploited by an attacker who can inject or manipulate content.

As discussed throughout this document, there are many values that can be considered "equivalent" in some circumstances and "not equivalent" in others. As just one example, the numeric value for the number "one" might be expressed as an integer or a bignum. A system interpreting CBOR input might accept either form for the number "one", or might reject one (or both) forms. Such acceptance or rejection can have security implications in the program that is using the interpreted input.

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 even arbitrary precision numbers) 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.

The input check itself may consume resources. This is usually linear in the size of the input, which means that an attacker has to spend resources that are commensurate to the resources spent by the defender on input validation. However, an attacker might be able to craft inputs that will take longer for a target decoder to process than for the attacker to produce. Processing for arbitrary-precision numbers may exceed linear effort. Also, some hash-table implementations that are used by decoders to build in-memory representations of maps can be attacked to spend quadratic effort, unless a secret key (see Section 7 of [SIPHASH_LNCS], also [SIPHASH_OPEN]) or some other mitigation is employed. Such superlinear efforts can be exploited by an attacker to exhaust resources at or before the input validator; they therefore need to be avoided in a CBOR decoder implementation. Note that tag number definitions and their implementations can add security considerations of this kind; this should then be discussed in the security considerations of the tag number definition.

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 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 should provide a validity checking mode of operation (Section 5.4). Note, however, that a generic decoder cannot know about all requirements that an application poses on its input data; it is therefore not relieving the application from performing its own input checking. Also, since the set of defined tag numbers evolves, the application may employ a tag number that is not yet supported for validity checking by the generic decoder it uses. Generic decoders therefore need to provide documentation which tag numbers they support and what validity checking they can provide for each of them as well as for basic CBOR validity (UTF-8 checking, duplicate map key checking).

Section 3.4.3 notes that using the non-preferred choice of a bignum representation instead of a basic integer for encoding a number is not intended to have application semantics, but it can have such semantics if an application receiving CBOR data is using a decoder in the basic generic data model. This disparity causes a security issue if the two sets of semantics differ. Thus, applications using CBOR need to specify the data model that they are using for each use of CBOR data.

It is common to convert CBOR data to other formats. In many cases, CBOR has more expressive types than other formats; this is particularly true for the common conversion to JSON. The loss of type information can cause security issues for the systems that are processing the less-expressive data.

Section 6.2 describes a possibly-common usage scenario of converting between CBOR and JSON that could allow an attack if the attcker knows that the application is performing the conversion.

Security considerations for the use of base16 and base64 from [RFC4648], and the use of UTF-8 from [RFC3629], are relevant to CBOR as well.

11. References

11.1. Normative References

[C] International Organization for Standardization, "Information technology Programming languages C", ISO/IEC 9899:2018, Fourth Edition, June 2018.

[Cplusplus17]
International Organization for Standardization, "Programming languages C++", ISO/IEC 14882:2017, Fifth Edition, December 2017.

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

[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, "Open Group Standard: Vol. 1: Base Definitions, Issue 7", Section 4.16 'Seconds Since the Epoch', IEEE Std 1003.1, 2018 Edition, 2018, <http://pubs.opengroup.org/onlinepubs/9699919799/basedefs/V1_chap04.html#tag_04_16>.

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/>.

[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>.

[I-D.bormann-cbor-notable-tags] Bormann, C., "Notable CBOR Tags", Work in Progress, Internet-Draft, draft-bormann-cbor-notable-tags-02, 25 June 2020, <http://www.ietf.org/internet-drafts/draft-bormann-cbor-notable-tags-02.txt>.

[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>.

[IANA.core-parameters] IANA, "Constrained RESTful Environments (CoRE) Parameters", <http://www.iana.org/assignments/core-parameters>.

[IANA.media-type-structured-suffix] IANA, "Structured Syntax Suffix Registry", <http://www.iana.org/assignments/media-type-structured-suffix>.

[IANA.media-types]
IANA, "Media Types",
<http://www.iana.org/assignments/media-types>.

[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>.

[RFC7493] Bray, T., Ed., "The I-JSON Message Format", RFC 7493,
DOI 10.17487/RFC7493, March 2015,
<https://www.rfc-editor.org/info/rfc7493>.

[RFC7991] Hoffman, P., "The "xml2rfc" Version 3 Vocabulary",
RFC 7991, DOI 10.17487/RFC7991, December 2016,
<https://www.rfc-editor.org/info/rfc7991>.

[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>.

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

[RFC8618] Dickinson, J., Hague, J., Dickinson, S., Manderson, T., and J. Bond, "Compacted-DNS (C-DNS): A Format for DNS Packet Capture", RFC 8618, DOI 10.17487/RFC8618, September 2019, <https://www.rfc-editor.org/info/rfc8618>.

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

[RFC8746] Bormann, C., Ed., "Concise Binary Object Representation (CBOR) Tags for Typed Arrays", RFC 8746, DOI 10.17487/RFC8746, February 2020, <https://www.rfc-editor.org/info/rfc8746>.

[SIPHASH_LNCS] Aumasson, J. and D. Bernstein, "SipHash: A Fast Short-Input PRF", Lecture Notes in Computer Science pp. 489-508, DOI 10.1007/978-3-642-34931-7_28, 2012, <https://doi.org/10.1007/978-3-642-34931-7_28>.

[SIPHASH_OPEN] Aumasson, J. and D.J. Bernstein, "SipHash: a fast short-input PRF", <https://131002.net/siphash/siphash.pdf>.

[YAML] Ben-Kiki, O., Evans, C., and I.d. 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 of Encoded CBOR Data Items

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.) 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'010000000000000000')).

Diagnostic	Encoded
0	0x00
1	0x01
10	0x0a
23	0x17
24	0x1818
25	0x1819
100	0x1864
1000	0x1903e8
1000000	0x1a000f4240
1000000000000	0x1b000000e8d4a51000
18446744073709551615	0x1bffffffffffffffff
18446744073709551616	0xc249010000000000000000
-18446744073709551616	0x3bffffffffffffffff
-18446744073709551617	0xc349010000000000000000
-1	0x20
-10	0x29
-100	0x3863
-1000	0x3903e7
0.0	0xf90000
-0.0	0xf98000
1.0	0xf93c00
1.1	0xfb3ff199999999999a
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	0xfbc010666666666666
Infinity	0xf97c00
NaN	0xf97e00
-Infinity	0xf9fc00
Infinity	0xfa7f800000
NaN	0xfa7fc00000
-Infinity	0xfaff800000
Infinity	0xfb7ff0000000000000
NaN	0xfb7ff8000000000000
-Infinity	0xfbfff0000000000000
false	0xf4
true	0xf5
null	0xf6
undefined	0xf7
simple(16)	0xf0
simple(255)	0xf8ff
0("2013-03-21T20:04:00Z")	0xc074323031332d30332d32315432303a

	30343a30305a
1(1363896240)	0xc11a514b67b0
1(1363896240.5)	0xc1fb41d452d9ec200000
23(h'01020304')	0xd74401020304
24(h'6449455446')	0xd818456449455446
32("http://www.example.com")	0xd82076687474703a2f2f7777772e6578 616d706c652e636f6d
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"}	0xa56161614161626142616361436164614461656145
(_ 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]	0x9f0102030405060708090a0b0c0d0e0f101112131415161718181819ff
{_ "a": 1, "b": [_ 2, 3]}	0xbf61610161629f0203ffff
["a", {_ "b": "c"}]	0x826161bf61626163ff
{_ "Fun": true, "Amt": -2}	0xbf6346756ef563416d7421ff

Table 6: Examples of Encoded CBOR Data Items

Appendix B. Jump Table for Initial Byte

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	Unsigned 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 3.4.1)
0xc1	Epoch-based date/time (data item follows; see Section 3.4.2)
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 3.4.4)
0xc5	Bigfloat (data item "array" follows; see Section 3.4.4)
0xc6..0xd4	(tag)
0xd5..0xd7	Expected Conversion (data item follows; see Section 3.4.5.2)
0xd8..0xdb	(more tags; 1/2/4/8 bytes of tag number 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 7: 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:

* the pseudocode does not "fail";

* after execution of the pseudocode, no bytes are left in the input (except in streaming applications)

The pseudocode has the following prerequisites:

* 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.

* uint() converts a byte string into an unsigned integer by interpreting the byte string in network byte order.

* Arithmetic works as in C.

* All variables are unsigned integers of sufficient range.

Note that "well_formed" returns the major type for well-formed definite length items, but 99 for an indefinite length item (or -1 for a "break" stop code, only if "breakable" is set). This is used in "well_formed_indefinite" to ascertain that indefinite length strings only contain definite length strings as chunks.

```
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;                    // definite-length data item
}

well_formed_indefinite(mt, breakable) {
  switch (mt) {
    case 2: case 3:
      while ((it = well_formed(true)) != -1)
        if (it != mt)           // need definite-length chunk
          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 99;                    // indefinite-length data item
}
```

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 ~n (bitwise complement) in C unsigned arithmetic; ~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
  unsigned mt = ui & 0x20; // extract (shifted) 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

See Section 1.2 for some specific assumptions about the profile of the C language used in these pieces of code.

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) {
  unsigned half = (halfp[0] << 8) + halfp[1];
  unsigned exp = (half >> 10) & 0x1f;
  unsigned 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

A discussion of CBOR and other formats with respect to a different set of design objectives is provided in Section 5 and Appendix C of [RFC8618].

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 item, 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, which prevents 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 8 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 8: Examples for Different Levels of Conciseness

Appendix F. Well-formedness errors and examples

There are three basic kinds of well-formedness errors that can occur in decoding a CBOR data item:

* Too much data: There are input bytes left that were not consumed. This is only an error if the application assumed that the input bytes would span exactly one data item. Where the application uses the self-delimiting nature of CBOR encoding to permit additional data after the data item, as is for example done in CBOR sequences [RFC8742], the CBOR decoder can simply indicate what part of the input has not been consumed.

* Too little data: The input data available would need additional bytes added at their end for a complete CBOR data item. This may indicate the input is truncated; it is also a common error when trying to decode random data as CBOR. For some applications, however, this may not actually be an error, as the application may not be certain it has all the data yet and can obtain or wait for additional input bytes. Some of these applications may have an upper limit for how much additional data can show up; here the decoder may be able to indicate that the encoded CBOR data item cannot be completed within this limit.

* Syntax error: The input data are not consistent with the requirements of the CBOR encoding, and this cannot be remedied by adding (or removing) data at the end.

In Appendix C, errors of the first kind are addressed in the first paragraph/bullet list (requiring "no bytes are left"), and errors of the second kind are addressed in the second paragraph/bullet list (failing "if n bytes are no longer available"). Errors of the third kind are identified in the pseudocode by specific instances of calling fail(), in order:

* a reserved value is used for additional information (28, 29, 30)

* major type 7, additional information 24, value < 32 (incorrect)

* incorrect substructure of indefinite length byte/text string (may only contain definite length strings of the same major type)

* "break" stop code (mt=7, ai=31) occurs in a value position of a map or except at a position directly in an indefinite length item where also another enclosed data item could occur

* additional information 31 used with major type 0, 1, or 6

F.1. Examples for CBOR data items that are not well-formed

This subsection shows a few examples for CBOR data items that are not well-formed. Each example is a sequence of bytes each shown in hexadecimal; multiple examples in a list are separated by commas.

Examples for well-formedness error kind 1 (too much data) can easily be formed by adding data to a well-formed encoded CBOR data item.

Similarly, examples for well-formedness error kind 2 (too little data) can be formed by truncating a well-formed encoded CBOR data item. In test suites, it may be beneficial to specifically test with incomplete data items that would require large amounts of addition to be completed (for instance by starting the encoding of a string of a very large size).

A premature end of the input can occur in a head or within the enclosed data, which may be bare strings or enclosed data items that are either counted or should have been ended by a "break" stop code.

* End of input in a head: 18, 19, 1a, 1b, 19 01, 1a 01 02, 1b 01 02 03 04 05 06 07, 38, 58, 78, 98, 9a 01 ff 00, b8, d8, f8, f9 00, fa 00 00, fb 00 00 00

* Definite length strings with short data: 41, 61, 5a ff ff ff ff 00, 5b ff ff ff ff ff ff ff ff 01 02 03, 7a ff ff ff ff 00, 7b 7f ff ff ff ff ff ff ff 01 02 03

* Definite length maps and arrays not closed with enough items: 81, 81 81 81 81 81 81 81 81 81, 82 00, a1, a2 01 02, a1 00, a2 00 00 00

* Tag number not followed by tag content: c0

* Indefinite length strings not closed by a "break" stop code: 5f 41 00, 7f 61 00

* Indefinite length maps and arrays not closed by a "break" stop code: 9f, 9f 01 02, bf, bf 01 02 01 02, 81 9f, 9f 80 00, 9f 9f 9f 9f 9f ff ff ff ff, 9f 81 9f 81 9f 9f ff ff ff

A few examples for the five subkinds of well-formedness error kind 3 (syntax error) are shown below.

Subkind 1:

* Reserved additional information values: 1c, 1d, 1e, 3c, 3d, 3e, 5c, 5d, 5e, 7c, 7d, 7e, 9c, 9d, 9e, bc, bd, be, dc, dd, de, fc, fd, fe,

Subkind 2:

* Reserved two-byte encodings of simple values: f8 00, f8 01, f8 18, f8 1f

Subkind 3:

* Indefinite length string chunks not of the correct type: 5f 00 ff, 5f 21 ff, 5f 61 00 ff, 5f 80 ff, 5f a0 ff, 5f c0 00 ff, 5f e0 ff, 7f 41 00 ff

* Indefinite length string chunks not definite length: 5f 5f 41 00 ff ff, 7f 7f 61 00 ff ff

Subkind 4:

* Break occurring on its own outside of an indefinite length item: ff

* Break occurring in a definite length array or map or a tag: 81 ff, 82 00 ff, a1 ff, a1 ff 00, a1 00 ff, a2 00 00 ff, 9f 81 ff, 9f 82 9f 81 9f 9f ff ff ff ff

* Break in indefinite length map would lead to odd number of items (break in a value position): bf 00 ff, bf 00 00 00 ff

Subkind 5:

* Major type 0, 1, 6 with additional information 31: 1f, 3f, df

Appendix G. Changes from RFC 7049

As discussed in the introduction, this document is a revised edition of RFC 7049, with editorial improvements, added detail, and fixed errata. This document formally obsoletes RFC 7049, while keeping full compatibility of the interchange format from RFC 7049. This document does not create a new version of the format.

G.1. Errata processing, clerical changes

The two verified errata on RFC 7049, EID 3764 and EID 3770, concerned two encoding examples in the text that have been corrected (Section 3.4.3: "29" -> "49", Section 5.5: "0b000_11101" -> "0b000_11001"). Also, RFC 7049 contained an example using the numeric value 24 for a simple value (EID 5917), which is not well-formed; this example has been removed. Errata report 5763 pointed to an accident in the wording of the definition of tags; this was resolved during a re-write of Section 3.4. Errata report 5434 pointed out that the UBJSON example in Appendix E no longer complied with the version of UBJSON current at the time of submitting the report. It turned out that the UBJSON specification had completely changed since 2013; this example therefore also was removed. Further errata reports (4409, 4963, 4964) complained that the map key sorting rules for canonical encoding were onerous; these led to a reconsideration of the canonical encoding suggestions and replacement by the deterministic encoding suggestions (described below). An editorial suggestion in errata report 4294 was also implemented (improved symmetry by adding "Second value" to a comment to the last example in Section 3.2.2).

Other more clerical changes include:

* use of new RFCXML functionality [RFC7991];

* explain some more of the notation used;

* updated references, e.g. for RFC4627 to [RFC8259] in many places, for CNN-TERMS to [RFC7228]; added missing reference to [IEEE754] (importing required definitions) and updated to [ECMA262]; added a reference to [RFC8618] that further illustrates the discussion in Appendix E;

* the discussion of diagnostic notation mentions the "Extended Diagnostic Notation" (EDN) defined in [RFC8610] as well as the gap diagnostic notation has in representing NaN payloads; an explanation was added on how to represent indefinite length strings with no chunks;

* the addition of this appendix.

G.2. Changes in IANA considerations

The IANA considerations were generally updated (clerical changes, e.g., now pointing to the CBOR working group as the author of the specification). References to the respective IANA registries have been added to the informative references.

Tags in the space from 256 to 32767 (lower half of "1+2") are no longer assigned by First Come First Served; this range is now Specification Required.

G.3. Changes in suggestions and other informational components

In revising the document, beyond processing errata reports, the WG could use nearly seven years of experience with the use of CBOR in a diverse set of applications. This led to a number of editorial changes, including adding tables for illustration, but also to emphasizing some aspects and de-emphasizing others.

A significant addition in this revision is Section 2, which discusses the CBOR data model and its small variations involved in the processing of CBOR. Introducing terms for those (basic generic, extended generic, specific) enables more concise language in other places of the document, but also helps in clarifying expectations on implementations and on the extensibility features of the format.

RFC 7049, as a format derived from the JSON ecosystem, was influenced by the JSON number system that was in turn inherited from JavaScript at the time. JSON does not provide distinct integers and floating-point values (and the latter are decimal in the format). CBOR provides binary representations of numbers, which do differ between integers and floating-point values. Experience from implementation and use now suggested that the separation between these two number domains should be more clearly drawn in the document; language that suggested an integer could seamlessly stand in for a floating-point value was removed. Also, a suggestion (based on I-JSON [RFC7493]) was added for handling these types when converting JSON to CBOR, and the use of a specific rounding mechanism has been recommended.

For a single value in the data model, CBOR often provides multiple encoding options. The revision adds a new section Section 4, which first introduces the term "preferred serialization" (Section 4.1) and defines it for various kinds of data items. On the basis of this terminology, the section goes on to discuss how a CBOR-based protocol can define "deterministic encoding" (Section 4.2), which now avoids the RFC 7049 terms "canonical" and "canonicalization". The suggestion of "Core Deterministic Encoding Requirements" Section 4.2.1 enables generic support for such protocol-defined encoding requirements. The present revision further eases the implementation of deterministic encoding by simplifying the map ordering suggested in RFC 7049 to simple lexicographic ordering of encoded keys. A description of the older suggestion is kept as an alternative, now termed "length-first map key ordering" (Section 4.2.3).

The terminology for well-formed and valid data was sharpened and more stringently used, avoiding less well-defined alternative terms such as "syntax error", "decoding error" and "strict mode" outside examples. Also, a third level of requirements beyond CBOR-level validity that an application has on its input data is now explicitly called out. Well-formed (processable at all), valid (checked by a validity-checking generic decoder), and expected input (as checked by the application) are treated as a hierarchy of layers of acceptability.

The handling of non-well-formed simple values was clarified in text and pseudocode. Appendix F was added to discuss well-formedness errors and provide examples for them. The pseudocode was updated to be more portable and some portability considerations were added.

The discussion of validity has been sharpened in two areas. Map validity (handling of duplicate keys) was clarified and the domain of applicability of certain implementation choices explained. Also, while streamlining the terminology for tags, tag numbers, and tag content, discussion was added on tag validity, and the restrictions were clarified on tag content, in general and specifically for tag 1.

An implementation note (and note for future tag definitions) was added to Section 3.4 about defining tags with semantics that depend on serialization order.

Tag 35 is no longer defined in this updated document; the registration based on the definition in RFC 7049 remains in place.

Terminology was introduced in Section 3 for "argument" and "head", simplifying further discussion.

The security considerations were mostly rewritten and significantly expanded; in multiple other places, the document is now more explicit that a decoder cannot simply condone well-formedness errors.

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 Yasskin, Joe Hildebrand, Keith Moore, Laurence Lundblade, Matthew Lepinski, Michael Richardson, Nico Williams, Peter Occil, Phillip Hallam-Baker, Ray Polk, Stuart Cheshire, Tim Bray, Tony Finch, Tony Hansen, and Yaron Sheffer. Benjamin Kaduk provided an extensive review during IESG processing. Éric Vyncke, Erik Kline, Robert Wilton, and Roman Danyliw provided further IESG comments, which included an IoT directorate review by Eve Schooler.

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

Network Working Group
Internet-Draft
Intended status: Standards Track
Expires: August 30, 2018

H. Birkholz
Fraunhofer SIT
C. Vigano
Universitaet Bremen
C. Bormann
Universitaet Bremen TZI

February 26, 2018

Concise data definition language (CDDL): a notational convention to express CBOR data structures
draft-ietf-cbor-cddl-02

Abstract

This document proposes a notational convention to express CBOR data structures (RFC 7049). Its main goal is to provide an easy and unambiguous way to express structures for protocol messages and data formats that use CBOR.

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 August 30, 2018.

Copyright Notice

Table of Contents

1. Introduction

In this document, a notational convention to express CBOR [RFC7049] data structures is defined.

The main goal for the convention is to provide a unified notation that can be used when defining protocols that use CBOR. We term the convention "Concise data definition language", or CDDL.

The CBOR notational convention has the following goals:

(G1) Provide an unambiguous description of the overall structure of a CBOR data structure.

(G2) Flexibility to express the freedoms of choice in the CBOR data format.

(G3) Possibility to restrict format choices where appropriate [_format].

(G4) Able to express common CBOR datatypes and structures.

(G5) Human and machine readable and processable.

(G6) Automatic checking of data format compliance.

(G7) Extraction of specific elements from CBOR data for further processing.

Not an explicit goal per se, but a convenient side effect of the JSON generic data model being a subset of the CBOR generic data model, is the fact that CDDL can also be used for describing JSON data structures (see Appendix E.1).

This document has the following structure:

The syntax of CDDL is defined in Section 3. Examples of CDDL and related CBOR data items ("instances") are defined in Appendix H. Section 4 discusses usage of CDDL. Examples are provided early in the text to better illustrate concept definitions. A formal definition of CDDL using ABNF grammar is provided in Appendix B. Finally, a prelude of standard CDDL definitions available in every CBOR specification is listed in Appendix E.

1.1. Requirements notation

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 RFC 2119, BCP 14 [RFC2119].

1.2. Terminology

New terms are introduced in _cursive_. CDDL text in the running text is in "typewriter".

2. The Style of Data Structure Specification

CDDL focuses on styles of specification that are in use in the community employing the data model as pioneered by JSON and now refined in CBOR.

There are a number of more or less atomic elements of a CBOR data model, such as numbers, simple values (false, true, nil), text and byte strings; CDDL does not focus on specifying their structure. CDDL of course also allows adding a CBOR tag to a data item.

