Concise Binary Object Representation (CBOR)
draft-ietf-cbor-7049bis-07

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.

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 February 26, 2020.

Copyright Notice

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

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

Table of Contents

1. Introduction ........................................... 4
   1.1. Objectives ...................................... 4
   1.2. Terminology ...................................... 6
2. CBOR Data Models ....................................... 7
   2.1. Extended Generic Data Models ...................... 8
   2.2. Specific Data Models .............................. 9
3. Specification of the CBOR Encoding .................... 9
   3.1. Major Types ..................................... 10
   3.2. Indefinite Lengths for Some Major Types .......... 13
       3.2.1. The "break" Stop Code ....................... 13
       3.2.2. Indefinite-Length Arrays and Maps .......... 13
       3.2.3. Indefinite-Length Byte Strings and Text Strings .. 15
   3.3. Floating-Point Numbers and Values with No Content .... 16
   3.4. Tagging of Items ................................ 17
       3.4.1. Date and Time ............................... 19
       3.4.2. Standard Date/Time String ................. 19
       3.4.3. Epoch-based Date/Time ....................... 20
       3.4.4. Bignums .................................. 20
       3.4.5. Decimal Fractions and Bigfloats ............ 21
       3.4.6. Content Hints ............................. 23
           3.4.6.1. Encoded CBOR Data Item ............. 23
           3.4.6.2. Expected Later Encoding for CBOR-to-JSON Converters ............... 23
           3.4.6.3. Encoded Text ......................... 24
       3.4.7. Self-Described CBOR ........................ 25
4. Serialization Considerations .......................... 25
4.1. Preferred Serialization .................................. 25
4.2. Deterministically Encoded CBOR ............................ 26
  4.2.1. Core Deterministic Encoding Requirements ............. 26
  4.2.2. Additional Deterministic Encoding Considerations .... 27
  4.2.3. Length-first map key ordering .......................... 28
5. Creating CBOR-Based Protocols ............................... 29
  5.1. CBOR in Streaming Applications ........................... 30
  5.2. Generic Encoders and Decoders ............................ 31
  5.3. Invalid Items ........................................ 31
  5.4. Handling Unknown Simple Values and Tags ................. 32
  5.5. Numbers .............................................. 33
  5.6. Specifying Keys for Maps ................................ 33
  5.6.1. Equivalence of Keys .................................. 34
  5.7. Undefined Values ..................................... 35
  5.8. Strict Decoding Mode ................................... 35
6. Converting Data between CBOR and JSON ....................... 37
  6.1. Converting from CBOR to JSON ............................. 37
  6.2. Converting from JSON to CBOR ............................. 38
7. Future Evolution of CBOR .................................. 39
  7.1. Extension Points ...................................... 40
  7.2. Curating the Additional Information Space ............... 40
8. Diagnostic Notation ......................................... 41
  8.1. Encoding Indicators .................................... 42
9. IANA Considerations ......................................... 42
  9.1. Simple Values Registry .................................. 43
  9.2. Tags Registry .......................................... 43
  9.3. Media Type ("MIME Type") ................................ 43
  9.4. CoAP Content-Format .................................... 44
  9.5. The +cbor Structured Syntax Suffix Registration ......... 45
10. Security Considerations ..................................... 45
11. References ............................................... 47
  11.1. Normative References ................................... 47
  11.2. Informative References ................................ 48
Appendix A. Examples .......................................... 50
Appendix B. Jump Table ........................................ 54
Appendix C. Pseudocode ......................................... 57
Appendix D. Half-Precision ...................................... 59
Appendix E. Comparison of Other Binary Formats to CBOR’s Design Objectives .......................................... 60
  E.1. ASN.1 DER, BER, and PER ................................ 61
  E.2. MessagePack ............................................ 61
  E.3. BSON .................................................. 62
  E.4. MSDTP: RFC 713 ........................................ 62
  E.5. Conciseness on the Wire .................................. 62
Appendix F. Changes from RFC 7049 ............................. 63
Acknowledgements .............................................. 63
Authors’ Addresses ........................................... 64
1. Introduction

There are hundreds of standardized formats for binary representation of structured data (also known as binary serialization formats). Of those, some are for specific domains of information, while others are generalized for arbitrary data. In the IETF, probably the best-known formats in the latter category are ASN.1’s BER and DER [ASN.1].

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

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

This document 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 implementation complexity maintaining a lower bound. Using either general compression schemes or extensive bit-fiddling violates the complexity goals.