The more important components of a data structure definition language are the data types used for composition: arrays and maps in CBOR (called arrays and objects in JSON). While these are only two representation formats, they are used to specify four loosely distinguishable styles of composition:

- A _vector_, an array of elements that are mostly of the same semantics. The set of signatures associated with a signed data item is a typical application of a vector.

- A _record_, an array the elements of which have different, positionally defined semantics, as detailed in the data structure definition. A 2D point, specified as an array of an x coordinate (which comes first) and a y coordinate (coming second) is an example of a record, as is the pair of exponent (first) and mantissa (second) in a CBOR decimal fraction.

- A _table_, a map from a domain of map keys to a domain of map values, that are mostly of the same semantics. A set of language tags, each mapped to a text string translated to that specific language, is an example of a table. The key domain is usually not limited to a specific set by the specification, but open for the application, e.g., in a table mapping IP addresses to MAC addresses, the specification does not attempt to foresee all possible IP addresses.

- A _struct_, a map from a domain of map keys as defined by the specification to a domain of map values the semantics of each of which is bound to a specific map key. This is what many people have in mind when they think about JSON objects; CBOR adds the ability to use map keys that are not just text strings. Structs can be used to solve similar problems as records; the use of explicit map keys facilitates optionality and extensibility.

Two important concepts provide the foundation for CDDL:

1. Instead of defining all four types of composition in CDDL separately, or even defining one kind for arrays (vectors and records) and one kind for maps (tables and structs), there is only one kind of composition in CDDL: the _group_ (Section 2.1).

2. The other important concept is that of a _type_. The entire CDDL specification defines a type (the one defined by its first _rule_), which formally is the set of CBOR data items that are acceptable as "instances" for this specification. CDDL predefines a number of basic types such as "uint" (unsigned integer) or "tstr" (text string), often making use of a simple formal notation for CBOR data items. Each value that can be expressed as a CBOR data item also is a type in its own right, e.g. "1". A type can be built as a _choice_ of other types, e.g., an "int" is either a "uint" or a "nint" (negative integer). Finally, a type can be built as an array or a map from a group.

The rest of this section introduces a number of basic concepts of CDDL, and section Section 3 defines additional syntax. Appendix C gives a concise summary of the semantics of CDDL.

2.1. Groups and Composition in CDDL

CDDL Groups are lists of name/value pairs (group _entries_).

In an array context, only the value of the entry is represented; the name is annotation only (and can be left off if not needed). In a map context, the names become the map keys ("member keys").

In an array context, the sequence of elements in the group is important, as it is the information that allows associating actual array elements with entries in the group. In a map context, the sequence of entries in a group is not relevant (but there is still a need to write down group entries in a sequence).

A simple example of using a group right in a map definition is:

```
person = {
  age: int,
  name: tstr,
  employer: tstr,
}
```

Figure 1: Using a group in a map

The three entries of the group are written between the curly braces that create the map: Here, "age", "name", and "employer" are the names that turn into the map key text strings, and "int" and "tstr" (text string) are the types of the map values under these keys.

A group by itself (without creating a map around it) can be placed in (round) parentheses, and given a name by using it in a rule:

```
pii = (
  age: int,
  name: tstr,
  employer: tstr,
)
```

Figure 2: A basic group

This separate, named group definition allows us to rephrase Figure 1 as:

```
person = {
  pii
}
```

Figure 3: Using a group by name

Note that the (curly) braces signify the creation of a map; the groups themselves are neutral as to whether they will be used in a map or an array.

As shown in Figure 1, the parentheses for groups are optional when there is some other set of brackets present. Note that they can still be used, leading to the not so realistic, but perfectly valid example:

```
person = {(
  age: int,
  name: tstr,
  employer: tstr,
)}
```

Groups can be used to factor out common parts of structs, e.g., instead of writing copy/paste style specifications such as in Figure 4, one can factor out the common subgroup, choose a name for it, and write only the specific parts into the individual maps (Figure 5).

```
person = {
  age: int,
  name: tstr,
  employer: tstr,
}

dog = {
  age: int,
  name: tstr,
  leash-length: float,
}
```

Figure 4: Maps with copy/paste

```
person = {
  identity,
  employer: tstr,
}

dog = {
  identity,
  leash-length: float,
}

identity = (
  age: int,
  name: tstr,
)
```

Figure 5: Using a group for factorization

Note that the lists inside the braces in the above definitions constitute (anonymous) groups, while "identity" is a named group.

2.1.1. Usage

Groups are the instrument used in composing data structures with CDDL. It is a matter of style in defining those structures whether to define groups (anonymously) right in their contexts or whether to define them in a separate rule and to reference them with their respective name (possibly more than once).

With this, one is allowed to define all small parts of their data structures and compose bigger protocol units with those or to have only one big protocol data unit that has all definitions ad hoc where needed.

2.1.2. Syntax

The composition syntax intends to be concise and easy to read:

- The start of a group can be marked by '('
- The end of a group can be marked by ')'
- Definitions of entries inside of a group are noted as follows: _keytype => valuetype,_ (read "keytype maps to valuetype"). The comma is actually optional (not just in the final entry), but it is considered good style to set it. The double arrow can be replaced by a colon in the common case of directly using a text string or integer literal as a key (see Section 3.5.1).

An entry consists of a _keytype_ and a _valuetype_:

- _keytype_ is either an atom used as the actual key or a type in general. The latter case may be needed when using groups in a table context, where the actual keys are of lesser importance than the key types, e.g in contexts verifying incoming data.

- _valuetype_ is a type, which could be derived from the major types defined in [RFC7049], could be a convenience valuetype defined in this document (Appendix E) or the name of a type defined in the specification.

A group definition can also contain choices between groups, see Section 2.2.2.

2.2. Types

2.2.1. Values

Values such as numbers and strings can be used in place of a type. (For instance, this is a very common thing to do for a keytype, common enough that CDDL provides additional convenience syntax for this.)

2.2.2. Choices

Many places that allow a type also allow a choice between types, delimited by a "/" (slash). The entire choice construct can be put into parentheses if this is required to make the construction unambiguous (please see Appendix B for the details).

Choices of values can be used to express enumerations:

```
attire = "bow tie" / "necktie" / "Internet attire"
protocol = 6 / 17
```

Similarly as for types, CDDL also allows choices between groups, delimited by a "//" (double slash).

```
address = { delivery }

delivery = (
street: tstr, ? number: uint, city //
po-box: uint, city //
per-pickup: true )

city = (
name: tstr, zip-code: uint
)
```

Both for type choices and for group choices, additional alternatives can be added to a rule later in separate rules by using "/=" and "//=", respectively, instead of "=":

```
attire /= "swimwear"

delivery //= (
lat: float, long: float, drone-type: tstr
)
```

It is not an error if a name is first used with a "/=" or "//=" (there is no need to "create it" with "=").

2.2.2.1. Ranges

Instead of naming all the values that make up a choice, CDDL allows building a _range_ out of two values that are in an ordering relationship. A range can be inclusive of both ends given (denoted by joining two values by ".."), or include the first and exclude the second (denoted by instead using "...").

```
device-address = byte
max-byte = 255
byte = 0..max-byte ; inclusive range
first-non-byte = 256
byte1 = 0...first-non-byte ; byte1 is equivalent to byte
```

CDDL currently only allows ranges between numbers [_range].

2.2.2.2. Turning a group into a choice

Some choices are built out of large numbers of values, often integers, each of which is best given a semantic name in the specification. Instead of naming each of these integers and then accumulating these into a choice, CDDL allows building a choice from a group by prefixing it with a "&" character:

```
terminal-color = &basecolors
basecolors = (
  black: 0, red: 1,  green: 2,  yellow: 3,
  blue: 4,  magenta: 5,  cyan: 6,  white: 7,
)
extended-color = &(
  basecolors,
  orange: 8,  pink: 9,  purple: 10,  brown: 11,
)
```

As with the use of groups in arrays (Section 3.4), the membernames have only documentary value (in particular, they might be used by a tool when displaying integers that are taken from that choice).

2.2.3. Representation Types

CDDL allows the specification of a data item type by referring to the CBOR representation (major and minor numbers). How this is used should be evident from the prelude (Appendix E).

It may be necessary to make use of representation types outside the prelude, e.g., a specification could start by making use of an existing tag in a more specific way, or define a new tag not defined in the prelude:

```
my_breakfast = #6.55799(breakfast)   ; cbor-any is too general!
breakfast = cereal / porridge
cereal = #6.998(tstr)
porridge = #6.999([liquid, solid])
liquid = milk / water
milk = 0
water = 1
solid = tstr
```

2.2.4. Root type

There is no special syntax to identify the root of a CDDL data structure definition: that role is simply taken by the first rule defined in the file.

This is motivated by the usual top-down approach for defining data structures, decomposing a big data structure unit into smaller parts; however, except for the root type, there is no need to strictly follow this sequence.

(Note that there is no way to use a group as a root - it must be a type. Using a group as the root might be employed as a way to specify a CBOR sequence in a future version of this specification;

this would act as if that group is used in an array and the data items in that fictional array form the members of the CBOR sequence.)

3. Syntax

In this section, the overall syntax of CDDL is shown, alongside some examples just illustrating syntax. (The definition will not attempt to be overly formal; refer to Appendix B for the details.)

3.1. General conventions

The basic syntax is inspired by ABNF [RFC5234], with

- rules, whether they define groups or types, are defined with a name, followed by an equals sign "=" and the actual definition according to the respective syntactic rules of that definition.

- A name can consist of any of the characters from the set {'A', ..., 'Z', 'a', ..., 'z', '0', ..., '9', '_', '-', '@', '.', '$'}, starting with an alphabetic character (including '@', '_', '$') and ending in one or a digit.

  * Names are case sensitive.

  * It is preferred style to start a name with a lower case letter.

  * The hyphen is preferred over the underscore (except in a "bareword" (Section 3.5.1), where the semantics may actually require an underscore).

  * The period may be useful for larger specifications, to express some module structure (as in "tcp.throughput" vs. "udp.throughput").

  * A number of names are predefined in the CDDL prelude, as listed in Appendix E.

  * Rule names (types or groups) do not appear in the actual CBOR encoding, but names used as "barewords" in member keys do.

- Comments are started by a ';' (semicolon) character and finish at the end of a line (LF or CRLF).

- outside strings, whitespace (spaces, newlines, and comments) is used to separate syntactic elements for readability (and to separate identifiers or numbers that follow each other); it is otherwise completely optional.

- Hexadecimal numbers are preceded by '0x' (without quotes, lower case x), and are case insensitive. Similarly, binary numbers are preceded by '0b'.

- Text strings are enclosed by double quotation '"' characters. They follow the conventions for strings as defined in section 7 of [RFC8259]. (ABNF users may want to note that there is no support in CDDL for the concept of case insensitivity in text strings; if necessary, regular expressions can be used (Section 3.8.3).)

- Byte strings are enclosed by single quotation "'" characters and may be prefixed by "h" or "b64". If unprefixed, the string is interpreted as with a text string, except that single quotes must be escaped and that the UTF-8 bytes resulting are marked as a byte string (major type 2). If prefixed as "h" or "b64", the string is interpreted as a sequence of hex digits or a base64(url) string, respectively (as with the diagnostic notation in section 6 of [RFC7049]; cf. Appendix G.2); any white space present within the string (including comments) is ignored in the prefixed case. [_strings]

- CDDL uses UTF-8 [RFC3629] for its encoding.

Example:

```
; This is a comment
person = { g }

g = (
  "name": tstr,
  age: int,  ; "age" is a bareword
)
```

3.2. Occurrence

An optional _occurrence_ indicator can be given in front of a group entry. It is either one of the characters '?' (optional), '*' (zero or more), or '+' (one or more), or is of the form n*m, where n and m are optional unsigned integers and n is the lower limit (default 0) and m is the upper limit (default no limit) of occurrences.

If no occurrence indicator is specified, the group entry is to occur exactly once (as if 1*1 were specified).

Note that CDDL, outside any directives/annotations that could possibly be defined, does not make any prescription as to whether arrays or maps use the definite length or indefinite length encoding. I.e., there is no correlation between leaving the size of an array

"open" in the spec and the fact that it is then interchanged with definite or indefinite length.

Please also note that CDDL can describe flexibility that the data model of the target representation does not have. This is rather obvious for JSON, but also is relevant for CBOR:

```
apartment = {
  kitchen: size,
  * bedroom: size,
}
size = float ; in m2
```

The previous specification does not mean that CBOR is changed to allow to use the key "bedroom" more than once. In other words, due to the restrictions imposed by the data model, the third line pretty much turns into:

```
  ? bedroom: size,
```

(Occurrence indicators beyond one still are useful in maps for groups that allow a variety of keys.)

3.3. Predefined names for types

CDDL predefines a number of names. This subsection summarizes these names, but please see Appendix E for the exact definitions.

The following keywords for primitive datatypes are defined:

"bool" Boolean value (major type 7, additional information 20 or 21).

"uint" An unsigned integer (major type 0).

"nint" A negative integer (major type 1).

"int" An unsigned integer or a negative integer.

"float16" A number representable as an IEEE 754 half-precision float (major type 7, additional information 25).

"float32" A number representable as an IEEE 754 single-precision float (major type 7, additional information 26).

"float64" A number representable as an IEEE 754 double-precision float (major type 7, additional information 27).

"float" One of float16, float32, or float64.

"bstr" or "bytes" A byte string (major type 2).

"tstr" or "text" Text string (major type 3)

(Note that there are no predefined names for arrays or maps; these are defined with the syntax given below.)

In addition, a number of types are defined in the prelude that are associated with CBOR tags, such as "tdate", "bigint", "regexp" etc.

3.4. Arrays

Array definitions surround a group with square brackets.

For each entry, an occurrence indicator as specified in Section 3.2 is permitted.

For example:

```
unlimited-people = [* person]
one-or-two-people = [1*2 person]
at-least-two-people = [2* person]
person = (
    name: tstr,
    age: uint,
)
```

The group "person" is defined in such a way that repeating it in the array each time generates alternating names and ages, so these are four valid values for a data item of type "unlimited-people":

```
["roundlet", 1047, "psychurgy", 2204, "extrarhythmical", 2231]
[]
["aluminize", 212, "climograph", 4124]
["penintime", 1513, "endocarditis", 4084, "impermeator", 1669,
 "coextension", 865]
```

3.5. Maps

The syntax for specifying maps merits special attention, as well as a number of optimizations and conveniences, as it is likely to be the focal point of many specifications employing CDDL. While the syntax does not strictly distinguish struct and table usage of maps, it caters specifically to each of them.

But first, let's reiterate a feature of CBOR that it has inherited from JSON: The key/value pairs in CBOR maps have no fixed ordering. (One could imagine situations where fixing the ordering may be of use. For example, a decoder could look for values related with integer keys 1, 3 and 7. If the order were fixed and the decoder encounters the key 4 without having encountered key 3, it could conclude that key 3 is not available without doing more complicated bookkeeping. Unfortunately, neither JSON nor CBOR support this, so no attempt was made to support this in CDDL either.)

3.5.1. Structs

The "struct" usage of maps is similar to the way JSON objects are used in many JSON applications.

A map is defined in the same way as defining an array (see Section 3.4), except for using curly braces "{}" instead of square brackets "[]".

An occurrence indicator as specified in Section 3.2 is permitted for each group entry.

The following is an example of a structure:

```
Geography = [
  city           : tstr,
  gpsCoordinates : GpsCoordinates,
]

GpsCoordinates = {
  longitude      : uint,            ; multiplied by 10^7
  latitude       : uint,            ; multiplied by 10^7
}
```

When encoding, the Geography structure is encoded using a CBOR array with two entries (the keys for the group entries are ignored), whereas the GpsCoordinates are encoded as a CBOR map with two key/value pairs.

Types used in a structure can be defined in separate rules or just in place (potentially placed inside parentheses, such as for choices). E.g.:

```
located-samples = {
  sample-point: int,
  samples: [+ float],
}
```

where "located-samples" is the datatype to be used when referring to the struct, and "sample-point" and "samples" are the keys to be used. This is actually a complete example: an identifier that is followed by a colon can be directly used as the text string for a member key (we speak of a "bareword" member key), as can a double-quoted string or a number. (When other types, in particular multi-valued ones, are used as keytypes, they are followed by a double arrow, see below.)

If a text string key does not match the syntax for an identifier (or if the specifier just happens to prefer using double quotes), the text string syntax can also be used in the member key position, followed by a colon. The above example could therefore have been written with quoted strings in the member key positions. More generally, all the types defined can be used in a keytype position by following them with a double arrow. A string also is a (single-valued) type, so another form for this example is:

```
located-samples = {
  "sample-point" => int,
  "samples" => [+ float],
}
```

See Section 3.5.3 below for how the colon shortcut described here also adds some implied semantics.

A better way to demonstrate the double-arrow use may be:

```
located-samples = {
  sample-point: int,
  samples: [+ float],
  * equipment-type => equipment-tolerances,
}
equipment-type = [name: tstr, manufacturer: tstr]
equipment-tolerances = [+ [float, float]]
```

The example below defines a struct with optional entries: display name (as a text string), the name components first name and family name (as a map of text strings), and age information (as an unsigned integer).

```
PersonalData = {
  ? displayName: tstr,
  NameComponents,
  ? age: uint,
}

NameComponents = (
  ? firstName: tstr,
  ? familyName: tstr,
)
```

Note that the group definition for NameComponents does not generate another map; instead, all four keys are directly in the struct built by PersonalData.

In this example, all key/value pairs are optional from the perspective of CDDL. With no occurrence indicator, an entry is mandatory.

If the addition of more entries not specified by the current specification is desired, one can add this possibility explicitly:

```
PersonalData = {
  ? displayName: tstr,
  NameComponents,
  ? age: uint,
  * tstr => any
}

NameComponents = (
  ? firstName: tstr,
  ? familyName: tstr,
)
```

Figure 6: Personal Data: Example for extensibility

The cddl tool (Appendix F) generated as one acceptable instance for this specification:

```
{"familyName": "agust", "antiforeignism": "pretzel",
 "springbuck": "illuminatingly", "exuviae": "ephemeris",
 "kilometrage": "frogfish"}
```

(See Section 3.9 for one way to explicitly identify an extension point.)

3.5.2. Tables

A table can be specified by defining a map with entries where the keytype is not single-valued, e.g.:

```
square-roots = {* x => y}
x = int
y = float
```

Here, the key in each key/value pair has datatype x (defined as int), and the value has datatype y (defined as float).

If the specification does not need to restrict one of x or y (i.e., the application is free to choose per entry), it can be replaced by the predefined name "any".

As another example, the following could be used as a conversion table converting from an integer or float to a string:

```
tostring = {* mynumber => tstr}
mynumber = int / float
```

3.5.3. Cuts in Maps

The extensibility idiom discussed above for structs has one problem:

```
extensible-map-example = {
  ? "optional-key" => int,
  * tstr => any
}
```

In this example, there is one optional key "optional-key", which, when present, maps to an integer. There is also a wild card for any future additions.

Unfortunately, the data item

```
{ "optional-key": "nonsense" }
```

does match this specification: While the first entry of the group does not match, the second one (the wildcard) does. This may be very well desirable (e.g., if a future extension is to be allowed to extend the type of "optional-key"), but in many cases isn't.

In anticipation of a more general potential feature called "cuts", CDDL allows inserting a cut "^" into the definition of the map entry:

```
extensible-map-example = {
  ? "optional-key" ^ => int,
  * tstr => any
}
```

A cut in this position means that once the map key matches the entry carrying the cut, other potential matches for the key that occur in later entries in the group of the map are no longer allowed. (This rule applies independent of whether the value matches, too.) So the example above no longer matches the version modified with a cut.

Since the desire for this kind of exclusive matching is so frequent, the ":" shortcut is actually defined to include the cut semantics. So the preceding example (including the cut) can be written more simply as:

```
extensible-map-example = {
  ? "optional-key": int,
  * tstr => any
}
```

or even shorter, using a bareword for the key:

```
extensible-map-example = {
  ? optional-key: int,
  * tstr => any
}
```

3.6. Tags

A type can make use of a CBOR tag (major type 6) by using the representation type notation, giving #6.nnn(type) where nnn is an unsigned integer giving the tag number and "type" is the type of the data item being tagged.

For example, the following line from the CDDL prelude (Appendix E) defines "biguint" as a type name for a positive bignum N:

```
biguint = #6.2(bstr)
```

The tags defined by [RFC7049] are included in the prelude. Additional tags since registered need to be added to a CDDL specification as needed; e.g., a binary UUID tag could be referenced as "buuid" in a specification after defining

```
buuid = #6.37(bstr)
```

In the following example, usage of the tag 32 for URIs is optional:

```
my_uri = #6.32(tstr) / tstr
```

3.7. Unwrapping

The group that is used to define a map or an array can often be reused in the definition of another map or array. Similarly, a type defined as a tag carries an internal data item that one would like to refer to. In these cases, it is expedient to simply use the name of the map, array, or tag type as a handle for the group or type defined inside it.

The "unwrap" operator (written by preceding a name by a tilde character "~") can be used to strip the type defined for a name by one layer, exposing the underlying group (for maps and arrays) or type (for tags).

For example, an application might want to define a basic and an advanced header. Without unwrapping, this might be done as follows:

```
basic-header-group = (
  field1: int,
  field2: text,
)

basic-header = { basic-header-group }

advanced-header = {
  basic-header-group,
  field3: bytes,
  field4: number, ; as in the tagged type "time"
}
```

Unwrapping simplifies this to:

```
basic-header = {
  field1: int,
  field2: text,
}

advanced-header = {
  ~basic-header,
  field3: bytes,
  field4: ~time,
}
```

(Note that leaving out the first unwrap operator in the latter example would lead to nesting the basic-header in its own map inside the advanced-header, while, with the unwrapped basic-header, the

definition of the group inside basic-header is essentially repeated inside advanced-header, leading to a single map. This can be used for various applications often solved by inheritance in programming languages. The effect of unwrapping can also be described as "threading in" the group or type inside the referenced type, which suggested the thread-like "~" character.)

3.8. Controls

A _control_ allows to relate a _target_ type with a _controller_ type via a _control operator_.

The syntax for a control type is "target .control-operator controller", where control operators are special identifiers prefixed by a dot. (Note that _target_ or _controller_ might need to be parenthesized.)

A number of control operators are defined at his point. Note that the CDDL tool does not currently support combining multiple controls on a single target.