5. The format must be applicable to both constrained nodes and high-volume applications.

* This means it must be reasonably frugal in CPU usage for both encoding and decoding. This is relevant both for constrained nodes and for potential usage in applications with a very high volume of data.

6. The format must support all JSON data types for conversion to and from JSON.

* It must support a reasonable level of conversion as long as the data represented is within the capabilities of JSON. It must be possible to define a unidirectional mapping towards JSON for all types of data.

7. The format must be extensible, and the extended data must be decodable by earlier decoders.

* The format is designed for decades of use.
* The format must support a form of extensibility that allows fallback so that a decoder that does not understand an extension can still decode the message.

* The format must be able to be extended in the future by later IETF standards.

1.2. Terminology

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

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

This specification makes use of the following terminology:

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

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

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

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

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

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

Where bit arithmetic or data types are explained, this document uses the notation familiar from the programming language C, except that "**" denotes exponentiation. Similar to the "0x" notation for hexadecimal numbers, numbers in binary notation are prefixed with "0b". Underscores can be added to such a number solely for readability, so 0b00100001 (0x21) might be written 0b001_00001 to emphasize the desired interpretation of the bits in the byte; in this case, it is split into three bits and five bits. Encoded CBOR data items are sometimes given in the "0x" or "0b" notation; these values are first interpreted as numbers as in C and are then interpreted as byte strings in network byte order, including any leading zero bytes expressed in the notation.

2. CBOR Data Models

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

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

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

- 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) [IEEE754]
o a sequence of zero or more bytes ("byte string")

o a sequence of zero or more Unicode code points ("text string")

o a sequence of zero or more data items ("array")

o a mapping (mathematical function) from zero or more data items ("keys") each to a data item ("values"), ("map")

o a tagged data item ("tag"), comprising a tag number (an integer in the range 0..2^64-1) and a tagged value (a data item)

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

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

2.1. Extended Generic Data Models

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

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

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

o 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 values of a simple type 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 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 6. 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.
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).

These values are reserved for future additions to the CBOR format. In the present version of CBOR, the encoded item is not well-formed.

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

The 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 6). A decoder in a constrained implementation can instead use the structure of the initial byte and following bytes for more compact code (see Appendix C for a rough impression of how this could look).

3.1. Major Types

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

Major type 0: an integer in the range 0..2**64-1 inclusive. The value of the encoded item is the argument itself. For example, the integer 10 is denoted as the one byte 0b000_01010 (major type
Integer 500 would be 0b000_11001 (major type 0, additional information 22) followed by the two bytes 0x01f4, which is 500 in decimal.

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

Major type 2: a byte string. The number of bytes in the string is equal to the argument. For example, a byte string whose length is 5 would have an initial byte of 0b010_00101 (major type 2, additional information 5 for the length), followed by 5 bytes of binary content. A byte string whose length is 500 would have 3 initial bytes of 0b010_11001 (major type 2, additional information 25 to indicate a two-byte length) followed by the two bytes 0x01f4 for a length of 500, followed by 500 bytes of binary content.

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

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

Major type 5: a map of pairs of data items. Maps are also called tables, dictionaries, hashes, or objects (in JSON). A map is comprised of pairs of data items, each pair consisting of a key that is immediately followed by a value. The argument is the number of pairs of data items in the map. For example, a map that contains 9 pairs would have an initial byte of 0b101_01001 (major type of 5, additional information of 9 for the number of pairs) followed by the 18 remaining items. The first item is the
first key, the second item is the first value, the third item is the second key, and so on. 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 is the argument and whose enclosed data item 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 6).

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.

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

Table 1: Overview over CBOR major types (definite length encoded)
3.2. Indefinite Lengths for Some Major Types

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

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

3.2.1. The "break" Stop Code

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

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

3.2.2. Indefinite-Length Arrays and Maps

Indefinite-length arrays and maps are represented using their major type with the additional information value of 31, followed by an arbitrary-length sequence of 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:
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
  9F -- Start indefinite-length array
    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"
```
An example of an indefinite-length map (that happens to have two key/value pairs) might be:

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

3.2.3. Indefinite-Length Byte Strings and Text Strings

Indefinite-length strings are represented by a byte containing the major type and additional information value of 31, followed by a series of zero or more byte or text strings ("chunks") that have definite lengths, followed by the "break" stop code (Section 3.2.1). The data item represented by the indefinite-length string is the concatenation of the chunks (i.e., the empty byte or text string, respectively, if no chunk is present).

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

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

For example, assume the sequence:

0b010_11111 0b010_00100 0xaabbccddd 0b010_00011 0xeeff99 0b111_11111
5F  -- Start indefinite-length byte string
44  -- Byte string of length 4
    aabbcdd  -- Bytes content
43  -- Byte string of length 3
    eeff99  -- Bytes content
FF  -- "break"

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

3.3. Floating-Point Numbers and Values with No Content

Major type 7 is for two types of data: floating-point numbers and "simple values" that do not need any content. Each value of the 5-bit additional information in the initial byte has its own separate meaning, as defined in Table 2. Like the major types for integers, items of this major type do not carry content data; all the information is in the initial bytes.

<table>
<thead>
<tr>
<th>5-Bit Value</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>0..23</td>
<td>Simple value (value 0..23)</td>
</tr>
<tr>
<td>24</td>
<td>Simple value (value 32..255 in following byte)</td>
</tr>
<tr>
<td>25</td>
<td>IEEE 754 Half-Precision Float (16 bits follow)</td>
</tr>
<tr>
<td>26</td>
<td>IEEE 754 Single-Precision Float (32 bits follow)</td>
</tr>
<tr>
<td>27</td>
<td>IEEE 754 Double-Precision Float (64 bits follow)</td>
</tr>
<tr>
<td>28-30</td>
<td>Reserved, not well-formed in the present document</td>
</tr>
<tr>
<td>31</td>
<td>&quot;break&quot; stop code for indefinite-length items</td>
</tr>
</tbody>
</table>

Table 2: 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 3 lists the values assigned and available for simple types.
Table 3: Simple Values

<table>
<thead>
<tr>
<th>Value</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>0..19</td>
<td>(Unassigned)</td>
</tr>
<tr>
<td>20</td>
<td>False</td>
</tr>
<tr>
<td>21</td>
<td>True</td>
</tr>
<tr>
<td>22</td>
<td>Null</td>
</tr>
<tr>
<td>23</td>
<td>Undefined value</td>
</tr>
<tr>
<td>24..31</td>
<td>(Reserved)</td>
</tr>
<tr>
<td>32..255</td>
<td>(Unassigned)</td>
</tr>
</tbody>
</table>

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

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

3.4. Tagging of Items

In CBOR, a data item can be enclosed by a tag to give it additional semantics while retaining its structure. The tag is major type 6, and represents an unsigned integer as indicated by the tag’s argument (Section 3); the (sole) enclosed 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 number usually restricts what kinds of nested data item or items are valid for tags using this tag number.

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 3.4.4). This would be marked as 0b110_00010 (major type 6, additional information 2 for the tag number) 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 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 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 applies semantics to the data item it encloses. Thus, 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. That is, a tagged item is a data item consisting of a tag number and an enclosed value. The content of the tagged item (the enclosed data item) is the data item (the value) that is being tagged.

IANA maintains a registry of tag numbers as described in Section 9.2. Table 4 provides a list of tag numbers that were defined in [RFC7049], with definitions in the rest of this section. 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.

<table>
<thead>
<tr>
<th>Tag Number</th>
<th>Data Item</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>text</td>
<td>Standard date/time string; see Section 3.4.2</td>
</tr>
<tr>
<td>1</td>
<td>multiple</td>
<td>Epoch-based date/time; see Section 3.4.3</td>
</tr>
<tr>
<td>2</td>
<td>byte string</td>
<td>Positive bignum; see Section 3.4.4</td>
</tr>
<tr>
<td>3</td>
<td>byte string</td>
<td>Negative bignum; see Section 3.4.4</td>
</tr>
<tr>
<td>4</td>
<td>array</td>
<td>Decimal fraction; see Section 3.4.5</td>
</tr>
<tr>
<td>5</td>
<td>array</td>
<td>Bigfloat; see Section 3.4.5</td>
</tr>
<tr>
<td>21</td>
<td>multiple</td>
<td>Expected conversion to base64url encoding;</td>
</tr>
</tbody>
</table>
### 3.4.1. Date and Time

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