3.8.1. Control operator .size

A ".size" control controls the size of the target in bytes by the control type. Examples:

```
full-address = [[+ label], ip4, ip6]
ip4 = bstr .size 4
ip6 = bstr .size 16
label = bstr .size (1..63)
```

Figure 7: Control for size in bytes

When applied to an unsigned integer, the ".size" control restricts the range of that integer by giving a maximum number of bytes that should be needed in a computer representation of that unsigned integer. In other words, "uint .size N" is equivalent to "0...BYTES_N", where BYTES_N == 256**N.

```
audio_sample = uint .size 3 ; 24-bit, equivalent to 0..16777215
```

Figure 8: Control for integer size in bytes

Note that, as with value restrictions in CDDL, this control is not a representation constraint; a number that fits into fewer bytes can still be represented in that form, and an inefficient implementation could use a longer form (unless that is restricted by some format

constraints outside of CDDL, such as the rules in Section 3.9 of [RFC7049]).

3.8.2. Control operator .bits

A ".bits" control on a byte string indicates that, in the target, only the bits numbered by a number in the control type are allowed to be set. (Bits are counted the usual way, bit number "n" being set in "str" meaning that "(str[n >> 3] & (1 << (n & 7))) != 0".) [_bitsendian]

Similarly, a ".bits" control on an unsigned integer "i" indicates that for all unsigned integers "n" where "(i & (1 << n)) != 0", "n" must be in the control type.

```
tcpflagbytes = bstr .bits flags
flags = &(
  fin: 8,
  syn: 9,
  rst: 10,
  psh: 11,
  ack: 12,
  urg: 13,
  ece: 14,
  cwr: 15,
  ns: 0,
) / (4..7) ; data offset bits

rwxbits = uint .bits rwx
rwx = &(r: 2, w: 1, x: 0)
```

Figure 9: Control for what bits can be set

The CDDL tool generates the following ten example instances for "tcpflagbytes":

```
h'906d' h'01fc' h'8145' h'01b7' h'013d' h'409f' h'018e' h'c05f'
h'01fa' h'01fe'
```

These examples do not illustrate that the above CDDL specification does not explicitly specify a size of two bytes: A valid all clear instance of flag bytes could be "h''" or "h'00'" or even "h'000000'" as well.

3.8.3. Control operator .regexp

A ".regexp" control indicates that the text string given as a target needs to match the XSD regular expression given as a value in the control type. XSD regular expressions are defined in Appendix F of [W3C.REC-xmlschema-2-20041028].

```
nai = tstr .regexp "[A-Za-z0-9]+@[A-Za-z0-9]+(\\.[A-Za-z0-9]+)+"
```

Figure 10: Control with an XSD regexp

The CDDL tool proposes:

"N1@CH57HF.4Znqe0.dYJRN.igjf"

3.8.3.1. Usage considerations

Note that XSD regular expressions do not support the usual \x or \u escapes for hexadecimal expression of bytes or unicode code points. However, in CDDL the XSD regular expressions are contained in text strings, the literal notation for which provides \u escapes; this should suffice for most applications that use regular expressions for text strings. (Note that this also means that there is one level of string escaping before the XSD escaping rules are applied.)

XSD regular expressions support character class subtraction, a feature often not found in regular expression libraries; specification writers may want to use this feature sparingly. Similar considerations apply to Unicode character classes; where these are used, the specification SHOULD identify which Unicode versions are addressed.

Other surprises for infrequent users of XSD regular expressions may include:

- No direct support for case insensitivity. While case insensitivity has gone mostly out of fashion in protocol design, it is sometimes needed and then needs to be expressed manually as in "[Cc][Aa][Ss][Ee]".

- The support for popular character classes such as \w and \d is based on Unicode character properties, which is often not what is desired in an ASCII-based protocol and thus might lead to surprises. (\s and \S do have their more conventional meanings, and "." matches any character but the line ending characters \r or \n.)

3.8.3.2. Discussion

There are many flavors of regular expression in use in the programming community. For instance, perl-compatible regular expressions (PCRE) are widely used and probably are more useful than XSD regular expressions. However, there is no normative reference for PCRE that could be used in the present document. Instead, we opt for XSD regular expressions for now. There is precedent for that choice in the IETF, e.g., in YANG [RFC7950].

Note that CDDL uses controls as its main extension point. This creates the opportunity to add further regular expression formats in addition to the one referenced here if desired. As an example, a control ".pcre" is defined in [I-D.bormann-cbor-cddl-freezer].

3.8.4. Control operators .cbor and .cborseq

A ".cbor" control on a byte string indicates that the byte string carries a CBOR encoded data item. Decoded, the data item matches the type given as the right-hand side argument (type1 in the following example).

```
"bytes .cbor type1"
```

Similarly, a ".cborseq" control on a byte string indicates that the byte string carries a sequence of CBOR encoded data items. When the data items are taken as an array, the array matches the type given as the right-hand side argument (type2 in the following example).

```
"bytes .cborseq type2"
```

(The conversion of the encoded sequence to an array can be effected for instance by wrapping the byte string between the two bytes 0x9f and 0xff and decoding the wrapped byte string as a CBOR encoded data item.)

3.8.5. Control operators .within and .and

A ".and" control on a type indicates that the data item matches both that left hand side type and the type given as the right hand side. (Formally, the resulting type is the intersection of the two types given.)

```
"type1 .and type2"
```

A variant of the ".and" control is the ".within" control, which expresses an additional intent: the left hand side type is meant to be a subset of the right-hand-side type.

"type1 .within type2"

While both forms have the identical formal semantics (intersection), the intention of the ".within" form is that the right hand side gives guidance to the types allowed on the left hand side, which typically is a socket (Section 3.9):

```
message = $message .within message-structure
message-structure = [message_type, *message_option]
message_type = 0..255
message_option = any

$message /= [3, dough: text, topping: [* text]]
$message /= [4, noodles: text, sauce: text, parmesan: bool]
```

For ".within", a tool might flag an error if type1 allows data items that are not allowed by type2. In contrast, for ".and", there is no expectation that type1 already is a subset of type2.

3.8.6. Control operators .lt, .le, .gt, .ge, .eq, .ne, and .default

The controls .lt, .le, .gt, .ge, .eq, .ne specify a constraint on the left hand side type to be a value less than, less than or equal, equal to, not equal to, greather than, or greater than or equal to a value given as a (single-valued) right hand side type. In the present specification, the first four controls (.lt, .le, .gt, .ge) are defined only for numeric types, as these have a natural ordering relationship.

```
speed = number .ge 0  ; unit: m/s
```

A variant of the ".ne" control is the ".default" control, which expresses an additional intent: the value specified by the right-hand-side type is intended as a default value for the left hand side type given, and the implied .ne control is there to prevent this value from being sent over the wire. This control is only meaningful when the controld type is used in an optional context; otherwise there would be no way to express the default value.

```
timer = {
  time: uint,
  ? displayed-step: (number .gt 0) .default 1
}
```

3.9. Socket/Plug

Both for type choices and group choices, a mechanism is defined that facilitates starting out with empty choices and assembling them later, potentially in separate files that are concatenated to build the full specification.

Per convention, CDDL extension points are marked with a leading dollar sign (types) or two leading dollar signs (groups). Tools honor that convention by not raising an error if such a type or group is not defined at all; the symbol is then taken to be an empty type choice (group choice), i.e., no choice is available.

```
tcp-header = {seq: uint, ack: uint, * $$tcp-option}

; later, in a different file

$$tcp-option //= (
sack: [+(left: uint, right: uint)]
)

; and, maybe in another file

$$tcp-option //= (
sack-permitted: true
)
```

Names that start with a single "$" are "type sockets", names with a double "$$" are "group sockets". It is not an error if there is no definition for a socket at all; this then means there is no way to satisfy the rule (i.e., the choice is empty).

All definitions (plugs) for socket names must be augmentations, i.e., they must be using "/=" and "//=", respectively.

To pick up the example illustrated in Figure 6, the socket/plug mechanism could be used as shown in Figure 11:

```
PersonalData = {
  ? displayName: tstr,
  NameComponents,
  ? age: uint,
  * $$personaldata-extensions
}

NameComponents = (
  ? firstName: tstr,
  ? familyName: tstr,
)

; The above already works as is.
; But then, we can add later:

$$personaldata-extensions //= (
  favorite-salsa: tstr,
)

; and again, somewhere else:

$$personaldata-extensions //= (
  shoesize: uint,
)
```

Figure 11: Personal Data example: Using socket/plug extensibility

3.10. Generics

Using angle brackets, the left hand side of a rule can add formal parameters after the name being defined, as in:

```
messages = message<"reboot", "now"> / message<"sleep", 1..100>
message<t, v> = {type: t, value: v}
```

When using a generic rule, the formal parameters are bound to the actual arguments supplied (also using angle brackets), within the scope of the generic rule (as if there were a rule of the form parameter = argument).

(There are some limitations to nesting of generics in Appendix F at this time.)

3.11. Operator Precedence

As with any language that has multiple syntactic features such as prefix and infix operators, CDDL has operators that bind more tightly than others. This is becoming more complicated than, say, in ABNF,

as CDDL has both types and groups, with operators that are specific to these concepts. Type operators (such as "/" for type choice) operate on types, while group operators (such as "//" for group choice) operate on groups. Types can simply be used in groups, but groups need to be bracketed (as arrays or maps) to become types. So, type operators naturally bind closer than group operators.

For instance, in

```
t = [group1]
group1 = (a / b // c / d)
a = 1 b = 2 c = 3 d = 4
```

group1 is a group choice between the type choice of a and b and the type choice of c and d. This becomes more relevant once member keys and/or occurrences are added in:

```
t = {group2}
group2 = (? ab: a / b // cd: c / d)
a = 1 b = 2 c = 3 d = 4
```

is a group choice between the optional member "ab" of type a or b and the member "cd" of type c or d. Note that the optionality is attached to the first choice ("ab"), not to the second choice.

Similarly, in

```
t = [group3]
group3 = (+ a / b / c)
a = 1 b = 2 c = 3
```

group3 is a repetition of a type choice between a, b, and c [unflex]; if just a is to be repeatable, a group choice is needed to focus the occurrence:

```
t = [group4]
group4 = (+ a // b / c)
a = 1 b = 2 c = 3
```

group4 is a group choice between a repeatable a and a single b or c.

In general, as with many other languages with operator precedence rules, it is best not to rely on them, but to insert parentheses for readability:

```
t = [group4a]
group4a = ((+ a) // (b / c))
a = 1 b = 2 c = 3
```

The operator precedences, in sequence of loose to tight binding, are defined in Appendix B and summarized in Table 1. (Arities given are 1 for unary prefix operators and 2 for binary infix operators.)

Operator	Ar	Operates on	Prec
=	2	name = type, name = group	1
/=	2	name /= type	1
//=	2	name //= group	1
//	2	group // group	2
,	2	group, group	3
*	1	* group	4
N*M	1	N*M group	4
+	1	+ group	4
?	1	? group	4
=>	2	type => type	5
:	2	name: type	5
/	2	type / type	6
&	1	&group	6
..	2	type..type	7
...	2	type...type	7
.anno	2	type .anno type	7

Table 1: Summary of operator precedences

4. Making Use of CDDL

In this section, we discuss several potential ways to employ CDDL.

4.1. As a guide to a human user

CDDL can be used to efficiently define the layout of CBOR data, such that a human implementer can easily see how data is supposed to be encoded.

Since CDDL maps parts of the CBOR data to human readable names, tools could be built that use CDDL to provide a human friendly representation of the CBOR data, and allow them to edit such data while remaining compliant to its CDDL definition.

4.2. For automated checking of CBOR data structure

CDDL has been specified such that a machine can handle the CDDL definition and related CBOR data (and, thus, also JSON data). For example, a machine could use CDDL to check whether or not CBOR data is compliant to its definition.

The need for thoroughness of such compliance checking depends on the application. For example, an application may decide not to check the data structure at all, and use the CDDL definition solely as a means to indicate the structure of the data to the programmer.

On the other end, the application may also implement a checking mechanism that goes as far as checking that all mandatory map members are available.

The matter in how far the data description must be enforced by an application is left to the designers and implementers of that application, keeping in mind related security considerations.

In no case the intention is that a CDDL tool would be "writing code" for an implementation.

4.3. For data analysis tools

In the long run, it can be expected that more and more data will be stored using the CBOR data format.

Where there is data, there is data analysis and the need to process such data automatically. CDDL can be used for such automated data processing, allowing tools to verify data, clean it, and extract particular parts of interest from it.

Since CBOR is designed with constrained devices in mind, a likely use of it would be small sensors. An interesting use would thus be automated analysis of sensor data.

5. Security considerations

This document presents a content rules language for expressing CBOR data structures. As such, it does not bring any security issues on itself, although specification of protocols that use CBOR naturally need security analysis when defined.

Topics that could be considered in a security considerations section that uses CDDL to define CBOR structures include the following:

- Where could the language maybe cause confusion in a way that will enable security issues?

6. IANA considerations

This document does not require any IANA registrations.

7. References

7.1. Normative References

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

[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>.

[RFC5234] Crocker, D., Ed. and P. Overell, "Augmented BNF for Syntax Specifications: ABNF", STD 68, RFC 5234, DOI 10.17487/RFC5234, January 2008, <https://www.rfc-editor.org/info/rfc5234>.

[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>.

[RFC7493] Bray, T., Ed., "The I-JSON Message Format", RFC 7493, DOI 10.17487/RFC7493, March 2015, <https://www.rfc-editor.org/info/rfc7493>.

[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>.

[W3C.REC-xmlschema-2-20041028] Biron, P. and A. Malhotra, "XML Schema Part 2: Datatypes Second Edition", World Wide Web Consortium Recommendation REC-xmlschema-2-20041028, October 2004, <http://www.w3.org/TR/2004/REC-xmlschema-2-20041028>.

7.2. Informative References

[I-D.bormann-cbor-cddl-freezer] Bormann, C., "A feature freezer for the Concise Data Definition Language (CDDL)", draft-bormann-cbor-cddl-freezer-00 (work in progress), January 2018.

[I-D.ietf-anima-grasp] Bormann, C., Carpenter, B., and B. Liu, "A Generic Autonomic Signaling Protocol (GRASP)", draft-ietf-anima-grasp-15 (work in progress), July 2017.

[I-D.ietf-core-senml]
Jennings, C., Shelby, Z., Arkko, J., Keranen, A., and C. Bormann, "Media Types for Sensor Measurement Lists (SenML)", draft-ietf-core-senml-12 (work in progress), December 2017.

[I-D.newton-json-content-rules]
Newton, A. and P. Cordell, "A Language for Rules Describing JSON Content", draft-newton-json-content-rules-09 (work in progress), September 2017.

[RELAXNG] ISO/IEC, "Information technology -- Document Schema Definition Language (DSDL) -- Part 2: Regular-grammar-based validation -- RELAX NG", ISO/IEC 19757-2, December 2008.

[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>.

[RFC7071] Borenstein, N. and M. Kucherawy, "A Media Type for Reputation Interchange", RFC 7071, DOI 10.17487/RFC7071, November 2013, <https://www.rfc-editor.org/info/rfc7071>.

[RFC7950] Bjorklund, M., Ed., "The YANG 1.1 Data Modeling Language", RFC 7950, DOI 10.17487/RFC7950, August 2016, <https://www.rfc-editor.org/info/rfc7950>.

[RFC8007] Murray, R. and B. Niven-Jenkins, "Content Delivery Network Interconnection (CDNI) Control Interface / Triggers", RFC 8007, DOI 10.17487/RFC8007, December 2016, <https://www.rfc-editor.org/info/rfc8007>.

[RFC8152] Schaad, J., "CBOR Object Signing and Encryption (COSE)", RFC 8152, DOI 10.17487/RFC8152, July 2017, <https://www.rfc-editor.org/info/rfc8152>.

7.3. URIs

[1] https://github.com/cabo/cbor-diag

Appendix A. (Not used.)

Appendix B. ABNF grammar

The following is a formal definition of the CDDL syntax in Augmented Backus-Naur Form (ABNF, [RFC5234]). [_abnftodo]

```
   cddl = S 1*rule
   rule = typename [genericparm] S assign S type S
        / groupname [genericparm] S assign S grpent S

   typename = id
   groupname = id

   assign = "=" / "/=" / "//="

   genericparm = "<" S id S *("," S id S ) ">"
   genericarg = "<" S type1 S *("," S type1 S ) ">"

   type = type1 S *("/" S type1 S)

   type1 = type2 [S (rangeop / ctlop) S type2]

   type2 = value
         / typename [genericarg]
         / "(" type ")"
         / "~" S groupname [genericarg]
         / "#" "6" ["." uint] "(" S type S ")" ; note no space!
         / "#" DIGIT ["." uint]                ; major/ai
         / "#"                                 ; any
         / "{" S group S "}"
         / "[" S group S "]"
         / "&" S "(" S group S ")"
         / "&" S groupname [genericarg]

   rangeop = "..." / ".."

   ctlop = "." id

   group = grpchoice S *("//" S grpchoice S)

   grpchoice = *grpent

   grpent = [occur S] [memberkey S] type optcom
          / [occur S] groupname [genericarg] optcom ; preempted by above
          / [occur S] "(" S group S ")" optcom

   memberkey = type1 S ["^" S] "=>"
             / bareword S ":"
             / value S ":"

   bareword = id

   optcom = S ["," S]
```

```
occur = [uint] "*" [uint]
      / "+"
      / "?"

uint = ["0x" / "0b"] "0"
     / DIGIT1 *DIGIT
     / "0x" 1*HEXDIG
     / "0b" 1*BINDIG

value = number
      / text
      / bytes

int = ["-"] uint

; This is a float if it has fraction or exponent; int otherwise
number = hexfloat / (int ["." fraction] ["e" exponent ])
hexfloat = "0x" 1*HEXDIG ["." 1*HEXDIG] "p" exponent
fraction = 1*DIGIT
exponent = ["+"/"-"] 1*DIGIT

text = %x22 *SCHAR %x22
SCHAR = %x20-21 / %x23-5B / %x5D-10FFFD / SESC
SESC = "\" %x20-10FFFD

bytes = [bsqual] %x27 *BCHAR %x27
BCHAR = %x20-26 / %x28-5B / %x5D-10FFFD / SESC / CRLF
bsqual = %x68 ; "h"
       / %x62.36.34 ; "b64"

id = EALPHA *(*("-" / ".") (EALPHA / DIGIT))
ALPHA = %x41-5A / %x61-7A
EALPHA = %x41-5A / %x61-7A / "@" / "_" / "$"
DIGIT = %x30-39
DIGIT1 = %x31-39
HEXDIG = DIGIT / "A" / "B" / "C" / "D" / "E" / "F"
BINDIG = %x30-31

S = *WS
WS = SP / NL
SP = %x20
NL = COMMENT / CRLF
COMMENT = ";" *PCHAR CRLF
PCHAR = %x20-10FFFD
CRLF = %x0A / %x0D.0A
```

Figure 12: CDDL ABNF

Appendix C. Matching rules

In this appendix, we go through the ABNF syntax rules defined in Appendix B and briefly describe the matching semantics of each syntactic feature. In this context, an instance (data item) "matches" a CDDL specification if it is allowed by the CDDL specification; this is then broken down to parts of specifications (type and group expressions) and parts of instances (data items).

```
cddl = S 1*rule
```

A CDDL specification is a sequence of one or more rules. Each rule gives a name to a right hand side expression, either a CDDL type or a CDDL group. Rule names can be used in the rule itself and/or other rules (and tools can output warnings if that is not the case). The order of the rules is significant only in two cases, including the following: The first rule defines the semantics of the entire specification; hence, its name may be descriptive only (or may be used in itself or other rules as with the other rule names).

```
rule = typename [genericparm] S assign S type S
     / groupname [genericparm] S assign S grpent S

typename = id
groupname = id
```

A rule defines a name for a type expression (production "type") or for a group expression (production "grpent"), with the intention that the semantics does not change when the name is replaced by its (parenthesized if needed) definition.

```
assign = "=" / "/=" / "//="
```

A plain equals sign defines the rule name as the equivalent of the expression to the right. A "/=" or "//=" extends a named type or a group by additional choices; a number of these could be replaced by collecting all the right hand sides and creating a single rule with a type choice or a group choice built from the right hand sides in the order of the rules given. (It is not an error to extend a rule name that has not yet been defined; this makes the right hand side the first entry in the choice being created.) The creation of the type choices and group choices from the right hand sides of rules is the other case where rule order can be significant.

```
genericparm = "<" S id S *("," S id S ) ">"
genericarg = "<" S type1 S *("," S type1 S ) ">"
```

Rule names can have generic parameters, which cause temporary assignments within the right hand sides to the parameter names from the arguments given when citing the rule name.

type = type1 S *("/" S type1 S)

A type can be given as a choice between one or more types. The choice matches a data item if the data item matches any one of the types given in the choice. The choice uses Parse Expression Grammar (PEG) semantics: The first choice that matches wins. (As a result, the order of rules that contribute to a single rule name can very well matter.)

type1 = type2 [S (rangeop / ctlop) S type2]

Two types can be combined with a range operator (which see below) or a control operator (see Section 3.8).

type2 = value

A type can be just a single value (such as 1 or "icecream" or h'0815'), which matches only a data item with that specific value (no conversions defined),

/ typename [genericarg]

or be defined by a rule giving a meaning to a name (possibly after supplying generic args as required by the generic parameters),

/ "(" type ")"

or be defined in a parenthesized type expression (parentheses may be necessary to override some operator precendence), or

/ "~" S groupname [genericarg]

an "unwrapped" group (see Section 3.7), which matches the group inside a type defined as a map or an array by wrapping the group, or

/ "#" "6" ["." uint] "(" S type S ")" ; note no space!

a tagged data item, tagged with the "uint" given and containing the type given as the tagged value, or

/ "#" DIGIT ["." uint] ; major/ai

a data item of a major type (given by the DIGIT), optionally constrained to the additional information given by the uint, or

```
      / "#"                                ; any
```

any data item, or

```
      / "{" S group S "}"
```

a map expression, which matches a valid CBOR map the key/value pairs of which can be ordered in such a way that the resulting sequence matches the group expression, or

```
      / "[" S group S "]"
```

an array expression, which matches a CBOR array the elements of which, when taken as values and complemented by a wildcard (matches anything) key each, match the group, or

```
      / "&" S "(" S group S ")"
      / "&" S groupname [genericarg]
```

an enumeration expression, which matches any a value that is within the set of values that the values of the group given can take.

```
rangeop = "..." / ".."
```

A range operator can be used to join two type expressions that stand for either two integer values or two floating point values; it matches any value that is between the two values, where the first value is always included in the matching set and the second value is included for ".." and excluded for "...".

```
ctlop = "." id
```

A control operator ties a _target_ type to a _controller_ type as defined in Section 3.8. Note that control operators are an extension point for CDDL; additional documents may want to define additional control operators.

```
group = grpchoice S *("//" S grpchoice S)
```

A group matches any sequence of key/value pairs that matches any of the choices given (again using Parse Expression Grammar semantics).

```
grpchoice = *grpent
```

Each of the component groups is given as a sequence of group entries. For a match, the sequence of key/value pairs given needs to match the sequence of group entries in the sequence given.

```
grpent = [occur S] [memberkey S] type optcom
```

A group entry can be given by a value type, which needs to be matched by the value part of a single element, and optionally a memberkey type, which needs to be matched by the key part of the element, if the memberkey is given. If the memberkey is not given, the entry can only be used for matching arrays, not for maps. (See below how that is modified by the occurrence indicator.)

```
       / [occur S] groupname [genericarg] optcom ; preempted by above
```

A group entry can be built from a named group, or

```
       / [occur S] "(" S group S ")" optcom
```

from a parenthesized group, again with a possible occurrence indicator.

```
memberkey = type1 S ["^" S] "=>"
          / bareword S ":"
          / value S ":"
```

Key types can be given by a type expression, a bareword (which stands for string value created from this bareword), or a value (which stands for a type that just contains this value). A key value matches its key type if the key value is a member of the key type, unless a cut preceding it in the group applies (see Section 3.5.3 how map matching is infuenced by the presence of the cuts denoted by "^" or ":" in previous entries).

```
bareword = id
```

A bareword is an alternative way to write a type with a single text string value; it can only be used in the syntactic context given above.

```
optcom = S ["," S]
```

(Optional commas do not influence the matching.)

```
occur = [uint] "*" [uint]
      / "+"
      / "?"
```

An occurrence indicator modifies the group given to its right by requiring the group to match the sequence to be matched exactly for a certain number of times (see Section 3.2) in sequence, i.e. it acts

as a (possibly infinite) group choice that contains choices with the group repeated each of the occurrences times.