### 3.4.2. Standard Date/Time String

Tag 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 that doesn’t match the [RFC4287] format is invalid.
3.4.3. 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 enclosed item MUST be an unsigned or negative integer (major types 0 and 1), or a floating-point number (major type 7 with additional information 25, 26, or 27). Other contained types are invalid.

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

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

To indicate fractional seconds, floating-point values can be used within Tag 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 enclosed value to be an integer (or a floating-point value) only.

3.4.4. Bignums

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

Bignums are encoded as a byte string data item, which is interpreted as an unsigned integer $n$ in network byte order. Contained items of other types are invalid. For tag number 2, the value of the bignum is $n$. For tag number 3, the value of the bignum is $-1 - n$. The
preferred encoding of the byte string is to leave out any leading zeroes (note that this means the preferred encoding for n = 0 is the empty byte string, but see below). Decoders that understand these tags MUST be able to decode bignums that do have leading zeroes. The preferred encoding of an integer that can be represented using major type 0 or 1 is to encode it this way instead of as a bignum (which means that the empty string never occurs in a bignum when using preferred encoding). Note that this means the non-preferred choice of a bignum representation instead of a basic integer for encoding a number is not intended to have application semantics (just as the choice of a longer basic integer representation than needed, such as 0x1800 for 0x00 does not).

For example, the number 18446744073709551616 (2**64) is represented as 0b10_00010 (major type 6, tag number 2), followed by 0b010_01001 (major type 2, length 9), followed by 0x0100000000000000 (one byte 0x01 and eight bytes 0x00). In hexadecimal:

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

3.4.5. Decimal Fractions and Bigfloats

Protocols using tag number 4 extend the generic data model with data items representing arbitrary-length decimal fractions \(m \times (10^e)\). Protocols using tag number 5 extend the generic data model with data items representing arbitrary-length binary fractions \(m \times (2^e)\). 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 number 4) use base-10 exponents; the value of a decimal fraction data item is \(m \times (10^{*e})\). Bigfloats
Internet-Draft                    CBOR                       August 2019

(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 also can be a bignum
(Section 3.4.4). Contained items with other structures are invalid.

An example of a decimal fraction is that the number 273.15 could be
represented as 0b110_00101 (major type of 6 for the tag, additional
information of 5 for the number 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_00000 (major type of 0 for the second integer,
additional information of 25 for a two-byte value), followed by
0b1101010101110111 (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 number of tag), followed by 0b100_00010 (major type of 4
for the array, additional information of 2 for the length of the
array), followed by 0b001_00000 (major type of 1 for the first
integer, additional information of 0 for the value of -1), followed
by 0b000_00011 (major type of 0 for the second integer, additional
information of 3 for the value of 3). In hexadecimal:

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

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

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

3.4.6.1. Encoded CBOR Data Item

Sometimes it is beneficial to carry an embedded CBOR data item that is not meant to be decoded immediately at the time the enclosing data item is being decoded. Tag number 24 (CBOR data item) can be used to tag the embedded byte string as a data item encoded in CBOR format. Contained items that aren’t byte strings are invalid. Any contained byte string is valid, even if it encodes an invalid or ill-formed CBOR item.

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

Tags number 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]. For base64url encoding (tag number 21), padding is not used (see Section 3.2 of RFC 4648); that is, all trailing equals signs (“=””) are removed from the encoded string. For base64 encoding (tag number 22), padding is used as defined in RFC 4648. For both base64url and base64, padding bits are set to zero (see Section 3.5 of RFC 4648), and encoding is performed without the inclusion of any line breaks, whitespace, or other additional characters. Note that, for all three tag numbers, the encoding of the empty byte string is the empty text string.
3.4.6.3. Encoded Text

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

- 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, as defined in [RFC4648]. If any of:
  * the encoded text string contains non-alphabet characters or only 1 character in the last block of 4, or
  * the padding bits in a 2- or 3-character block are not 0, or
  * the base64 encoding has the wrong number of padding characters, or
  * the base64url encoding has padding characters,
  the string is invalid.