The rest of the ABNF describes syntax for value notation that should be familiar from programming languages, with the possible exception of h'..' and b64'..' for byte strings, as well as syntactic elements such as comments and line ends.

Appendix D. (Not used.)

Appendix E. Standard Prelude

The following prelude is automatically added to each CDDL file [tdate]. (Note that technically, it is a postlude, as it does not disturb the selection of the first rule as the root of the definition.)

```
any = #

uint = #0
nint = #1
int = uint / nint

bstr = #2
bytes = bstr
tstr = #3
text = tstr

tdate = #6.0(tstr)
time = #6.1(number)
number = int / float
biguint = #6.2(bstr)
bignint = #6.3(bstr)
bigint = biguint / bignint
integer = int / bigint
unsigned = uint / biguint
decfrac = #6.4([e10: int, m: integer])
bigfloat = #6.5([e2: int, m: integer])
eb64url = #6.21(any)
eb64legacy = #6.22(any)
eb16 = #6.23(any)
encoded-cbor = #6.24(bstr)
uri = #6.32(tstr)
b64url = #6.33(tstr)
b64legacy = #6.34(tstr)
regexp = #6.35(tstr)
mime-message = #6.36(tstr)
cbor-any = #6.55799(any)

float16 = #7.25
float32 = #7.26
float64 = #7.27
float16-32 = float16 / float32
float32-64 = float32 / float64
float = float16-32 / float64

false = #7.20
true = #7.21
bool = false / true
nil = #7.22
null = nil
undefined = #7.23
```

Figure 13: CDDL Prelude

Note that the prelude is deemed to be fixed. This means, for instance, that additional tags beyond [RFC7049], as registered, need to be defined in each CDDL file that is using them.

A common stumbling point is that the prelude does not define a type "string". CBOR has byte strings ("bytes" in the prelude) and text strings ("text"), so a type that is simply called "string" would be ambiguous.

E.1. Use with JSON

The JSON generic data model (implicit in [RFC8259]) is a subset of the generic data model of CBOR. So one can use CDDL with JSON by limiting oneself to what can be represented in JSON. Roughly speaking, this means leaving out byte strings, tags, and simple values other than "false", "true", and "null", leading to the following limited prelude:

```
any = #

uint = #0
nint = #1
int = uint / nint

tstr = #3
text = tstr

number = int / float

float16 = #7.25
float32 = #7.26
float64 = #7.27
float16-32 = float16 / float32
float32-64 = float32 / float64
float = float16-32 / float64

false = #7.20
true = #7.21
bool = false / true
nil = #7.22
null = nil
```

Figure 14: JSON compatible subset of CDDL Prelude

(The major types given here do not have a direct meaning in JSON, but they can be interpreted as CBOR major types translated through Section 4 of [RFC7049].)

There are a few fine points in using CDDL with JSON. First, JSON does not distinguish between integers and floating point numbers; there is only one kind of number (which may happen to be integral). In this context, specifying a type as "uint", "nint" or "int" then becomes a predicate that the number be integral. As an example, this means that the following JSON numbers are all matching "uint":

10 10.0 1e1 1.0e1 100e-1

(The fact that these are all integers may be surprising to users accustomed to the long tradition in programming languages of using decimal points or exponents in a number to indicate a floating point literal.)

CDDL distinguishes the various CBOR number types, but there is only one number type in JSON. The effect of specifying a floating point precision (float16/float32/float64) is only to restrict the set of permissible values to those expressible with binary16/binary32/binary64; this is unlikely to be very useful when using CDDL for specifying JSON data structures.

Fundamentally, the number system of JSON itself is based on decimal numbers and decimal fractions and does not have limits to its precision or range. In practice, JSON numbers are often parsed into a number type that is called float64 here, creating a number of limitations to the generic data model [RFC7493]. In particular, this means that integers can only be expressed with interoperable exactness when they lie in the range [-(2**53)+1, (2**53)-1] -- a smaller range than that covered by CDDL "int".

JSON applications that want to stay compatible with I-JSON therefore may want to define integer types with more limited ranges, such as in Figure 15. Note that the types given here are not part of the prelude; they need to be copied into the CDDL specification if needed.

```
ij-uint = 0..9007199254740991
ij-nint = -9007199254740991..-1
ij-int = -9007199254740991..9007199254740991
```

Figure 15: I-JSON types for CDDL (not part of prelude)

JSON applications that do not need to stay compatible with I-JSON and that actually may need to go beyond the 64-bit unsigned and negative integers supported by "int" (= "uint"/"nint") may want to use the following additional types from the standard prelude, which are expressed in terms of tags but can straightforwardly be mapped into JSON (but not I-JSON) numbers:

```
biguint = #6.2(bstr)
bignint = #6.3(bstr)
bigint = biguint / bignint
integer = int / bigint
unsigned = uint / biguint
```

CDDL at this point does not have a way to express the unlimited floating point precision that is theoretically possible with JSON; at the time of writing, this is rarely used in protocols in practice.

Note that a data model described in CDDL is always restricted by what can be expressed in the serialization; e.g., floating point values such as NaN (not a number) and the infinities cannot be represented in JSON even if they are allowed in the CDDL generic data model.

Appendix F. The CDDL tool

A rough CDDL tool is available. For CDDL specifications, it can check the syntax, generate one or more instances (expressed in CBOR diagnostic notation or in pretty-printed JSON), and validate an existing instance against the specification:

```
Usage:
cddl spec.cddl generate [n]
cddl spec.cddl json-generate [n]
cddl spec.cddl validate instance.cbor
cddl spec.cddl validate instance.json
```

Figure 16: CDDL tool usage

Install on a system with a modern Ruby via:

```
gem install cddl
```

Figure 17: CDDL tool installation

The accompanying CBOR diagnostic tools (which are automatically installed by the above) are described in https://github.com/cabo/cbor-diag [1]; they can be used to convert between binary CBOR, a pretty-printed form of that, CBOR diagnostic notation, JSON, and YAML.

Appendix G. Extended Diagnostic Notation

Section 6 of [RFC7049] defines a "diagnostic notation" in order to be able to converse about CBOR data items without having to resort to binary data. Diagnostic notation is based on JSON, with extensions for representing CBOR constructs such as binary data and tags.

(Standardizing this together with the actual interchange format does not serve to create another interchange format, but enables the use of a shared diagnostic notation in tools for and documents about CBOR.)

This section discusses a few extensions to the diagnostic notation that have turned out to be useful since RFC 7049 was written. We refer to the result as extended diagnostic notation (EDN).

G.1. White space in byte string notation

Examples often benefit from some white space (spaces, line breaks) in byte strings. In extended diagnostic notation, white space is ignored in prefixed byte strings; for instance, the following are equivalent:

```
  h'48656c6c6f20776f726c64'
  h'48 65 6c 6c 6f 20 77 6f 72 6c 64'
  h'4 86 56c 6c6f
    20776 f726c64'
```

G.2. Text in byte string notation

Diagnostic notation notates Byte strings in one of the [RFC4648] base encodings,, enclosed in single quotes, prefixed by >h< for base16, >b32< for base32, >h32< for base32hex, >b64< for base64 or base64url. Quite often, byte strings carry bytes that are meaningfully interpreted as UTF-8 text. Extended Diagnostic Notation allows the use of single quotes without a prefix to express byte strings with UTF-8 text; for instance, the following are equivalent:

```
'hello world'
h'68656c6c6f20776f726c64'
```

The escaping rules of JSON strings are applied equivalently for text-based byte strings, e.g., \ stands for a single backslash and ' stands for a single quote. White space is included literally, i.e., the previous section does not apply to text-based byte strings.

G.3. Embedded CBOR and CBOR sequences in byte strings

Where a byte string is to carry an embedded CBOR-encoded item, or more generally a sequence of zero or more such items, the diagnostic notation for these zero or more CBOR data items, separated by commata, can be enclosed in << and >> to notate the byte string resulting from encoding the data items and concatenating the result. For instance, each pair of columns in the following are equivalent:

```
<<1>>              h'01'
<<1, 2>>           h'0102'
<<"foo", null>>    h'63666F6FF6'
<<>>               h''
```

G.4. Concatenated Strings

While the ability to include white space enables line-breaking of encoded byte strings, a mechanism is needed to be able to include text strings as well as byte strings in direct UTF-8 representation into line-based documents (such as RFCs and source code).

We extend the diagnostic notation by allowing multiple text strings or multiple byte strings to be notated separated by white space, these are then concatenated into a single text or byte string, respectively. Text strings and byte strings do not mix within such a concatenation, except that byte string notation can be used inside a sequence of concatenated text string notation to encode characters that may be better represented in an encoded way. The following four values are equivalent:

```
"Hello world"
"Hello " "world"
"Hello" h'20' "world"
"" h'48656c6c6f20776f726c64' ""
```

Similarly, the following byte string values are equivalent

```
'Hello world'
'Hello ' 'world'
'Hello ' h'776f726c64'
'Hello' h'20' 'world'
'' h'48656c6c6f20776f726c64' '' b64''
h'4 86 56c 6c6f' h' 20776 f726c64'
```

(Note that the approach of separating by whitespace, while familiar from the C language, requires some attention - a single comma makes a big difference here.)

G.5. Hexadecimal, octal, and binary numbers

In addition to JSON's decimal numbers, EDN provides hexadecimal, octal and binary numbers in the usual C-language notation (octal with 0o prefix present only).

The following are equivalent:

```
4711
0x1267
0o11147
0b1001001100111
```

As are:

```
1.5
0x1.8p0
0x18p-4
```

G.6. Comments

Longer pieces of diagnostic notation may benefit from comments. JSON famously does not provide for comments, and basic RFC 7049 diagnostic notation inherits this property.

In extended diagnostic notation, comments can be included, delimited by slashes ("/"). Any text within and including a pair of slashes is considered a comment.

Comments are considered white space. Hence, they are allowed in prefixed byte strings; for instance, the following are equivalent:

```
h'68656c6c6f20776f726c64'
h'68 65 6c /doubled l!/ 6c 6f /hello/
  20 /space/
  77 6f 72 6c 64' /world/
```

This can be used to annotate a CBOR structure as in:

```
/grasp-message/ [/M_DISCOVERY/ 1, /session-id/ 10584416,
                 /objective/ [/objective-name/ "opsonize",
                              /D, N, S/ 7, /loop-count/ 105]]
```

(There are currently no end-of-line comments. If we want to add them, "//" sounds like a reasonable delimiter given that we already use slashes for comments, but we also could go e.g. for "#".)

Appendix H. Examples

This section contains various examples of structures defined using CDDL.

The theme for the first example is taken from [RFC7071], which defines certain JSON structures in English. For a similar example, it may also be of interest to examine Appendix A of [RFC8007], which

contains a CDDL definition for a JSON structure defined in the main body of the RFC.

The second subsection in this appendix translates examples from [I-D.newton-json-content-rules] into CDDL.

These examples all happen to describe data that is interchanged in JSON. Examples for CDDL definitions of data that is interchanged in CBOR can be found in [RFC8152], [I-D.ietf-anima-grasp], or [I-D.ietf-core-senml].

H.1. RFC 7071

[RFC7071] defines the Reputon structure for JSON using somewhat formalized English text. Here is a (somewhat verbose) equivalent definition using the same terms, but notated in CDDL:

```
reputation-object = {
  reputation-context,
  reputon-list
}

reputation-context = (
  application: text
)

reputon-list = (
  reputons: reputon-array
)

reputon-array = [* reputon]

reputon = {
  rater-value,
  assertion-value,
  rated-value,
  rating-value,
  ? conf-value,
  ? normal-value,
  ? sample-value,
  ? gen-value,
  ? expire-value,
  * ext-value,
}

rater-value = ( rater: text )
assertion-value = ( assertion: text )
rated-value = ( rated: text )
rating-value = ( rating: float16 )
conf-value = ( confidence: float16 )
normal-value = ( normal-rating: float16 )
sample-value = ( sample-size: uint )
gen-value = ( generated: uint )
expire-value = ( expires: uint )
ext-value = ( text => any )
```

An equivalent, more compact form of this example would be:

```
reputation-object = {
  application: text
  reputons: [* reputon]
}

reputon = {
  rater: text
  assertion: text
  rated: text
  rating: float16
  ? confidence: float16
  ? normal-rating: float16
  ? sample-size: uint
  ? generated: uint
  ? expires: uint
  * text => any
}
```

Note how this rather clearly delineates the structure somewhat shrouded by so many words in section 6.2.2. of [RFC7071]. Also, this definition makes it clear that several ext-values are allowed (by definition with different member names); RFC 7071 could be read to forbid the repetition of ext-value ("A specific reputon-element MUST NOT appear more than once" is ambiguous.)

The CDDL tool (which hasn't quite been trained for polite conversation) says:

```
{
  "application": "tridentiferous",
  "reputons": [
    {
      "rater": "loamily",
      "assertion": "Dasyprocta",
      "rated": "uncommensurableness",
      "rating": 0.05055809746548934,
      "confidence": 0.7484706448605812,
      "normal-rating": 0.8677887734049299,
      "sample-size": 4059,
      "expires": 3969,
      "bearer": "nitty",
      "faucal": "postulnar",
      "naturalism": "sarcotic"
    },
    {
      "rater": "precreed",
      "assertion": "xanthosis",
      "rated": "balsamy",
      "rating": 0.36091333590593955,
      "confidence": 0.3700759808403371,
      "sample-size": 3904
    },
    {
      "rater": "urinosexual",
      "assertion": "malacostracous",
      "rated": "arenariae",
      "rating": 0.9210673488013762,
      "normal-rating": 0.4778762617112776,
      "sample-size": 4428,
      "generated": 3294,
      "backfurrow": "enterable",
      "fruitgrower": "flannelflower"
    },
    {
      "rater": "pedologistically",
      "assertion": "unmetaphysical",
      "rated": "elocutionist",
      "rating": 0.42073613384304287,
      "misimagine": "retinaculum",
      "snobbish": "contradict",
      "Bosporanic": "periostotomy",
      "dayworker": "intragyral"
    }
  ]
}
```

H.1.1. Examples from JSON Content Rules

Although JSON Content Rules [I-D.newton-json-content-rules] seems to address a more general problem than CDDL, it is still a worthwhile resource to explore for examples (beyond all the inspiration the format itself has had for CDDL).

Figure 2 of the JCR I-D looks very similar, if slightly less noisy, in CDDL:

```
root = [2*2 {
  precision: text,
  Latitude: float,
  Longitude: float,
  Address: text,
  City: text,
  State: text,
  Zip: text,
  Country: text
}]
```

Figure 18: JCR, Figure 2, in CDDL

Apart from the lack of a need to quote the member names, text strings are called "text" or "tstr" in CDDL ("string" would be ambiguous as CBOR also provides byte strings).

The CDDL tool creates the below example instance for this:

```
 [{"precision": "pyrosphere", "Latitude": 0.5399712314350172,
   "Longitude": 0.5157523963028087, "Address": "resow",
   "City": "problemwise", "State": "martyrlike", "Zip": "preprove",
   "Country": "Pace"},
  {"precision": "unrigging", "Latitude": 0.10422704368372193,
   "Longitude": 0.6279808663725834, "Address": "picturedom",
   "City": "decipherability", "State": "autometry", "Zip": "pout",
   "Country": "wimple"}]
```

Figure 4 of the JCR I-D in CDDL:

```
root = { image }

image = (
  Image: {
    size,
    Title: text,
    thumbnail,
    IDs: [* int]
  }
)

size = (
  Width: 0..1280
  Height: 0..1024
)

thumbnail = (
  Thumbnail: {
    size,
    Url: ~uri
  }
)
```

This shows how the group concept can be used to keep related elements (here: width, height) together, and to emulate the JCR style of specification. (It also shows referencing a type by unwrapping a tag from the prelude, "uri" - this could be done differently.) The more compact form of Figure 5 of the JCR I-D could be emulated like this:

```
root = {
  Image: {
    size, Title: text,
    Thumbnail: { size, Url: ~uri },
    IDs: [* int]
  }
}

size = (
  Width: 0..1280,
  Height: 0..1024,
)
```

The CDDL tool creates the below example instance for this:

```
{"Image": {"Width": 566, "Height": 516, "Title": "leisterer",
  "Thumbnail": {"Width": 1111, "Height": 176, "Url": 32("scrog")},
  "IDs": []}}
```

Acknowledgements

CDDL was originally conceived by Bert Greevenbosch, who also wrote the original five versions of this document.

Inspiration was taken from the C and Pascal languages, MPEG's conventions for describing structures in the ISO base media file format, Relax-NG and its compact syntax [RELAXNG], and in particular from Andrew Lee Newton's "JSON Content Rules" [I-D.newton-json-content-rules].

Useful feedback came from members of the IETF CBOR WG, in particular Joe Hildebrand, Sean Leonard and Jim Schaad. Also, Francesca Palombini and Joe volunteered to chair this WG, providing the framework for generating and processing this feedback.

The CDDL tool was written by Carsten Bormann, building on previous work by Troy Heninger and Tom Lord.

Editorial Comments

[_format] So far, the ability to restrict format choices have not been needed beyond the floating point formats. Those can be applied to ranges using the new .and control now. It is not clear we want to add more format control before we have a use case.

[_range] TO DO: define this precisely. This clearly includes integers and floats. Strings - as in "a".."z" - could be added if desired, but this would require adopting a definition of string ordering and possibly a successor function so "a".."z" does not include "bb".

[_strings] TO DO: This still needs to be fully realized in the ABNF and in the CDDL tool.

[_bitsendian] How useful would it be to have another variant that counts bits like in RFC box notation? (Or at least per-byte? 32-bit words don't always perfectly mesh with byte strings.)

[unflex] A comment has been that this is counter-intuitive. One solution would be to simply disallow unparenthesized usage of occurrence indicators in front of type choices unless a member key is also present like in group2 above.

[_abnftodo] Potential improvements: the prefixed byte strings are more liberally specified than they actually are.

[tdate] The prelude as included here does not yet have a .regexp control on tdate, but we probably do want to have one.

Authors' Addresses

Henk Birkholz
Fraunhofer SIT
Rheinstrasse 75
Darmstadt 64295
Germany

Email: henk.birkholz@sit.fraunhofer.de

Christoph Vigano
Universitaet Bremen

Email: christoph.vigano@uni-bremen.de

Carsten Bormann
Universitaet Bremen TZI
Bibliothekstr. 1
Bremen D-28359
Germany

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

CBOR
Internet-Draft
Intended status: Standards Track
Expires: September 25, 2019

H. Birkholz
Fraunhofer SIT
C. Vigano
Universitaet Bremen
C. Bormann
Universitaet Bremen TZI

March 24, 2019

Concise data definition language (CDDL): a notational convention to express CBOR and JSON data structures
draft-ietf-cbor-cddl-08

Abstract

This document proposes a notational convention to express CBOR data structures (RFC 7049, Concise Binary Object Representation). Its main goal is to provide an easy and unambiguous way to express structures for protocol messages and data formats that use CBOR or JSON.

Table of Contents

1. Introduction

In this document, a notational convention to express CBOR [RFC7049] data structures is defined.

The main goal for the convention is to provide a unified notation that can be used when defining protocols that use CBOR. We term the convention "Concise data definition language", or CDDL.

The CBOR notational convention has the following goals:

(G1) Provide an unambiguous description of the overall structure of a CBOR data item.

(G2) Be flexible in expressing the multiple ways in which data can be represented in the CBOR data format.

(G3) Be able to express common CBOR datatypes and structures.

(G4) Provide a single format that is both readable and editable for humans and processable by machine.

(G5) Enable automatic checking of CBOR data items for data format compliance.

(G6) Enable extraction of specific elements from CBOR data for further processing.

Not an original goal per se, but a convenient side effect of the JSON generic data model being a subset of the CBOR generic data model, is the fact that CDDL can also be used for describing JSON data structures (see Appendix E).

This document has the following structure:

The syntax of CDDL is defined in Section 3. Examples of CDDL and related CBOR data items ("instances", which all happen to be in JSON form) are given in Appendix H. Section 4 discusses usage of CDDL. Examples are provided early in the text to better illustrate concept definitions. A formal definition of CDDL using ABNF grammar is provided in Appendix B. Finally, a _prelude_ of standard CDDL definitions that is automatically prepended to and thus available in every CBOR specification is listed in Appendix D.

1.1. Requirements notation

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

1.2. Terminology

New terms are introduced in _cursive_, which is rendered in plain text as the new term surrounded by underscores. CDDL text in the running text is in "typewriter", which is rendered in plain text as the CDDL text in double quotes (double quotes are also used in the usual English sense; the reader is expected to disambiguate this by context).

In this specification, the term "byte" is used in its now customary sense as a synonym for "octet".

2. The Style of Data Structure Specification

CDDL focuses on styles of specification that are in use in the community employing the data model as pioneered by JSON and now refined in CBOR.

There are a number of more or less atomic elements of a CBOR data model, such as numbers, simple values (false, true, nil), text and

byte strings; CDDL does not focus on specifying their structure. CDDL of course also allows adding a CBOR tag to a data item.

Beyond those atomic elements, further components of a data structure definition language are the data types used for composition: arrays and maps in CBOR (called arrays and objects in JSON). While these are only two representation formats, they are used to specify four loosely distinguishable styles of composition:

- A _vector_, an array of elements that are mostly of the same semantics. The set of signatures associated with a signed data item is a typical application of a vector.

- A _record_, an array the elements of which have different, positionally defined semantics, as detailed in the data structure definition. A 2D point, specified as an array of an x coordinate (which comes first) and a y coordinate (coming second) is an example of a record, as is the pair of exponent (first) and mantissa (second) in a CBOR decimal fraction.

- A _table_, a map from a domain of map keys to a domain of map values, that are mostly of the same semantics. A set of language tags, each mapped to a text string translated to that specific language, is an example of a table. The key domain is usually not limited to a specific set by the specification, but open for the application, e.g., in a table mapping IP addresses to MAC addresses, the specification does not attempt to foresee all possible IP addresses. In a language such as JavaScript, a "Map" (as opposed to a plain "Object") would often be employed to achieve the generality of the key domain.

- A _struct_, a map from a domain of map keys as defined by the specification to a domain of map values the semantics of each of which is bound to a specific map key. This is what many people have in mind when they think about JSON objects; CBOR adds the ability to use map keys that are not just text strings. Structs can be used to solve similar problems as records; the use of explicit map keys facilitates optionality and extensibility.

Two important concepts provide the foundation for CDDL:

1. Instead of defining all four types of composition in CDDL separately, or even defining one kind for arrays (vectors and records) and one kind for maps (tables and structs), there is only one kind of composition in CDDL: the _group_ (Section 2.1).

2. The other important concept is that of a _type_. The entire CDDL specification defines a type (the one defined by its first

rule), which formally is the set of CBOR data items that are acceptable as "instances" for this specification. CDDL predefines a number of basic types such as "uint" (unsigned integer) or "tstr" (text string), often making use of a simple formal notation for CBOR data items. Each value that can be expressed as a CBOR data item also is a type in its own right, e.g. "1". A type can be built as a _choice_ of other types, e.g., an "int" is either a "uint" or a "nint" (negative integer). Finally, a type can be built as an array or a map from a group.

The rest of this section introduces a number of basic concepts of CDDL, and Section 3 defines additional syntax. Appendix C gives a concise summary of the semantics of CDDL.

2.1. Groups and Composition in CDDL

CDDL Groups are lists of group _entries_, each of which can be a name/value pair or a more complex group expression (which then in turn stands for a sequence of name/value pairs). A CDDL group is a production in a grammar that matches certain sequences of name/value pairs but not others. The grammar is based on the concepts of Parsing Expression Grammars (see Appendix A).

In an array context, only the value of the name/value pair is represented; the name is annotation only (and can be left off from the group specification if not needed). In a map context, the names become the map keys ("member keys").

In an array context, the actual sequence of elements in the group is important, as that sequence is the information that allows associating actual array elements with entries in the group. In a map context, the sequence of entries in a group is not relevant (but there is still a need to write down group entries in a sequence).

An array matches a specification given as a group when the group matches a sequence of name/value pairs the value parts of which exactly match the elements of the array in order.

A map matches a specification given as a group when the group matches a sequence of name/value pairs such that all of these name/value pairs are present in the map and the map has no name/value pair that is not covered by the group.

A simple example of using a group directly in a map definition is:

```
person = {
  age: int,
  name: tstr,
  employer: tstr,
}
```

Figure 1: Using a group directly in a map

The three entries of the group are written between the curly braces that create the map: Here, "age", "name", and "employer" are the names that turn into the map key text strings, and "int" and "tstr" (text string) are the types of the map values under these keys.

A group by itself (without creating a map around it) can be placed in (round) parentheses, and given a name by using it in a rule:

```
pii = (
  age: int,
  name: tstr,
  employer: tstr,
)
```

Figure 2: A basic group

This separate, named group definition allows us to rephrase Figure 1 as:

```
person = {
  pii
}
```

Figure 3: Using a group by name

Note that the (curly) braces signify the creation of a map; the groups themselves are neutral as to whether they will be used in a map or an array.

As shown in Figure 1, the parentheses for groups are optional when there is some other set of brackets present. Note that they can still be used, leading to the not so realistic, but perfectly valid example:

```
person = {(
  age: int,
  name: tstr,
  employer: tstr,
)}
```

Figure 4: Using a parenthesized group in a map

Groups can be used to factor out common parts of structs, e.g., instead of writing copy/paste style specifications such as in Figure 5, one can factor out the common subgroup, choose a name for it, and write only the specific parts into the individual maps (Figure 6).

```
person = {
  age: int,
  name: tstr,
  employer: tstr,
}

dog = {
  age: int,
  name: tstr,
  leash-length: float,
}
```

Figure 5: Maps with copy/paste

```
person = {
  identity,
  employer: tstr,
}

dog = {
  identity,
  leash-length: float,
}

identity = (
  age: int,
  name: tstr,
)
```

Figure 6: Using a group for factorization

Note that the lists inside the braces in the above definitions constitute (anonymous) groups, while "identity" is a named group,

which can then be included as part of other groups (anonymous as in the example, or themselves named).

2.1.1. Usage

Groups are the instrument used in composing data structures with CDDL. It is a matter of style in defining those structures whether to define groups (anonymously) right in their contexts or whether to define them in a separate rule and to reference them with their respective name (possibly more than once).

With this, one is allowed to define all small parts of their data structures and compose bigger protocol units with those or to have only one big protocol data unit that has all definitions ad hoc where needed.

2.1.2. Syntax

The composition syntax is intended to be concise and easy to read:

- The start and end of a group can be marked by '(' and ')'
- Definitions of entries inside of a group are noted as follows: _keytype => valuetype,_ (read "keytype maps to valuetype"). The comma is actually optional (not just in the final entry), but it is considered good style to set it. The double arrow can be replaced by a colon in the common case of directly using a text string or integer literal as a key (see Section 3.5.1; this is also the common way of naming elements of an array just for documentation, see Section 3.4).

A basic entry consists of a _keytype_ and a _valuetype_, both of which are types (Section 2.2); this entry matches any name-value pair the name of which is in the keytype and the value of which is in the valuetype.

A group defined as a sequence of group entries matches any sequence of name-value pairs that is composed by concatenation in order of what the entries match.

A group definition can also contain choices between groups, see Section 2.2.2.

2.2. Types

2.2.1. Values

Values such as numbers and strings can be used in place of a type. (For instance, this is a very common thing to do for a keytype, common enough that CDDL provides additional convenience syntax for this.)

The value notation is based on the C language, but does not offer all the syntactic variations (see Appendix B for details). The value notation for numbers inherits from C the distinction between integer values (no fractional part or exponent given -- NR1 [ISO6093]) and floating point values (where a fractional part and/or an exponent is present -- NR2 or NR3), so the type "1" does not include any floating point numbers while the types "1e3" and "1.5" are both floating point numbers and do not include any integer numbers.

2.2.2. Choices

Many places that allow a type also allow a choice between types, delimited by a "/" (slash). The entire choice construct can be put into parentheses if this is required to make the construction unambiguous (please see Appendix B for the details).

Choices of values can be used to express enumerations:

```
attire = "bow tie" / "necktie" / "Internet attire"
protocol = 6 / 17
```

Similarly as for types, CDDL also allows choices between groups, delimited by a "//" (double slash). Note that the "//" operator binds much more weakly than the other CDDL operators, so each line within "delivery" in the following example is its own alternative in the group choice:

```
address = { delivery }

delivery = (
street: tstr, ? number: uint, city //
po-box: uint, city //
per-pickup: true )

city = (
name: tstr, zip-code: uint
)
```

A group choice matches the union of the sets of name-value pair sequences that the alternatives in the choice can.

Both for type choices and for group choices, additional alternatives can be added to a rule later in separate rules by using "/=" and "//=", respectively, instead of "=":

```
attire /= "swimwear"

delivery //= (
lat: float, long: float, drone-type: tstr
)
```

It is not an error if a name is first used with a "/=" or "//=" (there is no need to "create it" with "=").

2.2.2.1. Ranges

Instead of naming all the values that make up a choice, CDDL allows building a _range_ out of two values that are in an ordering relationship: A lower bound (first value) and an upper bound (second value). A range can be inclusive of both bounds given (denoted by joining two values by ".."), or include the lower bound and exclude the upper bound (denoted by instead using "..."). If the lower bound exceeds the upper bound, the resulting type is the empty set (this behavior can be desirable when generics, Section 3.10, are being used).

```
device-address = byte
max-byte = 255
byte = 0..max-byte ; inclusive range
first-non-byte = 256
byte1 = 0...first-non-byte ; byte1 is equivalent to byte
```

CDDL currently only allows ranges between integers (matching integer values) or between floating point values (matching floating point values). If both are needed in a type, a type choice between the two kinds of ranges can be (clumsily) used:

```
int-range = 0..10 ; only integers match
float-range = 0.0..10.0 ; only floats match
BAD-range1 = 0..10.0 ; NOT DEFINED
BAD-range2 = 0.0..10 ; NOT DEFINED
numeric-range = int-range / float-range
```

(See also the control operators .lt/.ge and .le/.gt in Section 3.8.6.)

Note that the dot is a valid name continuation character in CDDL, so

```
min..max
```

is not a range expression but a single name. When using a name as the left hand side of a range operator, use spacing as in

```
min .. max
```

to separate off the range operator.

2.2.2.2. Turning a group into a choice

Some choices are built out of large numbers of values, often integers, each of which is best given a semantic name in the specification. Instead of naming each of these integers and then accumulating these into a choice, CDDL allows building a choice from a group by prefixing it with a "&" character:

```
terminal-color = &basecolors
basecolors = (
  black: 0, red: 1,  green: 2,  yellow: 3,
  blue: 4,  magenta: 5,  cyan: 6,  white: 7,
)
extended-color = &(
  basecolors,
  orange: 8,  pink: 9,  purple: 10,  brown: 11,
)
```

As with the use of groups in arrays (Section 3.4), the member names have only documentary value (in particular, they might be used by a tool when displaying integers that are taken from that choice).

2.2.3. Representation Types

CDDL allows the specification of a data item type by referring to the CBOR representation (major types and additional information, Section 2 of [RFC7049]). How this is used should be evident from the prelude (Appendix D): a hash mark ("#") optionally followed by a number from 0 to 7 identifying the major type, which then can be followed by a dot and a number specifying the additional information. This construction specifies the set of values that can be serialized in CBOR (i.e., "any"), by the given major type if one is given, or by the given major type with the additional information if both are given. Where a major type of 6 (Tag) is used, the type of the tagged item can be specified by appending it in parentheses.

Note that although this notation is based on the CBOR serialization, it is about a set of values at the data model level, e.g. "#7.25" specifies the set of values that can be represented as half-precision floats; it does not mandate that these values also do have to be serialized as half-precision floats: CDDL does not provide any

language means to restrict the choice of serialization variants. This also enables the use of CDDL with JSON, which uses a fundamentally different way of serializing (some of) the same values.

It may be necessary to make use of representation types outside the prelude, e.g., a specification could start by making use of an existing tag in a more specific way, or define a new tag not defined in the prelude:

```
my_breakfast = #6.55799(breakfast)   ; cbor-any is too general!
breakfast = cereal / porridge
cereal = #6.998(tstr)
porridge = #6.999([liquid, solid])
liquid = milk / water
milk = 0
water = 1
solid = tstr
```

2.2.4. Root type

There is no special syntax to identify the root of a CDDL data structure definition: that role is simply taken by the first rule defined in the file.

This is motivated by the usual top-down approach for defining data structures, decomposing a big data structure unit into smaller parts; however, except for the root type, there is no need to strictly follow this sequence.

(Note that there is no way to use a group as a root - it must be a type.)

3. Syntax

In this section, the overall syntax of CDDL is shown, alongside some examples just illustrating syntax. (The definition will not attempt to be overly formal; refer to Appendix B for the details.)

3.1. General conventions

The basic syntax is inspired by ABNF [RFC5234], with

- rules, whether they define groups or types, are defined with a name, followed by an equals sign "=" and the actual definition according to the respective syntactic rules of that definition.

- A name can consist of any of the characters from the set {'A' to 'Z', 'a' to 'z', '0' to '9', '_', '-', '@', '.', '$'}, starting

with an alphabetic character (including '@', '_', '$') and ending in such a character or or a digit.

* Names are case sensitive.

* It is preferred style to start a name with a lower case letter.

* The hyphen is preferred over the underscore (except in a "bareword" (Section 3.5.1), where the semantics may actually require an underscore).

* The period may be useful for larger specifications, to express some module structure (as in "tcp.throughput" vs. "udp.throughput").

* A number of names are predefined in the CDDL prelude, as listed in Appendix D.

* Rule names (types or groups) do not appear in the actual CBOR encoding, but names used as "barewords" in member keys do.

o Comments are started by a ';' (semicolon) character and finish at the end of a line (LF or CRLF).

o outside strings, whitespace (spaces, newlines, and comments) is used to separate syntactic elements for readability (and to separate identifiers, range operators, or numbers that follow each other); it is otherwise completely optional.

o Hexadecimal numbers are preceded by '0x' (without quotes, lower case x), and are case insensitive. Similarly, binary numbers are preceded by '0b'.

o Text strings are enclosed by double quotation '"' characters. They follow the conventions for strings as defined in section 7 of [RFC8259]. (ABNF users may want to note that there is no support in CDDL for the concept of case insensitivity in text strings; if necessary, regular expressions can be used (Section 3.8.3).)

o Byte strings are enclosed by single quotation "'" characters and may be prefixed by "h" or "b64". If unprefixed, the string is interpreted as with a text string, except that single quotes must be escaped and that the UTF-8 bytes resulting are marked as a byte string (major type 2). If prefixed as "h" or "b64", the string is interpreted as a sequence of pairs of hex digits (base16, Section 8 of [RFC4648]) or a base64(url) string (Sections 4 or 5 of [RFC4648]), respectively (as with the diagnostic notation in section 6 of [RFC7049]; cf. Appendix G.2); any white space present

within the string (including comments) is ignored in the prefixed case.

- CDDL uses UTF-8 [RFC3629] for its encoding. Processing of CDDL does not involve Unicode normalization processes.

Example:

```
; This is a comment
person = { g }

g = (
  "name": tstr,
  age: int,  ; "age" is a bareword
)
```

3.2. Occurrence

An optional _occurrence_ indicator can be given in front of a group entry. It is either one of the characters '?' (optional), '*' (zero or more), or '+' (one or more), or is of the form n*m, where n and m are optional unsigned integers and n is the lower limit (default 0) and m is the upper limit (default no limit) of occurrences.

If no occurrence indicator is specified, the group entry is to occur exactly once (as if 1*1 were specified). A group entry with an occurrence indicator matches sequences of name-value pairs that are composed by concatenating a number of sequences that the basic group entry matches, where the number needs to be allowed by the occurrence indicator.

Note that CDDL, outside any directives/annotations that could possibly be defined, does not make any prescription as to whether arrays or maps use the definite length or indefinite length encoding. I.e., there is no correlation between leaving the size of an array "open" in the spec and the fact that it is then interchanged with definite or indefinite length.

Please also note that CDDL can describe flexibility that the data model of the target representation does not have. This is rather obvious for JSON, but also is relevant for CBOR:

```
apartment = {
  kitchen: size,
  * bedroom: size,
}
size = float ; in m2
```

The previous specification does not mean that CBOR is changed to allow to use the key "bedroom" more than once. In other words, due to the restrictions imposed by the data model, the third line pretty much turns into:

```
? bedroom: size,
```

(Occurrence indicators beyond one still are useful in maps for groups that allow a variety of keys.)

3.3. Predefined names for types

CDDL predefines a number of names. This subsection summarizes these names, but please see Appendix D for the exact definitions.

The following keywords for primitive datatypes are defined:

"bool" Boolean value (major type 7, additional information 20 or 21).

"uint" An unsigned integer (major type 0).

"nint" A negative integer (major type 1).

"int" An unsigned integer or a negative integer.

"float16" A number representable as an IEEE 754 half-precision float (major type 7, additional information 25).

"float32" A number representable as an IEEE 754 single-precision float (major type 7, additional information 26).

"float64" A number representable as an IEEE 754 double-precision float (major type 7, additional information 27).

"float" One of float16, float32, or float64.

"bstr" or "bytes" A byte string (major type 2).

"tstr" or "text" Text string (major type 3)

(Note that there are no predefined names for arrays or maps; these are defined with the syntax given below.)

In addition, a number of types are defined in the prelude that are associated with CBOR tags, such as "tdate", "bigint", "regexp" etc.

3.4. Arrays

Array definitions surround a group with square brackets.

For each entry, an occurrence indicator as specified in Section 3.2 is permitted.

For example:

```
unlimited-people = [* person]
one-or-two-people = [1*2 person]
at-least-two-people = [2* person]
person = (
    name: tstr,
    age: uint,
)
```

The group "person" is defined in such a way that repeating it in the array each time generates alternating names and ages, so these are four valid values for a data item of type "unlimited-people":

```
["roundlet", 1047, "psychurgy", 2204, "extrarhythmical", 2231]
[]
["aluminize", 212, "climograph", 4124]
["penintime", 1513, "endocarditis", 4084, "impermeator", 1669,
 "coextension", 865]
```

3.5. Maps

The syntax for specifying maps merits special attention, as well as a number of optimizations and conveniences, as it is likely to be the focal point of many specifications employing CDDL. While the syntax does not strictly distinguish struct and table usage of maps, it caters specifically to each of them.

But first, let’s reiterate a feature of CBOR that it has inherited from JSON: The key/value pairs in CBOR maps have no fixed ordering. (One could imagine situations where fixing the ordering may be of use. For example, a decoder could look for values related with integer keys 1, 3 and 7. If the order were fixed and the decoder encounters the key 4 without having encountered key 3, it could conclude that key 3 is not available without doing more complicated bookkeeping. Unfortunately, neither JSON nor CBOR support this, so no attempt was made to support this in CDDL either.)

3.5.1. Structs

The "struct" usage of maps is similar to the way JSON objects are used in many JSON applications.

A map is defined in the same way as defining an array (see Section 3.4), except for using curly braces "{}" instead of square brackets "[]".

An occurrence indicator as specified in Section 3.2 is permitted for each group entry.

The following is an example of a record with a structure enbedded:

```
Geography = [
  city           : tstr,
  gpsCoordinates : GpsCoordinates,
]

GpsCoordinates = {
  longitude      : uint,            ; degrees, scaled by 10^7
  latitude       : uint,            ; degreed, scaled by 10^7
}
```

When encoding, the Geography record is encoded using a CBOR array with two members (the keys for the group entries are ignored), whereas the GpsCoordinates structure is encoded as a CBOR map with two key/value pairs.

Types used in a structure can be defined in separate rules or just in place (potentially placed inside parentheses, such as for choices). E.g.:

```
located-samples = {
  sample-point: int,
  samples: [+ float],
}
```

where "located-samples" is the datatype to be used when referring to the struct, and "sample-point" and "samples" are the keys to be used. This is actually a complete example: an identifier that is followed by a colon can be directly used as the text string for a member key (we speak of a "bareword" member key), as can a double-quoted string or a number. (When other types, in particular ones that contain more than one value, are used as the types of keys, they are followed by a double arrow, see below.)

If a text string key does not match the syntax for an identifier (or if the specifier just happens to prefer using double quotes), the text string syntax can also be used in the member key position, followed by a colon. The above example could therefore have been written with quoted strings in the member key positions.

More generally, types specified in other ways than the cases described above can be used in a keytype position by following them with a double arrow -- in particular, the double arrow is necessary if a type is named by an identifier (which, when followed by a colon, would be interpreted as a "bareword" and turned into a text string). A literal text string also gives rise to a type (which contains a single value only -- the given string), so another form for this example is:

```
located-samples = {
  "sample-point" => int,
  "samples" => [+ float],
}
```

See Section 3.5.4 below for how the colon shortcut described here also adds some implied semantics.

A better way to demonstrate the double-arrow use may be:

```
located-samples = {
  sample-point: int,
  samples: [+ float],
  * equipment-type => equipment-tolerances,
}
equipment-type = [name: tstr, manufacturer: tstr]
equipment-tolerances = [+ [float, float]]
```

The example below defines a struct with optional entries: display name (as a text string), the name components first name and family name (as text strings), and age information (as an unsigned integer).

```
PersonalData = {
  ? displayName: tstr,
  NameComponents,
  ? age: uint,
}

NameComponents = (
  ? firstName: tstr,
  ? familyName: tstr,
)
```

Note that the group definition for NameComponents does not generate another map; instead, all four keys are directly in the struct built by PersonalData.

In this example, all key/value pairs are optional from the perspective of CDDL. With no occurrence indicator, an entry is mandatory.

If the addition of more entries not specified by the current specification is desired, one can add this possibility explicitly:

```
PersonalData = {
  ? displayName: tstr,
  NameComponents,
  ? age: uint,
  * tstr => any
}

NameComponents = (
  ? firstName: tstr,
  ? familyName: tstr,
)
```

Figure 7: Personal Data: Example for extensibility

The CDDL tool reported on in Appendix F generated as one acceptable instance for this specification:

```
{"familyName": "agust", "antiforeignism": "pretzel",
 "springbuck": "illuminatingly", "exuviae": "ephemeris",
 "kilometrage": "frogfish"}
```

(See Section 3.9 for one way to explicitly identify an extension point.)

3.5.2. Tables

A table can be specified by defining a map with entries where the keytype allows more than just a single value, e.g.:

```
square-roots = {* x => y}
x = int
y = float
```

Here, the key in each key/value pair has datatype x (defined as int), and the value has datatype y (defined as float).

If the specification does not need to restrict one of x or y (i.e., the application is free to choose per entry), it can be replaced by the predefined name "any".

As another example, the following could be used as a conversion table converting from an integer or float to a string:

```
tostring = {* mynumber => tstr}
mynumber = int / float
```

3.5.3. Non-deterministic order

While the way arrays are matched is fully determined by the Parsing Expression Grammar (PEG) formalism (see Appendix A), matching is more complicated for maps, as maps do not have an inherent order. For each candidate name/value pair that the PEG algorithm would try, a matching member is picked out of the entire map. For certain group expressions, more than one member in the map may match. Most often, this is inconsequential, as the group expression tends to consume all matches:

```
labeled-values = {
  ? fritz: number,
  * label => value
}
label = text
value = number
```

Here, if any member with the key "fritz" is present, this will be picked by the first entry of the group; all remaining text/number member will be picked by the second entry (and if anything remains unpicked, the map does not match).