- Tag number 35 is for regular expressions that are roughly in Perl Compatible Regular Expressions (PCRE/PCRE2) form [PCRE] or a version of the JavaScript regular expression syntax [ECMA262]. (Note that more specific identification may be necessary if the actual version of the specification underlying the regular expression, or more than just the text of the regular expression itself, need to be conveyed.) Any contained string value is valid.

- 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.

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.
3.4.7. Self-Described CBOR

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

Tag number 55799 is defined for this purpose. It does not impart any special semantics on the data item that it encloses; that is, the semantics of a data item enclosed in tag number 55799 is exactly identical to the semantics of the data item 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; however, the present specification does not put the burden of enforcing this preference on either encoder or decoder.

Some constrained decoders may be limited in their ability to decode non-preferred serializations: For example, if only integers below 1_000_000_000 are expected in an application, the decoder may leave out the code that would be needed to decode 64-bit arguments in integers. An encoder that always uses preferred serialization ("preferred encoder") interoperates with this decoder for the numbers that can occur in this application. More generally speaking, it therefore can be said that a preferred encoder is more universally interoperable (and also less wasteful) than one that, say, always uses 64-bit integers.
Similarly, a constrained encoder may be limited in the variety of representation variants it supports in such a way that it does not emit preferred serializations ("variant encoder"): Say, it could be designed to always use the 32-bit variant for an integer that it encodes even if a short representation is available (again, assuming that there is no application need for integers that can only be represented with the 64-bit variant). A decoder that does not rely on only ever receiving preferred serializations ("variation-tolerant decoder") can there be said to be more universally interoperable (it might very well optimize for the case of receiving preferred serializations, though). Full implementations of CBOR decoders are by definition variation-tolerant; the distinction is only relevant if a constrained implementation of a CBOR decoder meets a variant encoder.

The preferred serialization always uses the shortest form of representing the argument (Section 3)); it also uses the shortest floating-point encoding that preserves the value being encoded (see Section 5.5). Definite length encoding is preferred whenever the length is known at the time the serialization of the item starts.

4.2. Deterministically Encoded CBOR

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

4.2.1. Core Deterministic Encoding Requirements

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

o Arguments (see Section 3) for integers, lengths in major types 2 through 5, and tags MUST be as short as possible. In particular:

  * 0 to 23 and -1 to -24 MUST be expressed in the same byte as the major type;
  * 24 to 255 and -25 to -256 MUST be expressed only with an additional uint8_t;
  * 256 to 65535 and -257 to -65536 MUST be expressed only with an additional uint16_t;
* 65536 to 4294967295 and -65537 to -4294967296 MUST be expressed only with an additional uint32_t.

- 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.

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

4.2.2. Additional Deterministic Encoding Considerations

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

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

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

- If a protocol includes a field that can express floating-point values (Section 3.3), the protocol’s deterministic encoding needs to specify whether the integer 1.0 is encoded as 0x01, 0xf93c00, 0xfa3f800000, or 0xfb3ff00000000000. 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 with an absolute value of 2^64 or larger using tag numbers 2 or 3 (Section 3.4.4), the protocol’s deterministic encoding needs to specify whether small integers are expressed using the tag or major types 0 and 1.

- A protocol might give encoders the choice of representing a URL as either a text string or, using Section 3.4.6.3, 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’s absent, not allow either one.

4.2.3. Length-first map key ordering

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

Bormann & Hoffman Expires February 26, 2020 [Page 28]
A CBOR encoding satisfies the "length-first core deterministic encoding requirements" if it satisfies the core deterministic encoding requirements except that the keys in every map MUST be sorted such that:

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

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

1. 10, encoded as 0x0a.
2. -1, encoded as 0x20.
3. false, encoded as 0xf4.
4. 100, encoded as 0x1864.
5. "z", encoded as 0x617a.
6. [-1], encoded as 0x8120.
7. "aa", encoded as 0x626161.
8. [100], encoded as 0x811864.

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

5. Creating CBOR-Based Protocols

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

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

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

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

5.1. CBOR in Streaming Applications

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

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

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

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

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

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

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

5.3. Invalid Items

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

1. Replace the problematic item with an error marker and continue with the next item, or

2. Issue an error and stop processing altogether.

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

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

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

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

5.4. Handling Unknown Simple Values and Tags

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

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

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

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

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

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

5.6. Specifying Keys for Maps

The encoding and decoding applications need to agree on what types of keys are going to be used in maps. In applications that need to interwork with JSON-based applications, keys probably should be limited to UTF-8 strings only; otherwise, there has to be a specified mapping from the other CBOR types to Unicode characters, and this often leads to implementation errors. In applications where keys are numeric in nature and numeric ordering of keys is important to the application, directly using the numbers for the keys is useful.
If multiple types of keys are to be used, consideration should be given to how these types would be represented in the specific programming environments that are to be used. For example, in JavaScript Maps [ECMA262], a key of integer 1 cannot be distinguished from a key of floating-point 1.0. This means that, if integer keys are used, the protocol needs to avoid use of floating-point keys the values of which happen to be integer numbers in the same map.

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

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

The CBOR data model for maps does not allow ascribing semantics to the order of the key/value pairs in the map representation. Thus, a CBOR-based protocol MUST NOT specify that changing the key/value pair order in a map would change the semantics, except to specify that some, orders are disallowed, for example where they would not meet the requirements of a deterministic encoding (Section 4.2). (Any secondary effects of map ordering such as on timing, cache usage, and other potential side channels are not considered part of the semantics but may be enough reason on its own for a protocol to require a deterministic encoding format.)

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

5.6.1. Equivalence of Keys

The specific data model applying to a CBOR data item is used to determine whether keys occurring in maps are duplicates or distinct.
At the generic data model level, numerically equivalent integer and floating-point values are distinct from each other, as they are from the various big numbers (Tags 2 to 5). Similarly, text strings are distinct from byte strings, even if composed of the same bytes. A tagged value is distinct from an untagged value or from a value tagged with a different tag.

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

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

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

As discussed in Section 2.2, specific data models can make values equivalent for the purpose of comparing map keys that are distinct in the generic data model. Note that this implies that a generic decoder may deliver a decoded map to an application that needs to be checked for duplicate map keys by that application (alternatively, the decoder may provide a programming interface to perform this service for the application). Specific data models cannot distinguish values for map keys that are equal for this purpose at the generic data model level.

5.7. Undefined Values

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

5.8. Strict Decoding Mode

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

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

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

- 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 number that it has been tagged with

A decoder in strict mode can do one of two things when it encounters a tag number 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 number or simple value.

The latter approach, which is also appropriate for non-strict decoders, supports forward compatibility with newly registered tags and simple values without the requirement to update the encoder at the same time as the calling application. (For this, the API for the decoder needs to have a way to mark unknown items so that the calling application can handle them in a manner appropriate for the program.)
Since some of this processing may have an appreciable cost (in particular with duplicate detection for maps), support of strict mode is not a requirement placed on all CBOR decoders.

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

6. Converting Data between CBOR and JSON

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

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

6.1. Converting from CBOR to JSON

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

- 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