However, it is possible to construct group expressions where what is actually picked is indeterminate, and does matter:

```
do-not-do-this = {
  int => int,
  int => 6,
}
```

When this expression is matched against "{3: 5, 4: 6}", the first group entry might pick off the "3: 5", leaving "4: 6" for matching the second one. Or it might pick off "4: 6", leaving nothing for the second entry. This pathological non-determinism is caused by specifying more general before more specific, and by having a general rule that only consumes a subset of the map key/value pairs that it is able to match -- both tend not to occur in real-world

specifications of maps. At the time of writing, CDDL tools cannot detect such cases automatically, and for the present version of the CDDL specification, the specification writer is simply urged to not write pathologically non-deterministic specifications.

(The astute reader will be reminded of what was called "ambiguous content models" in SGML and "non-deterministic content models" in XML. That problem is related to the one described here, but the problem here is specifically caused by the lack of order in maps, something that the XML schema languages do not have to contend with. Note that Relax-NG's "interleave" pattern handles lack of order explicitly on the specification side, while the instances in XML always have determinate order.)

3.5.4. Cuts in Maps

The extensibility idiom discussed above for structs has one problem:

```
extensible-map-example = {
  ? "optional-key" => int,
  * tstr => any
}
```

In this example, there is one optional key "optional-key", which, when present, maps to an integer. There is also a wild card for any future additions.

Unfortunately, the data item

```
{ "optional-key": "nonsense" }
```

does match this specification: While the first entry of the group does not match, the second one (the wildcard) does. This may be very well desirable (e.g., if a future extension is to be allowed to extend the type of "optional-key"), but in many cases isn't.

In anticipation of a more general potential feature called "cuts", CDDL allows inserting a cut "^" into the definition of the map entry:

```
extensible-map-example = {
  ? "optional-key" ^ => int,
  * tstr => any
}
```

A cut in this position means that once the member key matches the name part of an entry that carries a cut, other potential matches for the key of the member that occur in later entries in the group of the map are no longer allowed. In other words, when a group entry would

pick a key/value pair based on just a matching key, it "locks in" the pick -- this rule applies independent of whether the value matches as well, so when it does not, the entire map fails to match. In summary, the example above no longer matches the specification as modified with the cut.

Since the desire for this kind of exclusive matching is so frequent, the ":" shortcut is actually defined to include the cut semantics. So the preceding example (including the cut) can be written more simply as:

```
extensible-map-example = {
  ? "optional-key": int,
  * tstr => any
}
```

or even shorter, using a bareword for the key:

```
extensible-map-example = {
  ? optional-key: int,
  * tstr => any
}
```

3.6. Tags

A type can make use of a CBOR tag (major type 6) by using the representation type notation, giving #6.nnn(type) where nnn is an unsigned integer giving the tag number and "type" is the type of the data item being tagged.

For example, the following line from the CDDL prelude (Appendix D) defines "biguint" as a type name for a positive bignum N:

```
biguint = #6.2(bstr)
```

The tags defined by [RFC7049] are included in the prelude. Additional tags since registered need to be added to a CDDL specification as needed; e.g., a binary UUID tag could be referenced as "buuid" in a specification after defining

```
buuid = #6.37(bstr)
```

In the following example, usage of the tag 32 for URIs is optional:

```
my_uri = #6.32(tstr) / tstr
```

3.7. Unwrapping

The group that is used to define a map or an array can often be reused in the definition of another map or array. Similarly, a type defined as a tag carries an internal data item that one would like to refer to. In these cases, it is expedient to simply use the name of the map, array, or tag type as a handle for the group or type defined inside it.

The "unwrap" operator (written by preceding a name by a tilde character "~") can be used to strip the type defined for a name by one layer, exposing the underlying group (for maps and arrays) or type (for tags).

For example, an application might want to define a basic and an advanced header. Without unwrapping, this might be done as follows:

```
basic-header-group = (
  field1: int,
  field2: text,
)

basic-header = [ basic-header-group ]

advanced-header = [
  basic-header-group,
  field3: bytes,
  field4: number, ; as in the tagged type "time"
]
```

Unwrapping simplifies this to:

```
basic-header = [
  field1: int,
  field2: text,
]

advanced-header = [
  ~basic-header,
  field3: bytes,
  field4: ~time,
]
```

(Note that leaving out the first unwrap operator in the latter example would lead to nesting the basic-header in its own array inside the advanced-header, while, with the unwrapped basic-header, the definition of the group inside basic-header is essentially repeated inside advanced-header, leading to a single array. This can

be used for various applications often solved by inheritance in programming languages. The effect of unwrapping can also be described as "threading in" the group or type inside the referenced type, which suggested the thread-like "~" character.)

3.8. Controls

A _control_ allows to relate a _target_ type with a _controller_ type via a _control operator_.

The syntax for a control type is "target .control-operator controller", where control operators are special identifiers prefixed by a dot. (Note that _target_ or _controller_ might need to be parenthesized.)

A number of control operators are defined at this point. Further control operators may be defined by new versions of this specification or by registering them according to the procedures in Section 6.1.

3.8.1. Control operator .size

A ".size" control controls the size of the target in bytes by the control type. The control is defined for text and byte strings, where it directly controls the number of bytes in the string. It is also defined for unsigned integers (see below). Figure 8 shows example usage for byte strings.

```
full-address = [[+ label], ip4, ip6]
ip4 = bstr .size 4
ip6 = bstr .size 16
label = bstr .size (1..63)
```

Figure 8: Control for size in bytes

When applied to an unsigned integer, the ".size" control restricts the range of that integer by giving a maximum number of bytes that should be needed in a computer representation of that unsigned integer. In other words, "uint .size N" is equivalent to "0...BYTES_N", where BYTES_N == 256**N.

```
audio_sample = uint .size 3 ; 24-bit, equivalent to 0...16777216
```

Figure 9: Control for integer size in bytes

Note that, as with value restrictions in CDDL, this control is not a representation constraint; a number that fits into fewer bytes can still be represented in that form, and an inefficient implementation

could use a longer form (unless that is restricted by some format constraints outside of CDDL, such as the rules in Section 3.9 of [RFC7049]).

3.8.2. Control operator .bits

A ".bits" control on a byte string indicates that, in the target, only the bits numbered by a number in the control type are allowed to be set. (Bits are counted the usual way, bit number "n" being set in "str" meaning that "(str[n >> 3] & (1 << (n & 7))) != 0".) Similarly, a ".bits" control on an unsigned integer "i" indicates that for all unsigned integers "n" where "(i & (1 << n)) != 0", "n" must be in the control type.

```
tcpflagbytes = bstr .bits flags
flags = &(
  fin: 8,
  syn: 9,
  rst: 10,
  psh: 11,
  ack: 12,
  urg: 13,
  ece: 14,
  cwr: 15,
  ns: 0,
) / (4..7) ; data offset bits

rwxbits = uint .bits rwx
rwx = &(r: 2, w: 1, x: 0)
```

Figure 10: Control for what bits can be set

The CDDL tool reported on in Appendix F generates the following ten example instances for "tcpflagbytes":

```
h'906d' h'01fc' h'8145' h'01b7' h'013d' h'409f' h'018e' h'c05f'
h'01fa' h'01fe'
```

These examples do not illustrate that the above CDDL specification does not explicitly specify a size of two bytes: A valid all clear instance of flag bytes could be "h''" or "h'00'" or even "h'000000'" as well.

3.8.3. Control operator .regexp

A ".regexp" control indicates that the text string given as a target needs to match the XSD regular expression given as a value in the

control type. XSD regular expressions are defined in Appendix F of [W3C.REC-xmlschema-2-20041028].

```
nai = tstr .regexp "[A-Za-z0-9]+@[A-Za-z0-9]+(\\.[A-Za-z0-9]+)+"
```

Figure 11: Control with an XSD regexp

An example matching this regular expression:

"N1@CH57HF.4Znqe0.dYJRN.igjf"

3.8.3.1. Usage considerations

Note that XSD regular expressions do not support the usual \x or \u escapes for hexadecimal expression of bytes or unicode code points. However, in CDDL the XSD regular expressions are contained in text strings, the literal notation for which provides \u escapes; this should suffice for most applications that use regular expressions for text strings. (Note that this also means that there is one level of string escaping before the XSD escaping rules are applied.)

XSD regular expressions support character class subtraction, a feature often not found in regular expression libraries; specification writers may want to use this feature sparingly. Similar considerations apply to Unicode character classes; where these are used, the specification that employs CDDL SHOULD identify which Unicode versions are addressed.

Other surprises for infrequent users of XSD regular expressions may include:

- No direct support for case insensitivity. While case insensitivity has gone mostly out of fashion in protocol design, it is sometimes needed and then needs to be expressed manually as in "[Cc][Aa][Ss][Ee]".

- The support for popular character classes such as \w and \d is based on Unicode character properties, which is often not what is desired in an ASCII-based protocol and thus might lead to surprises. (\s and \S do have their more conventional meanings, and "." matches any character but the line ending characters \r or \n.)

3.8.3.2. Discussion

There are many flavors of regular expression in use in the programming community. For instance, perl-compatible regular expressions (PCRE) are widely used and probably are more useful than

XSD regular expressions. However, there is no normative reference for PCRE that could be used in the present document. Instead, we opt for XSD regular expressions for now. There is precedent for that choice in the IETF, e.g., in YANG [RFC7950].

Note that CDDL uses controls as its main extension point. This creates the opportunity to add further regular expression formats in addition to the one referenced here if desired. As an example, a control ".pcre" is defined in [I-D.bormann-cbor-cddl-freezer].

3.8.4. Control operators .cbor and .cborseq

A ".cbor" control on a byte string indicates that the byte string carries a CBOR encoded data item. Decoded, the data item matches the type given as the right-hand side argument (type1 in the following example).

```
"bytes .cbor type1"
```

Similarly, a ".cborseq" control on a byte string indicates that the byte string carries a sequence of CBOR encoded data items. When the data items are taken as an array, the array matches the type given as the right-hand side argument (type2 in the following example).

```
"bytes .cborseq type2"
```

(The conversion of the encoded sequence to an array can be effected for instance by wrapping the byte string between the two bytes 0x9f and 0xff and decoding the wrapped byte string as a CBOR encoded data item.)

3.8.5. Control operators .within and .and

A ".and" control on a type indicates that the data item matches both that left hand side type and the type given as the right hand side. (Formally, the resulting type is the intersection of the two types given.)

```
"type1 .and type2"
```

A variant of the ".and" control is the ".within" control, which expresses an additional intent: the left hand side type is meant to be a subset of the right-hand-side type.

```
"type1 .within type2"
```

While both forms have the identical formal semantics (intersection), the intention of the ".within" form is that the right hand side gives

guidance to the types allowed on the left hand side, which typically is a socket (Section 3.9):

```
message = $message .within message-structure
message-structure = [message_type, *message_option]
message_type = 0..255
message_option = any

$message /= [3, dough: text, topping: [* text]]
$message /= [4, noodles: text, sauce: text, parmesan: bool]
```

For ".within", a tool might flag an error if type1 allows data items that are not allowed by type2. In contrast, for ".and", there is no expectation that type1 already is a subset of type2.

3.8.6. Control operators .lt, .le, .gt, .ge, .eq, .ne, and .default

The controls .lt, .le, .gt, .ge, .eq, .ne specify a constraint on the left hand side type to be a value less than, less than or equal, greater than, greater than or equal, equal, or not equal, to a value given as a right hand side type (containing just that single value). In the present specification, the first four controls (.lt, .le, .gt, .ge) are defined only for numeric types, as these have a natural ordering relationship.

```
speed = number .ge 0  ; unit: m/s
```

.ne and .eq are defined both for numeric values and values of other types. If one of the values is not of a numeric type, equality is determined as follows: Text strings are equal (satisfy .eq/do not satisfy .ne) if they are byte-wise identical; the same applies for byte strings. Arrays are equal if they have the same number of elements, all of which are equal pairwise in order between the arrays. Maps are equal if they have the same number of key/value pairs, and there is pairwise equality between the key/value pairs between the two maps. Tagged values are equal if they both have the same tag and the values are equal. Values of simple types match if they are the same values. Numeric types that occur within arrays, maps, or tagged values are equal if their numeric value is equal and they are both integers or both floating point values. All other cases are not equal (e.g., comparing a text string with a byte string).

A variant of the ".ne" control is the ".default" control, which expresses an additional intent: the value specified by the right-hand-side type is intended as a default value for the left hand side type given, and the implied .ne control is there to prevent this value from being sent over the wire. This control is only meaningful

when the control type is used in an optional context; otherwise there would be no way to make use of the default value.

```
timer = {
  time: uint,
  ? displayed-step: (number .gt 0) .default 1
}
```

3.9. Socket/Plug

Both for type choices and group choices, a mechanism is defined that facilitates starting out with empty choices and assembling them later, potentially in separate files that are concatenated to build the full specification.

Per convention, CDDL extension points are marked with a leading dollar sign (types) or two leading dollar signs (groups). Tools honor that convention by not raising an error if such a type or group is not defined at all; the symbol is then taken to be an empty type choice (group choice), i.e., no choice is available.

```
tcp-header = {seq: uint, ack: uint, * $$tcp-option}

; later, in a different file

$$tcp-option //= (
sack: [+(left: uint, right: uint)]
)

; and, maybe in another file

$$tcp-option //= (
sack-permitted: true
)
```

Names that start with a single "$" are "type sockets", starting out as an empty type, and intended to be extended via "/=". Names that start with a double "$$" are "group sockets", starting out as an empty group choice, and intended to be extended via "//=". In either case, it is not an error if there is no definition for a socket at all; this then means there is no way to satisfy the rule (i.e., the choice is empty).

As a convention, all definitions (plugs) for socket names must be augmentations, i.e., they must be using "/=" and "//=", respectively.

To pick up the example illustrated in Figure 7, the socket/plug mechanism could be used as shown in Figure 12:

```
PersonalData = {
  ? displayName: tstr,
  NameComponents,
  ? age: uint,
  * $$personaldata-extensions
}

NameComponents = (
  ? firstName: tstr,
  ? familyName: tstr,
)

; The above already works as is.
; But then, we can add later:

$$personaldata-extensions //= (
  favorite-salsa: tstr,
)

; and again, somewhere else:

$$personaldata-extensions //= (
  shoesize: uint,
)
```

Figure 12: Personal Data example: Using socket/plug extensibility

3.10. Generics

Using angle brackets, the left hand side of a rule can add formal parameters after the name being defined, as in:

```
messages = message<"reboot", "now"> / message<"sleep", 1..100>
message<t, v> = {type: t, value: v}
```

When using a generic rule, the formal parameters are bound to the actual arguments supplied (also using angle brackets), within the scope of the generic rule (as if there were a rule of the form parameter = argument).

Generic rules can be used for establishing names for both types and groups.

(There are some limitations to nesting of generics in the tool described in Appendix F at this time.)

3.11. Operator Precedence

As with any language that has multiple syntactic features such as prefix and infix operators, CDDL has operators that bind more tightly than others. This is becoming more complicated than, say, in ABNF, as CDDL has both types and groups, with operators that are specific to these concepts. Type operators (such as "/" for type choice) operate on types, while group operators (such as "//" for group choice) operate on groups. Types can simply be used in groups, but groups need to be bracketed (as arrays or maps) to become types. So, type operators naturally bind closer than group operators.

For instance, in

```
t = [group1]
group1 = (a / b // c / d)
a = 1 b = 2 c = 3 d = 4
```

group1 is a group choice between the type choice of a and b and the type choice of c and d. This becomes more relevant once member keys and/or occurrences are added in:

```
t = {group2}
group2 = (? ab: a / b // cd: c / d)
a = 1 b = 2 c = 3 d = 4
```

is a group choice between the optional member "ab" of type a or b and the member "cd" of type c or d. Note that the optionality is attached to the first choice ("ab"), not to the second choice.

Similarly, in

```
t = [group3]
group3 = (+ a / b / c)
a = 1 b = 2 c = 3
```

group3 is a repetition of a type choice between a, b, and c; if just a is to be repeatable, a group choice is needed to focus the occurrence:

(A comment has been that this could be counter-intuitive. The specification writer is encouraged to use parentheses liberally to guide readers that are not familiar with CDDL precedence rules.)

```
t = [group4]
group4 = (+ a // b / c)
a = 1 b = 2 c = 3
```

group4 is a group choice between a repeatable a and a single b or c.

In general, as with many other languages with operator precedence rules, it is best not to rely on them, but to insert parentheses for readability:

```
t = [group4a]
group4a = ((+ a) // (b / c))
a = 1 b = 2 c = 3
```

The operator precedences, in sequence of loose to tight binding, are defined in Appendix B and summarized in Table 1. (Arities given are 1 for unary prefix operators and 2 for binary infix operators.)

Operator	Ar	Operates on	Prec
=	2	name = type, name = group	1
/=	2	name /= type	1
//=	2	name //= group	1
//	2	group // group	2
,	2	group, group	3
*	1	* group	4
N*M	1	N*M group	4
+	1	+ group	4
?	1	? group	4
=>	2	type => type	5
:	2	name: type	5
/	2	type / type	6
..	2	type..type	7
...	2	type...type	7
.ctrl	2	type .ctrl type	7
&	1	&group	8
~	1	~type	8

Table 1: Summary of operator precedences

4. Making Use of CDDL

In this section, we discuss several potential ways to employ CDDL.

4.1. As a guide to a human user

CDDL can be used to efficiently define the layout of CBOR data, such that a human implementer can easily see how data is supposed to be encoded.

Since CDDL maps parts of the CBOR data to human readable names, tools could be built that use CDDL to provide a human friendly representation of the CBOR data, and allow them to edit such data while remaining compliant to its CDDL definition.

4.2. For automated checking of CBOR data structure

CDDL has been specified such that a machine can handle the CDDL definition and related CBOR data (and, thus, also JSON data). For example, a machine could use CDDL to check whether or not CBOR data is compliant to its definition.

The need for thoroughness of such compliance checking depends on the application. For example, an application may decide not to check the data structure at all, and use the CDDL definition solely as a means to indicate the structure of the data to the programmer.

On the other end, the application may also implement a checking mechanism that goes as far as checking that all mandatory map members are available.

The matter in how far the data description must be enforced by an application is left to the designers and implementers of that application, keeping in mind related security considerations.

In no case the intention is that a CDDL tool would be "writing code" for an implementation.

4.3. For data analysis tools

In the long run, it can be expected that more and more data will be stored using the CBOR data format.

Where there is data, there is data analysis and the need to process such data automatically. CDDL can be used for such automated data processing, allowing tools to verify data, clean it, and extract particular parts of interest from it.

Since CBOR is designed with constrained devices in mind, a likely use of it would be small sensors. An interesting use would thus be automated analysis of sensor data.

5. Security considerations

This document presents a content rules language for expressing CBOR data structures. As such, it does not bring any security issues on itself, although specifications of protocols that use CBOR naturally

need security analyses when defined. General guidelines for writing security considerations are defined in

Security Considerations Guidelines [RFC3552] (BCP 72).
Specifications using CDDL to define CBOR structures in protocols need to follow those guidelines. Additional topics that could be considered in a security considerations section for a specification that uses CDDL to define CBOR structures include the following:

- Where could the language maybe cause confusion in a way that will enable security issues?
- Where a CDDL matcher is part of the implementation of a system, the security of the system ought not depend on the correctness of the CDDL specification or CDDL implementation without any further defenses in place.
- Where the CDDL includes extension points, the impact of extensions on the security of the system needs to be carefully considered.

Writers of CDDL specifications are strongly encouraged to value clarity and transparency of the specification over its elegance. Keep it as simple as possible while still expressing the needed data model.

A related observation about formal description techniques in general that is strongly recommended to be kept in mind by writers of CDDL specifications: Just because CDDL makes it easier to handle complexity in a specification, that does not make that complexity somehow less bad (except maybe on the level of the humans having to grasp the complex structure while reading the spec).

6. IANA Considerations

6.1. CDDL control operator registry

IANA is requested to create a registry for control operators Section 3.8. The name of this registry is "CDDL Control Operators".

Each entry in the subregistry must include the name of the control operator (by convention given with the leading dot) and a reference to its documentation. Names must be composed of the leading dot followed by a text string conforming to the production "id" in Appendix B.

Initial entries in this registry are as follows:

name	documentation
.size	[RFCthis]
.bits	[RFCthis]
.regexp	[RFCthis]
.cbor	[RFCthis]
.cborseq	[RFCthis]
.within	[RFCthis]
.and	[RFCthis]
.lt	[RFCthis]
.le	[RFCthis]
.gt	[RFCthis]
.ge	[RFCthis]
.eq	[RFCthis]
.ne	[RFCthis]
.default	[RFCthis]

All other control operator names are Unassigned.

The IANA policy for additions to this registry is "Specification Required" as defined in [RFC8126] (which involves an Expert Review) for names that do not include an internal dot, and "IETF Review" for names that do include an internal dot. The Expert is specifically instructed that other Standards Development Organizations (SDOs) may want to define control operators that are specific to their fields (e.g., based on a binary syntax already in use at the SDO); the review process should strive to facilitate such an undertaking.

7. References

7.1. Normative References

[ISO6093] ISO, "Information processing -- Representation of numerical values in character strings for information interchange", ISO 6093, 1985.

[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>.

[RFC3552] Rescorla, E. and B. Korver, "Guidelines for Writing RFC Text on Security Considerations", BCP 72, RFC 3552, DOI 10.17487/RFC3552, July 2003, <https://www.rfc-editor.org/info/rfc3552>.

[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>.

[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>.

[RFC5234] Crocker, D., Ed. and P. Overell, "Augmented BNF for Syntax Specifications: ABNF", STD 68, RFC 5234, DOI 10.17487/RFC5234, January 2008, <https://www.rfc-editor.org/info/rfc5234>.

[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>.

[RFC7493] Bray, T., Ed., "The I-JSON Message Format", RFC 7493, DOI 10.17487/RFC7493, March 2015, <https://www.rfc-editor.org/info/rfc7493>.

[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>.

[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>.

[W3C.REC-xmlschema-2-20041028] Biron, P. and A. Malhotra, "XML Schema Part 2: Datatypes Second Edition", World Wide Web Consortium Recommendation REC-xmlschema-2-20041028, October 2004, <http://www.w3.org/TR/2004/REC-xmlschema-2-20041028>.

7.2. Informative References

[I-D.bormann-cbor-cddl-freezer] Bormann, C., "A feature freezer for the Concise Data Definition Language (CDDL)", draft-bormann-cbor-cddl-freezer-01 (work in progress), August 2018.

[I-D.ietf-anima-grasp]
Bormann, C., Carpenter, B., and B. Liu, "A Generic Autonomic Signaling Protocol (GRASP)", draft-ietf-anima-grasp-15 (work in progress), July 2017.

[I-D.newton-json-content-rules]
Newton, A. and P. Cordell, "A Language for Rules Describing JSON Content", draft-newton-json-content-rules-09 (work in progress), September 2017.

[PEG] Ford, B., "Parsing expression grammars", Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL '04, DOI 10.1145/964001.964011, 2004.

[RELAXNG] ISO/IEC, "Information technology -- Document Schema Definition Language (DSDL) -- Part 2: Regular-grammar-based validation -- RELAX NG", ISO/IEC 19757-2, December 2008.

[RFC7071] Borenstein, N. and M. Kucherawy, "A Media Type for Reputation Interchange", RFC 7071, DOI 10.17487/RFC7071, November 2013, <https://www.rfc-editor.org/info/rfc7071>.

[RFC7950] Bjorklund, M., Ed., "The YANG 1.1 Data Modeling Language", RFC 7950, DOI 10.17487/RFC7950, August 2016, <https://www.rfc-editor.org/info/rfc7950>.

[RFC8007] Murray, R. and B. Niven-Jenkins, "Content Delivery Network Interconnection (CDNI) Control Interface / Triggers", RFC 8007, DOI 10.17487/RFC8007, December 2016, <https://www.rfc-editor.org/info/rfc8007>.

[RFC8152] Schaad, J., "CBOR Object Signing and Encryption (COSE)", RFC 8152, DOI 10.17487/RFC8152, July 2017, <https://www.rfc-editor.org/info/rfc8152>.

[RFC8428] Jennings, C., Shelby, Z., Arkko, J., Keranen, A., and C. Bormann, "Sensor Measurement Lists (SenML)", RFC 8428, DOI 10.17487/RFC8428, August 2018, <https://www.rfc-editor.org/info/rfc8428>.

7.3. URIs

[1] https://github.com/cabo/cbor-diag

Appendix A. Parsing Expression Grammars (PEG)

This appendix is normative.

Since the 1950s, many grammar notations are based on Backus-Naur Form (BNF), a notation for context-free grammars (CFGs) within Chomsky's generative system of grammars. ABNF [RFC5234], the Augmented Backus-Naur Form widely used in IETF specifications and also inspiring the syntax of CDDL, is an example of this.

Generative grammars can express ambiguity well, but this very property may make them hard to use in recognition systems, spawning a number of subdialects that pose constraints on generative grammars to be used with parser generators, which may be hard to manage for the specification writer.

Parsing Expression Grammars [PEG] provide an alternative formal foundation for describing grammars that emphasizes recognition over generation, and resolves what would have been ambiguity in generative systems by introducing the concept of "prioritized choice".

The notation for Parsing Expression Grammars is quite close to BNF, with the usual "Extended BNF" features such as repetition added. However, where BNF uses the unordered (symmetrical) choice operator "|" (incidentally notated as "/" in ABNF), PEG provides a prioritized choice operator "/". The two alternatives listed are to be tested in left-to-right order, locking in the first successful match and disregarding any further potential matches within the choice (but not disabling alternatives in choices containing this choice, as a "cut" would - Section 3.5.4}.

For example, the ABNF expressions

```
A = "a" "b" / "a"    (1)
```

and

```
A = "a" / "a" "b"    (2)
```

are equivalent in ABNF's original generative framework, but very different in PEG: In (2), the second alternative will never match, as any input string starting with an "a" will already succeed in the first alternative, locking in the match.

Similarly, the occurrence indicators ("?", "*", "+") are "greedy" in PEG, i.e., they consume as much input as they match (and, as a consequence, "a* a" in PEG notation or "*a a" in CDDL syntax never

can match anything as all input matching "a" is already consumed by the initial "a*", leaving nothing to match the second "a").

Incidentally, the grammar of the CDDL language itself, as written in ABNF in Appendix B, can be interpreted both in the generative framework on which RFC 5234 is based, and as a PEG. This was made possible by ordering the choices in the grammar such that a successful match made on the left hand side of a "/" operator is always the intended match, instead of relying on the power of symmetrical choices (for example, note the sequence of alternatives in the rule for "uint", where the lone zero is behind the longer match alternatives that start with a zero).

The syntax used for expressing the PEG component of CDDL is based on ABNF, interpreted in the obvious way with PEG semantics. The ABNF convention of notating occurrence indicators before the controlled primary, and of allowing numeric values for minimum and maximum occurrence around a "*" sign, is copied. While PEG is only about characters, CDDL has a richer set of elements, such as types and groups. Specifically, the following constructs map:

CDDL	PEG	Remark
"="	"<-"	/= and //= are abbreviations
"//"	"/"	prioritized choice
"/"	"/"	prioritized choice, limited to types only
"?" P	P "?"	zero or one
"*" P	P "*"	zero or more
"+" P	P "+"	one or more
A B	A B	sequence
A, B	A B	sequence, comma is decoration only

The literal notation and the use of square brackets, curly braces, tildes, ampersands, and hash marks is specific to CDDL and unrelated to the conventional PEG notation. The DOT (".") is replaced by the unadorned "#" or its alias "any". Also, CDDL does not provide the syntactic predicate operators NOT ("!") or AND ("&") from PEG, reducing expressiveness as well as complexity.

For more details about PEG's theoretical foundation and interesting properties of the operators such as associativity and distributivity, the reader is referred to [PEG].

Appendix B. ABNF grammar

This appendix is normative.

The following is a formal definition of the CDDL syntax in Augmented Backus-Naur Form (ABNF, [RFC5234]). Note that, as is defined in ABNF, the quote-delimited strings below are case-insensitive (while string values and names are case-sensitive in CDDL).

```
  cddl = S 1*(rule S)
  rule = typename [genericparm] S assignt S type
       / groupname [genericparm] S assigng S grpent

  typename = id
  groupname = id

  assignt = "=" / "/="
  assigng = "=" / "//="

  genericparm = "<" S id S *("," S id S ) ">"
  genericarg = "<" S type1 S *("," S type1 S ) ">"

  type = type1 *(S "/" S type1)

  type1 = type2 [S (rangeop / ctlop) S type2]
  ; space may be needed before the operator if type2 ends in a name

  type2 = value
        / typename [genericarg]
        / "(" S type S ")"
        / "{" S group S "}"
        / "[" S group S "]"
        / "~" S typename [genericarg]
        / "&" S "(" S group S ")"
        / "&" S groupname [genericarg]
        / "#" "6" ["." uint] "(" S type S ")"
        / "#" DIGIT ["." uint]                ; major/ai
        / "#"                                 ; any

  rangeop = "..." / ".."

  ctlop = "." id

  group = grpchoice *(S "//" S grpchoice)

  grpchoice = *(grpent optcom)

  grpent = [occur S] [memberkey S] type
```

```
        / [occur S] groupname [genericarg]  ; preempted by above
        / [occur S] "(" S group S ")"

memberkey = type1 S ["^" S] "=>"
          / bareword S ":"
          / value S ":"

bareword = id

optcom = S ["," S]

occur = [uint] "*" [uint]
      / "+"
      / "?"

uint = DIGIT1 *DIGIT
     / "0x" 1*HEXDIG
     / "0b" 1*BINDIG
     / "0"

value = number
      / text
      / bytes

int = ["-"] uint

; This is a float if it has fraction or exponent; int otherwise
number = hexfloat / (int ["." fraction] ["e" exponent ])
hexfloat = "0x" 1*HEXDIG ["." 1*HEXDIG] "p" exponent
fraction = 1*DIGIT
exponent = ["+"/"-"] 1*DIGIT

text = %x22 *SCHAR %x22
SCHAR = %x20-21 / %x23-5B / %x5D-7E / %x80-10FFFD / SESC
SESC = "\" (%x20-7E / %x80-10FFFD)

bytes = [bsqual] %x27 *BCHAR %x27
BCHAR = %x20-26 / %x28-5B / %x5D-10FFFD / SESC / CRLF
bsqual = "h" / "b64"

id = EALPHA *(*("-" / ".") (EALPHA / DIGIT))
ALPHA = %x41-5A / %x61-7A
EALPHA = ALPHA / "@" / "_" / "$"
DIGIT = %x30-39
DIGIT1 = %x31-39
HEXDIG = DIGIT / "A" / "B" / "C" / "D" / "E" / "F"
BINDIG = %x30-31
```

```
S = *WS
WS = SP / NL
SP = %x20
NL = COMMENT / CRLF
COMMENT = ";" *PCHAR CRLF
PCHAR = %x20-7E / %x80-10FFFD
CRLF = %x0A / %x0D.0A
```

Figure 13: CDDL ABNF

Note that this ABNF does not attempt to reflect the detailed rules of what can be in a prefixed byte string.

Appendix C. Matching rules

This appendix is normative.

In this appendix, we go through the ABNF syntax rules defined in Appendix B and briefly describe the matching semantics of each syntactic feature. In this context, an instance (data item) "matches" a CDDL specification if it is allowed by the CDDL specification; this is then broken down to parts of specifications (type and group expressions) and parts of instances (data items).

```
cddl = S 1*(rule S)
```

A CDDL specification is a sequence of one or more rules. Each rule gives a name to a right hand side expression, either a CDDL type or a CDDL group. Rule names can be used in the rule itself and/or other rules (and tools can output warnings if that is not the case). The order of the rules is significant only in two cases:

1. The first rule defines the semantics of the entire specification; hence, there is no need to give that root rule a special name or special syntax in the language (as, e.g., with "start" in Relax-NG); its name can be therefore chosen to be descriptive. (As with all other rule names, the name of the initial rule may be used in itself or in other rules).

2. Where a rule contributes to a type or group choice (using "/=" or "//="), that choice is populated in the order the rules are given; see below.

```
rule = typename [genericparm] S assignt S type
     / groupname [genericparm] S assigng S grpent

typename = id
groupname = id
```

A rule defines a name for a type expression (production "type") or for a group expression (production "grpent"), with the intention that the semantics does not change when the name is replaced by its (parenthesized if needed) definition. Note that whether the name defined by a rule stands for a type or a group isn't always determined by syntax alone: e.g., "a = b" can make "a" a type if "b" is a type, or a group if "b" is a group. More subtly, in "a = (b)", "a" may be used as a type if "b" is a type, or as a group both when "b" is a group and when "b" is a type (a good convention to make the latter case stand out to the human reader is to write "a = (b,)"). (Note that the same dual meaning of parentheses applies within an expression, but often can be resolved by the context of the parenthesized expression. On the more general point, it may not be clear immediately either whether "b" stands for a group or a type -- this semantic processing may need to span several levels of rule definitions before a determination can be made.)

```
assignt = "=" / "/="
assigng = "=" / "//="
```

A plain equals sign defines the rule name as the equivalent of the expression to the right; it is an error if the name already was defined with a different expression. A "/=" or "//=" extends a named type or a group by additional choices; a number of these could be replaced by collecting all the right hand sides and creating a single rule with a type choice or a group choice built from the right hand sides in the order of the rules given. (It is not an error to extend a rule name that has not yet been defined; this makes the right hand side the first entry in the choice being created.)

```
genericparm = "<" S id S *("," S id S ) ">"
genericarg = "<" S type1 S *("," S type1 S ) ">"
```

Rule names can have generic parameters, which cause temporary assignments within the right hand sides to the parameter names from the arguments given when citing the rule name.

```
type = type1 *(S "/" S type1)
```

A type can be given as a choice between one or more types. The choice matches a data item if the data item matches any one of the types given in the choice. The choice uses Parsing Expression Grammar semantics as discussed in Appendix A: The first choice that matches wins. (As a result, the order of rules that contribute to a single rule name can very well matter.)

```
type1 = type2 [S (rangeop / ctlop) S type2]
```

Two types can be combined with a range operator (which see below) or a control operator (see Section 3.8).

type2 = value

A type can be just a single value (such as 1 or "icecream" or h'0815'), which matches only a data item with that specific value (no conversions defined),

/ typename [genericarg]

or be defined by a rule giving a meaning to a name (possibly after supplying generic arguments as required by the generic parameters),

/ "(" S type S ")"

or be defined in a parenthesized type expression (parentheses may be necessary to override some operator precedence), or

/ "{" S group S "}"

a map expression, which matches a valid CBOR map the key/value pairs of which can be ordered in such a way that the resulting sequence matches the group expression, or

/ "[" S group S "]"

an array expression, which matches a CBOR array the elements of which, when taken as values and complemented by a wildcard (matches anything) key each, match the group, or

/ "~" S typename [genericarg]

an "unwrapped" group (see Section 3.7), which matches the group inside a type defined as a map or an array by wrapping the group, or

/ "&" S "(" S group S ")"
/ "&" S groupname [genericarg]

an enumeration expression, which matches any a value that is within the set of values that the values of the group given can take, or

/ "#" "6" ["." uint] "(" S type S ")"

a tagged data item, tagged with the "uint" given and containing the type given as the tagged value, or

/ "#" DIGIT ["." uint] ; major/ai

a data item of a major type (given by the DIGIT), optionally constrained to the additional information given by the uint, or

```
      / "#"                                  ; any
```

any data item.

```
rangeop = "..." / ".."
```

A range operator can be used to join two type expressions that stand for either two integer values or two floating point values; it matches any value that is between the two values, where the first value is always included in the matching set and the second value is included for ".." and excluded for "...".

```
ctlop = "." id
```

A control operator ties a _target_ type to a _controller_ type as defined in Section 3.8. Note that control operators are an extension point for CDDL; additional documents may want to define additional control operators.

```
group = grpchoice *(S "//" S grpchoice)
```

A group matches any sequence of key/value pairs that matches any of the choices given (again using Parsing Expression Grammar semantics).

```
grpchoice = *(grpent optcom)
```

Each of the component groups is given as a sequence of group entries. For a match, the sequence of key/value pairs given needs to match the sequence of group entries in the sequence given.

```
grpent = [occur S] [memberkey S] type
```

A group entry can be given by a value type, which needs to be matched by the value part of a single element, and optionally a memberkey type, which needs to be matched by the key part of the element, if the memberkey is given. If the memberkey is not given, the entry can only be used for matching arrays, not for maps. (See below how that is modified by the occurrence indicator.)

```
       / [occur S] groupname [genericarg]  ; preempted by above
```

A group entry can be built from a named group, or

```
       / [occur S] "(" S group S ")"
```

from a parenthesized group, again with a possible occurrence indicator.

```
memberkey = type1 S ["^" S] "=>"
          / bareword S ":"
          / value S ":"
```

Key types can be given by a type expression, a bareword (which stands for a type that just contains a string value created from this bareword), or a value (which stands for a type that just contains this value). A key value matches its key type if the key value is a member of the key type, unless a cut preceding it in the group applies (see Section 3.5.4 how map matching is influenced by the presence of the cuts denoted by "^" or ":" in previous entries).

```
bareword = id
```

A bareword is an alternative way to write a type with a single text string value; it can only be used in the syntactic context given above.

```
optcom = S ["," S]
```

(Optional commas do not influence the matching.)

```
occur = [uint] "*" [uint]
      / "+"
      / "?"
```

An occurrence indicator modifies the group given to its right by requiring the group to match the sequence to be matched exactly for a certain number of times (see Section 3.2) in sequence, i.e. it acts as a (possibly infinite) group choice that contains choices with the group repeated each of the occurrences times.

The rest of the ABNF describes syntax for value notation that should be familiar from programming languages, with the possible exception of h'..' and b64'..' for byte strings, as well as syntactic elements such as comments and line ends.

Appendix D. Standard Prelude

This appendix is normative.

The following prelude is automatically added to each CDDL file. (Note that technically, it is a postlude, as it does not disturb the selection of the first rule as the root of the definition.)

```
any = #

uint = #0
nint = #1
int = uint / nint

bstr = #2
bytes = bstr
tstr = #3
text = tstr

tdate = #6.0(tstr)
time = #6.1(number)
number = int / float
biguint = #6.2(bstr)
bignint = #6.3(bstr)
bigint = biguint / bignint
integer = int / bigint
unsigned = uint / biguint
decfrac = #6.4([e10: int, m: integer])
bigfloat = #6.5([e2: int, m: integer])
eb64url = #6.21(any)
eb64legacy = #6.22(any)
eb16 = #6.23(any)
encoded-cbor = #6.24(bstr)
uri = #6.32(tstr)
b64url = #6.33(tstr)
b64legacy = #6.34(tstr)
regexp = #6.35(tstr)
mime-message = #6.36(tstr)
cbor-any = #6.55799(any)

float16 = #7.25
float32 = #7.26
float64 = #7.27
float16-32 = float16 / float32
float32-64 = float32 / float64
float = float16-32 / float64

false = #7.20
true = #7.21
bool = false / true
nil = #7.22
null = nil
undefined = #7.23
```

Figure 14: CDDL Prelude

Note that the prelude is deemed to be fixed. This means, for instance, that additional tags beyond [RFC7049], as registered, need to be defined in each CDDL file that is using them.

A common stumbling point is that the prelude does not define a type "string". CBOR has byte strings ("bytes" in the prelude) and text strings ("text"), so a type that is simply called "string" would be ambiguous.

Appendix E. Use with JSON

This appendix is normative.

The JSON generic data model (implicit in [RFC8259]) is a subset of the generic data model of CBOR. So one can use CDDL with JSON by limiting oneself to what can be represented in JSON. Roughly speaking, this means leaving out byte strings, tags, and simple values other than "false", "true", and "null", leading to the following limited prelude:

```
any = #

uint = #0
nint = #1
int = uint / nint

tstr = #3
text = tstr

number = int / float

float16 = #7.25
float32 = #7.26
float64 = #7.27
float16-32 = float16 / float32
float32-64 = float32 / float64
float = float16-32 / float64

false = #7.20
true = #7.21
bool = false / true
nil = #7.22
null = nil
```

Figure 15: JSON compatible subset of CDDL Prelude

(The major types given here do not have a direct meaning in JSON, but they can be interpreted as CBOR major types translated through Section 4 of [RFC7049].)

There are a few fine points in using CDDL with JSON. First, JSON does not distinguish between integers and floating point numbers; there is only one kind of number (which may happen to be integral). In this context, specifying a type as "uint", "nint" or "int" then becomes a predicate that the number be integral. As an example, this means that the following JSON numbers are all matching "uint":

10 10.0 1e1 1.0e1 100e-1

(The fact that these are all integers may be surprising to users accustomed to the long tradition in programming languages of using decimal points or exponents in a number to indicate a floating point literal.)

CDDL distinguishes the various CBOR number types, but there is only one number type in JSON. The effect of specifying a floating point precision (float16/float32/float64) is only to restrict the set of permissible values to those expressible with binary16/binary32/ binary64; this is unlikely to be very useful when using CDDL for specifying JSON data structures.

Fundamentally, the number system of JSON itself is based on decimal numbers and decimal fractions and does not have limits to its precision or range. In practice, JSON numbers are often parsed into a number type that is called float64 here, creating a number of limitations to the generic data model [RFC7493]. In particular, this means that integers can only be expressed with interoperable exactness when they lie in the range [-(2**53)+1, (2**53)-1] -- a smaller range than that covered by CDDL "int".

JSON applications that want to stay compatible with I-JSON ([RFC7493], "Internet JSON") therefore may want to define integer types with more limited ranges, such as in Figure 16. Note that the types given here are not part of the prelude; they need to be copied into the CDDL specification if needed.

```
ij-uint = 0..9007199254740991
ij-nint = -9007199254740991..-1
ij-int = -9007199254740991..9007199254740991
```

Figure 16: I-JSON types for CDDL (not part of prelude)

JSON applications that do not need to stay compatible with I-JSON and that actually may need to go beyond the 64-bit unsigned and negative

integers supported by "int" (= "uint"/"nint") may want to use the following additional types from the standard prelude, which are expressed in terms of tags but can straightforwardly be mapped into JSON (but not I-JSON) numbers:

```
biguint = #6.2(bstr)
bignint = #6.3(bstr)
bigint = biguint / bignint
integer = int / bigint
unsigned = uint / biguint
```

CDDL at this point does not have a way to express the unlimited floating point precision that is theoretically possible with JSON; at the time of writing, this is rarely used in protocols in practice.

Note that a data model described in CDDL is always restricted by what can be expressed in the serialization; e.g., floating point values such as NaN (not a number) and the infinities cannot be represented in JSON even if they are allowed in the CDDL generic data model.

Appendix F. A CDDL tool

This appendix is for information only.

A rough CDDL tool is available. For CDDL specifications, it can check the syntax, generate one or more instances (expressed in CBOR diagnostic notation or in pretty-printed JSON), and validate an existing instance against the specification:

```
Usage:
cddl spec.cddl generate [n]
cddl spec.cddl json-generate [n]
cddl spec.cddl validate instance.cbor
cddl spec.cddl validate instance.json
```

Figure 17: CDDL tool usage

Install on a system with a modern Ruby via:

```
gem install cddl
```

Figure 18: CDDL tool installation

The accompanying CBOR diagnostic tools (which are automatically installed by the above) are described in https://github.com/cabo/cbor-diag [1]; they can be used to convert between binary CBOR, a pretty-printed form of that, CBOR diagnostic notation, JSON, and YAML.

Appendix G. Extended Diagnostic Notation

This appendix is normative.

Section 6 of [RFC7049] defines a "diagnostic notation" in order to be able to converse about CBOR data items without having to resort to binary data. Diagnostic notation is based on JSON, with extensions for representing CBOR constructs such as binary data and tags.

(Standardizing this together with the actual interchange format does not serve to create another interchange format, but enables the use of a shared diagnostic notation in tools for and documents about CBOR.)

This section discusses a few extensions to the diagnostic notation that have turned out to be useful since RFC 7049 was written. We refer to the result as extended diagnostic notation (EDN).