Bormann & Hoffman Expires February 26, 2020 [Page 37]
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 and becomes a JSON string.
- For all other tags (major type 6, any other tag number), the enclosed CBOR item is represented as a JSON value; the tag number is ignored.
- Indefinite-length items are made definite before conversion.

6.2. Converting from JSON to CBOR

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

- 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 (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.

7. Future Evolution of CBOR

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

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

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

7.1. Extension Points

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

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

CBOR has three major extension points:

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

- 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 number can choose to simply ignore it or to process it as an unknown tag number wrapping the enclosed data item. 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 to this space is a major step. There are also very few codepoints left.

7.2. Curating the Additional Information Space

The human mind is sometimes drawn to filling in little perceived gaps to make something neat. We expect the remaining gaps in the codepoint space for the additional information values to be an attractor for new ideas, just because they are there.
The present specification does not manage the additional information codepoint space by an IANA registry. Instead, allocations out of this space can only be done by updating this specification.

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

8. Diagnostic Notation

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

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

The diagnostic notation is loosely based on JSON as it is defined in RFC 8259, extending it where needed.

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

\[
0(\"2013-03-21T20:04:00Z\")
\]

or the equivalent relative time as

\[
1(1363896240)
\]
Byte strings are notated in one of the base encodings, without padding, enclosed in single quotes, prefixed by >h< for base16, >b32< for base32, >h32< for base32hex, >b64< for base64 or base64url (the actual encodings do not overlap, so the string remains unambiguous). For example, the byte string 0x12345678 could be written h’12345678’, b32’CI2FM6A’, or b64’EjRWeA’.

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

8.1. Encoding Indicators

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

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

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

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

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

9. IANA Considerations

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

9.1.  Simple Values Registry

IANA has created the "Concise Binary Object Representation (CBOR) Simple Values" registry at [IANA.cbor-simple-values]. The initial values are shown in Table 3.

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

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

9.2.  Tags Registry

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

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

- 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.

9.3.  Media Type ("MIME Type")

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

Type name: application
Subtype name: cbor

Required parameters: n/a

Optional parameters: n/a

Encoding considerations: binary

Security considerations: See Section 10 of this document

Interoperability considerations: n/a

Published specification: This document

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

Additional information:
Magic number(s): n/a
File extension(s): .cbor
Macintosh file type code(s): n/a

Person & email address to contact for further information:
Carsten Bormann
cabo@tzi.org

Intended usage: COMMON

Restrictions on usage: none

Author:
Carsten Bormann <cabotzi.org>

Change controller:
The IESG <iesg@ietf.org>

9.4. CoAP Content-Format

Media Type: application/cbor

Encoding: -

Id: 60

Reference: [RFCthis]
9.5. The +cbor Structured Syntax Suffix Registration

Name: Concise Binary Object Representation (CBOR)

+suffix: +cbor

References: [RFCthis]

Encoding Considerations: CBOR is a binary format.

Interoperability Considerations: n/a

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

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

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

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

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

Security Considerations: See Section 10 of this document

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

Author/Change Controller:
The Apps Area Working Group.
The IESG has change control over this registration.

10. Security Considerations

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

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