G.1. White space in byte string notation

Examples often benefit from some white space (spaces, line breaks) in byte strings. In extended diagnostic notation, white space is ignored in prefixed byte strings; for instance, the following are equivalent:

```
h'48656c6c6f20776f726c64'
h'48 65 6c 6c 6f 20 77 6f 72 6c 64'
h'4 86 56c 6c6f
  20776 f726c64'
```

G.2. Text in byte string notation

Diagnostic notation notates Byte strings in one of the [RFC4648] base encodings,, enclosed in single quotes, prefixed by >h< for base16, >b32< for base32, >h32< for base32hex, >b64< for base64 or base64url. Quite often, byte strings carry bytes that are meaningfully interpreted as UTF-8 text. Extended Diagnostic Notation allows the use of single quotes without a prefix to express byte strings with UTF-8 text; for instance, the following are equivalent:

```
'hello world'
h'68656c6c6f20776f726c64'
```

The escaping rules of JSON strings are applied equivalently for text-based byte strings, e.g., \ stands for a single backslash and ' stands for a single quote. White space is included literally, i.e., the previous section does not apply to text-based byte strings.

G.3. Embedded CBOR and CBOR sequences in byte strings

Where a byte string is to carry an embedded CBOR-encoded item, or more generally a sequence of zero or more such items, the diagnostic notation for these zero or more CBOR data items, separated by commata, can be enclosed in << and >> to notate the byte string resulting from encoding the data items and concatenating the result. For instance, each pair of columns in the following are equivalent:

```
<<1>>              h'01'
<<1, 2>>           h'0102'
<<"foo", null>>    h'63666F6FF6'
<<>>               h''
```

G.4. Concatenated Strings

While the ability to include white space enables line-breaking of encoded byte strings, a mechanism is needed to be able to include text strings as well as byte strings in direct UTF-8 representation into line-based documents (such as RFCs and source code).

We extend the diagnostic notation by allowing multiple text strings or multiple byte strings to be notated separated by white space, these are then concatenated into a single text or byte string, respectively. Text strings and byte strings do not mix within such a concatenation, except that byte string notation can be used inside a sequence of concatenated text string notation to encode characters that may be better represented in an encoded way. The following four values are equivalent:

```
"Hello world"
"Hello " "world"
"Hello" h'20' "world"
"" h'48656c6c6f20776f726c64' ""
```

Similarly, the following byte string values are equivalent

```
'Hello world'
'Hello ' 'world'
'Hello ' h'776f726c64'
'Hello' h'20' 'world'
'' h'48656c6c6f20776f726c64' '' b64''
h'4 86 56c 6c6f' h' 20776 f726c64'
```

(Note that the approach of separating by whitespace, while familiar from the C language, requires some attention - a single comma makes a big difference here.)

G.5. Hexadecimal, octal, and binary numbers

In addition to JSON's decimal numbers, EDN provides hexadecimal, octal and binary numbers in the usual C-language notation (octal with 0o prefix present only).

The following are equivalent:

```
4711
0x1267
0o11147
0b1001001100111
```

As are:

```
1.5
0x1.8p0
0x18p-4
```

G.6. Comments

Longer pieces of diagnostic notation may benefit from comments. JSON famously does not provide for comments, and basic RFC 7049 diagnostic notation inherits this property.

In extended diagnostic notation, comments can be included, delimited by slashes ("/"). Any text within and including a pair of slashes is considered a comment.

Comments are considered white space. Hence, they are allowed in prefixed byte strings; for instance, the following are equivalent:

```
h'68656c6c6f20776f726c64'
h'68 65 6c /doubled l!/ 6c 6f /hello/
  20 /space/
  77 6f 72 6c 64' /world/
```

This can be used to annotate a CBOR structure as in:

```
/grasp-message/ [/M_DISCOVERY/ 1, /session-id/ 10584416,
                 /objective/ [/objective-name/ "opsonize",
                              /D, N, S/ 7, /loop-count/ 105]]
```

(There are currently no end-of-line comments. If we want to add them, "//" sounds like a reasonable delimiter given that we already use slashes for comments, but we also could go e.g. for "#".)

Appendix H. Examples

This appendix is for information only.

This section contains a few examples of structures defined using CDDL.

The theme for the first example is taken from [RFC7071], which defines certain JSON structures in English. For a similar example, it may also be of interest to examine Appendix A of [RFC8007], which contains a CDDL definition for a JSON structure defined in the main body of the RFC.

The second subsection in this appendix translates examples from [I-D.newton-json-content-rules] into CDDL.

These examples all happen to describe data that is interchanged in JSON. Examples for CDDL definitions of data that is interchanged in CBOR can be found in [RFC8152], [I-D.ietf-anima-grasp], or [RFC8428].

H.1. RFC 7071

[RFC7071] defines the Reputon structure for JSON using somewhat formalized English text. Here is a (somewhat verbose) equivalent definition using the same terms, but notated in CDDL:

```
reputation-object = {
  reputation-context,
  reputon-list
}

reputation-context = (
  application: text
)

reputon-list = (
  reputons: reputon-array
)

reputon-array = [* reputon]

reputon = {
  rater-value,
  assertion-value,
  rated-value,
  rating-value,
  ? conf-value,
  ? normal-value,
  ? sample-value,
  ? gen-value,
  ? expire-value,
  * ext-value,
}

rater-value = ( rater: text )
assertion-value = ( assertion: text )
rated-value = ( rated: text )
rating-value = ( rating: float16 )
conf-value = ( confidence: float16 )
normal-value = ( normal-rating: float16 )
sample-value = ( sample-size: uint )
gen-value = ( generated: uint )
expire-value = ( expires: uint )
ext-value = ( text => any )
```

An equivalent, more compact form of this example would be:

```
reputation-object = {
  application: text
  reputons: [* reputon]
}

reputon = {
  rater: text
  assertion: text
  rated: text
  rating: float16
  ? confidence: float16
  ? normal-rating: float16
  ? sample-size: uint
  ? generated: uint
  ? expires: uint
  * text => any
}
```

Note how this rather clearly delineates the structure somewhat shrouded by so many words in section 6.2.2. of [RFC7071]. Also, this definition makes it clear that several ext-values are allowed (by definition with different member names); RFC 7071 could be read to forbid the repetition of ext-value ("A specific reputon-element MUST NOT appear more than once" is ambiguous.)

The CDDL tool reported on in Appendix F generates as one example:

```
{
  "application": "conchometry",
  "reputons": [
    {
      "rater": "Ephthianura",
      "assertion": "codding",
      "rated": "sphaerolitic",
      "rating": 0.34133473256800795,
      "confidence": 0.9481983064298332,
      "expires": 1568,
      "unplaster": "grassy"
    },
    {
      "rater": "nonchargeable",
      "assertion": "raglan",
      "rated": "alienage",
      "rating": 0.5724646875815566,
      "sample-size": 3514,
      "Aldebaran": "unchurched",
      "puruloid": "impersonable",
      "uninfracted": "pericarpoidal",
      "schorl": "Caro"
    },
    {
      "rater": "precollectable",
      "assertion": "Merat",
      "rated": "thermonatrite",
      "rating": 0.19164006323936977,
      "confidence": 0.6065252103391268,
      "normal-rating": 0.5187773690879303,
      "generated": 899,
      "speedy": "solidungular",
      "noviceship": "medicine",
      "checkrow": "epidictic"
    }
  ]
}
```

H.2. Examples from JSON Content Rules

Although JSON Content Rules [I-D.newton-json-content-rules] seems to address a more general problem than CDDL, it is still a worthwhile resource to explore for examples (beyond all the inspiration the format itself has had for CDDL).

Figure 2 of the JCR I-D looks very similar, if slightly less noisy, in CDDL:

```
root = [2*2 {
  precision: text,
  Latitude: float,
  Longitude: float,
  Address: text,
  City: text,
  State: text,
  Zip: text,
  Country: text
}]
```

Figure 19: JCR, Figure 2, in CDDL

Apart from the lack of a need to quote the member names, text strings are called "text" or "tstr" in CDDL ("string" would be ambiguous as CBOR also provides byte strings).

The CDDL tool reported on in Appendix F creates the below example instance for this:

```
 [{"precision": "pyrosphere", "Latitude": 0.5399712314350172,
   "Longitude": 0.5157523963028087, "Address": "resow",
   "City": "problemwise", "State": "martyrlike", "Zip": "preprove",
   "Country": "Pace"},
  {"precision": "unrigging", "Latitude": 0.10422704368372193,
   "Longitude": 0.6279808663725834, "Address": "picturedom",
   "City": "decipherability", "State": "autometry", "Zip": "pout",
   "Country": "wimple"}]
```

Figure 4 of the JCR I-D in CDDL:

```
root = { image }

image = (
  Image: {
    size,
    Title: text,
    thumbnail,
    IDs: [* int]
  }
)

size = (
  Width: 0..1280
  Height: 0..1024
)

thumbnail = (
  Thumbnail: {
    size,
    Url: ~uri
  }
)
```

This shows how the group concept can be used to keep related elements (here: width, height) together, and to emulate the JCR style of specification. (It also shows referencing a type by unwrapping a tag from the prelude, "uri" - this could be done differently.) The more compact form of Figure 5 of the JCR I-D could be emulated like this:

```
root = {
  Image: {
    size, Title: text,
    Thumbnail: { size, Url: ~uri },
    IDs: [* int]
  }
}

size = (
  Width: 0..1280,
  Height: 0..1024,
)
```

The CDDL tool reported on in Appendix F creates the below example instance for this:

```
{"Image": {"Width": 566, "Height": 516, "Title": "leisterer",
  "Thumbnail": {"Width": 1111, "Height": 176, "Url": 32("scrog")},
  "IDs": []}}
```

Contributors

CDDL was originally conceived by Bert Greevenbosch, who also wrote the original five versions of this document.

Acknowledgements

Inspiration was taken from the C and Pascal languages, MPEG's conventions for describing structures in the ISO base media file format, Relax-NG and its compact syntax [RELAXNG], and in particular from Andrew Lee Newton's "JSON Content Rules" [I-D.newton-json-content-rules].

Lots of highly useful feedback came from members of the IETF CBOR WG, in particular Ari Keraenen, Brian Carpenter, Burt Harris, Jeffrey Yasskin, Jim Hague, Jim Schaad, Joe Hildebrand, Max Pritikin, Michael Richardson, Pete Cordell, Sean Leonard, and Yaron Sheffer. Also, Francesca Palombini and Joe volunteered to chair the WG when it was created, providing the framework for generating and processing this feedback; with Barry Leiba having taken over from Joe since. Chris Lonvick and Ines Robles provided additional reviews during IESG processing, and Alexey Melnikov steered the process as the responsible area director.

The CDDL tool reported on in Appendix F was written by Carsten Bormann, building on previous work by Troy Heninger and Tom Lord.

Authors' Addresses

Henk Birkholz
Fraunhofer SIT
Rheinstrasse 75
Darmstadt 64295
Germany

Email: henk.birkholz@sit.fraunhofer.de

Christoph Vigano
Universitaet Bremen

Email: christoph.vigano@uni-bremen.de

Carsten Bormann
Universitaet Bremen TZI
Bibliothekstr. 1
Bremen D-28359
Germany

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

Network Working Group
Internet-Draft
Intended status: Informational
Expires: August 30, 2018

J. Roatch

C. Bormann
Universitaet Bremen TZI

February 26, 2018

Concise Binary Object Representation (CBOR) Tags for Typed Arrays
draft-jroatch-cbor-tags-07

Abstract

The Concise Binary Object Representation (CBOR, RFC 7049) 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.

The present document makes use of this extensibility to define a number of CBOR tags for typed arrays of numeric data, as well as two additional tags for multi-dimensional and homogeneous arrays. It is intended as the reference document for the IANA registration of the CBOR tags defined.

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 August 30, 2018.

Table of Contents

1. Introduction

The Concise Binary Object Representation (CBOR, [RFC7049]) provides for the interchange of structured data without a requirement for a pre-agreed schema. RFC 7049 defines a basic set of data types, as well as a tagging mechanism that enables extending the set of data types supported via an IANA registry.

Recently, a simple form of typed arrays of numeric data have received interest both in the Web graphics community [TypedArray] and in the JavaScript specification [TypedArrayES6], as well as in corresponding implementations [ArrayBuffer].

Since these typed arrays may carry significant amounts of data, there is interest in interchanging them in CBOR without the need of lengthy conversion of each number in the array.

This document defines a number of interrelated CBOR tags that cover these typed arrays, as well as two additional tags for multi-dimensional and homogeneous arrays. It is intended as the reference document for the IANA registration of the tags defined.

1.1. Terminology

The term "byte" is used in its now customary sense as a synonym for "octet". Where bit arithmetic is explained, this document uses the notation familiar from the programming language C (including C++14's 0bnnn binary literals), except that the operator "**" stands for exponentiation.

2. Typed Arrays

Typed arrays are homogeneous arrays of numbers, all of which are encoded in a single form of binary representation. The concatenation of these representations is encoded as a single CBOR byte string (major type 2), enclosed by a single tag indicating the type and encoding of all the numbers represented in the byte string.

2.1. Types of numbers

Three classes of numbers are of interest: unsigned integers (uint), signed integers (twos' complement, sint), and IEEE 754 binary floating point numbers (which are always signed). For each of these classes, there are multiple representation lengths in active use:

Length ll	uint	sint	float
0	uint8	sint8	binary16
1	uint16	sint16	binary32
2	uint32	sint32	binary64
3	uint64	sint64	binary128

Table 1: Length values

Here, sintN stands for a signed integer of exactly N bits (for instance, sint16), and uintN stands for an unsigned integer of exactly N bits (for instance, uint32). The name binaryN stands for the number form of the same name defined in IEEE 754.

Since one objective of these tags is to be able to directly ship the ArrayBuffers underlying the Typed Arrays without re-encoding them, and these may be either in big endian (network byte order) or in little endian form, we need to define tags for both variants.

In total, this leads to 24 variants. In the tag, we need to express the choice between integer and floating point, the signedness (for integers), the endianness, and one of the four length values.

In order to simplify implementation, a range of tags is being allocated that allows retrieving all this information from the bits of the tag: Tag values from TBD64 to TBD87.

The value is split up into 5 bit fields: TDB0b010_f_s_e_ll, as detailed in Table 2.

Field	Use
TBD0b010	a constant such as '010', to be defined
f	0 for integer, 1 for float
s	0 for unsigned integer or float, 1 for signed integer
e	0 for big endian, 1 for little endian
ll	A number for the length (Table 1).

Table 2: Bit fields in the low 8 bits of the tag

The number of bytes in each array element can then be calculated by "2**(f + ll)" (or "1 << (f + ll)" in a typical programming language). (Notice that f and ll are the lsb of each nibble (4bit) in the byte.)

In the CBOR representation, the total number of elements in the array is not expressed explicitly, but implied from the length of the byte string and the length of each representation. It can be computed inversely to the previous formula: "bytelength >> (f + ll)".

For the uint8/sint8 values, the endianness is redundant. Only the big endian variant is used. As a special case, what would be the little endian variant of uint8 is used to signify that the numbers in the array are using clamped conversion from integers, as described in more detail in Section 7.1 of [TypedArrayUpdate].

3. Additional Array Tags

This specification defines two additional array tags. The Multi-dimensional Array tag can be combined with classical CBOR arrays as well as with Typed Arrays in order to build multi-dimensional arrays with constant numbers of elements in the sub-arrays. The Homogeneous Array tag can be used to facilitate the ingestion of homogeneous classical CBOR arrays, providing performance advantages even when a Typed Array does not apply.

3.1. Multi-dimensional Array

Tag: TBD40

Data Item: array (major type 4) of two arrays, one array (major type 4) of dimensions, and one array (major type 4, a Typed Array, or a Homogeneous Array) of elements

A multi-dimensional array is represented as a tagged array that contains two (one-dimensional) arrays. The first array defines the dimensions of the multi-dimensional array (in the sequence of outer dimensions towards inner dimensions) while the second array represents the contents of the multi-dimensional array. If the second array is itself tagged as a Typed Array then the element type of the multi-dimensional array is known to be the same type as that of the Typed Array. Data in the Typed Array byte string consists of consecutive values where the last dimension is considered contiguous (row-major order).

```
uint16_t a[2][3] = {
  {0, 1, 2},   /* row 0 */
  {3, 4, 5},
};

<Tag TBD40> # multi-dimensional array tag
   82       # array(2)
     82      # array(2)
       02     # unsigned(2) 1st Dimension
       03     # unsigned(3) 2nd Dimension
     d8 41   # uint16 array
       4a     # byte string(12)
         00 00 # unsigned(0)
         00 01 # unsigned(1)
         00 02 # unsigned(2)
         00 03 # unsigned(3)
         00 04 # unsigned(4)
         00 05 # unsigned(5)
```

Figure 1: Multi-dimensional array in C and CBOR

3.2. Homogeneous Array

Tag: TBD41

Data Item: array (major type 4)

This tag provides a hint to decoders that the array tagged by it has elements that are all of the same application type. The element type

of the array is thus determined by the application type of the first array element. This can be used by implementations in strongly typed languages while decoding to create native homogeneous arrays of specific types instead of ordered lists.

Which CBOR data items constitute elements of the same application type is specific to the application. However, type systems of programming languages have enough commonality that an application should be able to create portable homogeneous arrays.

```
bool boolArray[2] = { true, false };

<Tag TBD41>  # Homogeneous Array Tag
   82           #array(2)
      F5        # true
      F4        # false
```

Figure 2: Homogeneous array in C and CBOR

4. Discussion

Support for both little- and big-endian representation may seem out of character with CBOR, which is otherwise fully big endian. This support is in line with the intended use of the typed arrays and the objective not to require conversion of each array element.

This specification allocates a sizable chunk out of the single-byte tag space. This use of code point space is justified by the wide use of typed arrays in data interchange.

Applying a Homogeneous Array tag to a Typed Array would be redundant and is therefore not provided by the present specification.

5. CDDL typenames

For the use with CDDL [I-D.ietf-cbor-cddl], the typenames defined in Figure 3 are recommended:

```
ta-uint8 = #6.TBD64(bstr)
ta-uint16be = #6.TBD65(bstr)
ta-uint32be = #6.TBD66(bstr)
ta-uint64be = #6.TBD67(bstr)
ta-uint8-clamped = #6.TBD68(bstr)
ta-uint16le = #6.TBD69(bstr)
ta-uint32le = #6.TBD70(bstr)
ta-uint64le = #6.TBD71(bstr)
ta-sint8 = #6.TBD72(bstr)
ta-sint16be = #6.TBD73(bstr)
ta-sint32be = #6.TBD74(bstr)
ta-sint64be = #6.TBD75(bstr)
; reserved: #6.TBD76(bstr)
ta-sint16le = #6.TBD77(bstr)
ta-sint32le = #6.TBD78(bstr)
ta-sint64le = #6.TBD79(bstr)
ta-float16be = #6.TBD80(bstr)
ta-float32be = #6.TBD81(bstr)
ta-float64be = #6.TBD82(bstr)
ta-float128be = #6.TBD83(bstr)
ta-float16le = #6.TBD84(bstr)
ta-float32le = #6.TBD85(bstr)
ta-float64le = #6.TBD86(bstr)
ta-float128le = #6.TBD87(bstr)
homogeneous<array> = #6.TBD41(array)
multi-dim<dim, array> = #6.TBD40([dim, array])
```

Figure 3: Recommended typenames for CDDL

6. IANA Considerations

IANA is requested to allocate the tags in Table 3, with the present document as the specification reference.

Tag	Data Item	Semantics
TBD64	byte string	uint8 Typed Array
TBD65	byte string	uint16, big endian, Typed Array
TBD66	byte string	uint32, big endian, Typed Array
TBD67	byte string	uint64, big endian, Typed Array
TBD68	byte string	uint8 Typed Array, clamped arithmetic
TBD69	byte string	uint16, little endian, Typed Array
TBD70	byte string	uint32, little endian, Typed Array
TBD71	byte string	uint64, little endian, Typed Array
TBD72	byte string	sint8 Typed Array
TBD73	byte string	sint16, big endian, Typed Array
TBD74	byte string	sint32, big endian, Typed Array
TBD75	byte string	sint64, big endian, Typed Array
TBD76	byte string	(reserved)
TBD77	byte string	sint16, little endian, Typed Array
TBD78	byte string	sint32, little endian, Typed Array
TBD79	byte string	sint64, little endian, Typed Array
TBD80	byte string	IEEE 754 binary16, big endian, Typed Array
TBD81	byte string	IEEE 754 binary32, big endian, Typed Array
TBD82	byte string	IEEE 754 binary64, big endian, Typed Array
TBD83	byte string	IEEE 754 binary128, big endian, Typed Array
TBD84	byte string	IEEE 754 binary16, little endian, Typed Array
TBD85	byte string	IEEE 754 binary32, little endian, Typed Array
TBD86	byte string	IEEE 754 binary64, little endian, Typed Array
TBD87	byte string	IEEE 754 binary128, little endian, Typed Array
TBD40	array of two arrays*	Multi-dimensional Array
TBD41	array	Homogeneous Array

Table 3: Values for Tags

*) TBD40 data item: second element of outer array in data item is native CBOR array (major type 4) or Typed Array (one of Tag TBD64..TBD87)

RFC editor note: Please replace TBDnn by the tag numbers allocated by IANA throughout the document and delete this note. IANA note: To make the calculations work, TDB64 to TBD87 need to come from a contiguous range the start of which is divisible by 32.

TO DO: The WG needs to figure out whether it is OK to spend 24 "good" (1+1 byte) tags for this, whether this all goes to 1+2 byte tags, or whether maybe the layout of the bits in the tag should change to move the larger datatypes into the 1+2 range and just the 8-bit ones into the 1+1 range.

7. Security Considerations

The security considerations of RFC 7049 apply; the tags introduced here are not expected to raise security considerations beyond those.

8. References

8.1. Normative References

[I-D.ietf-cbor-cddl]
Birkholz, H., Vigano, C., and C. Bormann, "Concise data definition language (CDDL): a notational convention to express CBOR data structures", draft-ietf-cbor-cddl-02 (work in progress), February 2018.

[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>.

8.2. Informative References

[ArrayBuffer]
Mozilla Developer Network, "JavaScript typed arrays", 2013, <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Typed_arrays>.

[TypedArray]
Vukicevic, V. and K. Russell, "Typed Array Specification", February 2011, <https://www.khronos.org/registry/typedarray/specs/1.0/>.

[TypedArrayES6]
"22.2 TypedArray Objects", in: ECMA-262 6th Edition, The ECMAScript 2015 Language Specification, June 2015, <http://www.ecma-international.org/ecma-262/6.0/#sec-typedarray-objects>.

[TypedArrayUpdate]
Herman, D. and K. Russell, "Typed Array Specification", July 2013, <https://www.khronos.org/registry/typedarray/specs/latest/>.

Contributors

Glenn Engel suggested the tags for multi-dimensional arrays and homogeneous arrays.

Acknowledgements

TBD

Authors' Addresses

Johnathan Roatch

Email: jroatch@gmail.com

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

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