Hostile input may be constructed to overrun buffers, overflow or underflow integer arithmetic, or cause other decoding disruption. CBOR data items might have lengths or sizes that are intentionally extremely large or too short. Resource exhaustion attacks might attempt to lure a decoder into allocating very big data items (strings, arrays, maps, or 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. 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 is employed (see Section 7 of [SIPHASH]). Such superlinear efforts can be employed 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 that are used in a security context should be defined in such a way that potential multiple interpretations are reliably reduced to a single interpretation. For example, an attacker could make use of invalid input such as duplicate keys in maps, or exploit different precision in processing numbers to make one application base its decisions on a different interpretation than the one that will be used by a second application. To facilitate consistent interpretation, encoder and decoder implementations used in such contexts should provide at least one strict mode of operation (Section 5.8).

11. References

11.1. Normative References


11.2. Informative References


IANA, "Concise Binary Object Representation (CBOR) Simple Values",

IANA, "Concise Binary Object Representation (CBOR) Tags",


<https://www.rfc-editor.org/info/rfc713>.


Bormann, C. and P. Hoffman, "Concise Binary Object Representation (CBOR)", RFC 7049, DOI 10.17487/RFC7049, October 2013,


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')).

<table>
<thead>
<tr>
<th>Diagnostic</th>
<th>Encoded</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0x00</td>
</tr>
<tr>
<td>1</td>
<td>0x01</td>
</tr>
<tr>
<td>10</td>
<td>0x0a</td>
</tr>
<tr>
<td>23</td>
<td>0x17</td>
</tr>
<tr>
<td>24</td>
<td>0x1818</td>
</tr>
<tr>
<td>25</td>
<td>0x1819</td>
</tr>
<tr>
<td>100</td>
<td>0x1864</td>
</tr>
<tr>
<td>1000</td>
<td>0x1903e8</td>
</tr>
<tr>
<td>1000000</td>
<td>0x1a000f4240</td>
</tr>
<tr>
<td>1000000000000</td>
<td>0x1b0000000e8d4a51000</td>
</tr>
<tr>
<td>18446744073709551615</td>
<td>0x1bffffffffffffffff</td>
</tr>
<tr>
<td>18446744073709551616</td>
<td>0xc249010000000000000000</td>
</tr>
<tr>
<td>-18446744073709551616</td>
<td>0x3bffffffffffffffff</td>
</tr>
<tr>
<td>Value</td>
<td>CBOR</td>
</tr>
<tr>
<td>------------------</td>
<td>-------------</td>
</tr>
<tr>
<td>-18446744073709551617</td>
<td>0xc349010000000000000000</td>
</tr>
<tr>
<td>-1</td>
<td>0x20</td>
</tr>
<tr>
<td>-10</td>
<td>0x29</td>
</tr>
<tr>
<td>-100</td>
<td>0x3863</td>
</tr>
<tr>
<td>-1000</td>
<td>0x3903e7</td>
</tr>
<tr>
<td>0.0</td>
<td>0xf90000</td>
</tr>
<tr>
<td>-0.0</td>
<td>0xf98000</td>
</tr>
<tr>
<td>1.0</td>
<td>0xf93c00</td>
</tr>
<tr>
<td>1.1</td>
<td>0xfb3ff199999999999a</td>
</tr>
<tr>
<td>1.5</td>
<td>0xf93e00</td>
</tr>
<tr>
<td>65504.0</td>
<td>0xf97bff</td>
</tr>
<tr>
<td>100000.0</td>
<td>0xfa47c35000</td>
</tr>
<tr>
<td>3.4028234663852886e+38</td>
<td>0xfa7f7fffff</td>
</tr>
<tr>
<td>1.0e+300</td>
<td>0xfb7e37e43c8800759c</td>
</tr>
<tr>
<td>5.960464477539063e-8</td>
<td>0xf90001</td>
</tr>
<tr>
<td>0.00006103515625</td>
<td>0xf90400</td>
</tr>
<tr>
<td>-4.0</td>
<td>0xf9c400</td>
</tr>
<tr>
<td>-4.1</td>
<td>0xfbc01066666666666</td>
</tr>
<tr>
<td>Infinity</td>
<td>0xf97c00</td>
</tr>
<tr>
<td>NaN</td>
<td>0xf97e00</td>
</tr>
<tr>
<td>-Infinity</td>
<td>0xf9fc00</td>
</tr>
<tr>
<td>Infinity</td>
<td>0xfa7f800000</td>
</tr>
<tr>
<td>NaN</td>
<td>0xfa7fc0000</td>
</tr>
<tr>
<td>-Infinity</td>
<td>0xfa7f800000</td>
</tr>
<tr>
<td>Value</td>
<td>CBOR Value</td>
</tr>
<tr>
<td>-------------</td>
<td>------------</td>
</tr>
<tr>
<td>Infinity</td>
<td>0xfb7ff0000000000000000</td>
</tr>
<tr>
<td>NaN</td>
<td>0xfb7ff8000000000000000</td>
</tr>
<tr>
<td>-Infinity</td>
<td>0xfbfff0000000000000000</td>
</tr>
<tr>
<td>false</td>
<td>0xf4</td>
</tr>
<tr>
<td>true</td>
<td>0xf5</td>
</tr>
<tr>
<td>null</td>
<td>0xf6</td>
</tr>
<tr>
<td>undefined</td>
<td>0xf7</td>
</tr>
<tr>
<td>simple(16)</td>
<td>0xf0</td>
</tr>
<tr>
<td>simple(255)</td>
<td>0xf8ff</td>
</tr>
<tr>
<td>0(&quot;2013-03-21T20:04:00Z&quot;)</td>
<td>0xc074323031332d30332d32315432303a30343a30305a</td>
</tr>
<tr>
<td>1(1363896240)</td>
<td>0xc11a514b67b0</td>
</tr>
<tr>
<td>1(1363896240.5)</td>
<td>0xc1fb41d452d9ec200000</td>
</tr>
<tr>
<td>23(h'01020304')</td>
<td>0xd74401020304</td>
</tr>
<tr>
<td>24(h'6449455446')</td>
<td>0xd818456449455446</td>
</tr>
<tr>
<td>32(&quot;<a href="http://www.example.com">http://www.example.com</a>&quot;)</td>
<td>0xd82076687474703a2f2f7777772e657861d706c652e636f6d</td>
</tr>
<tr>
<td>h''</td>
<td>0x40</td>
</tr>
<tr>
<td>h'01020304'</td>
<td>0x4401020304</td>
</tr>
<tr>
<td>&quot;&quot;</td>
<td>0x60</td>
</tr>
<tr>
<td>&quot;a&quot;</td>
<td>0x6161</td>
</tr>
<tr>
<td>&quot;IETF&quot;</td>
<td>0x6449455446</td>
</tr>
<tr>
<td>&quot;\&quot;</td>
<td>0x62225c</td>
</tr>
<tr>
<td>&quot;\u00fc&quot;</td>
<td>0x62c3bc</td>
</tr>
<tr>
<td>&quot;\u6c34&quot;</td>
<td>0x63e6b0b4</td>
</tr>
</tbody>
</table>
"\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] | 0x98190102030405060708090a0b0c0d0e0f10111213141516171819181919
{} | 0xa0
{1: 2, 3: 4} | 0xa201020304
{"a": 1, "b": [2, 3]} | 0xa26161016162820203
{"a": ["A", "B", "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] | 0x9f0102030405060708090a0b0c0d0e0f1011121314151617181819181919
{"a": 1, "b": [ _ 2, 3] } | 0xbf61610161629f0203ffff
{"a", [ _ "b": \"c\"] } | 0xbf6346756ef563416d7421ff
{"Fun": true, "Amt": -2} | 0xbf6346756ef563416d7421ff
### Table 5: 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.)

<table>
<thead>
<tr>
<th>Byte</th>
<th>Structure/Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00..0x17</td>
<td>Integer 0x00..0x17 (0..23)</td>
</tr>
<tr>
<td>0x18</td>
<td>Unsigned integer (one-byte uint8_t follows)</td>
</tr>
<tr>
<td>0x19</td>
<td>Unsigned integer (two-byte uint16_t follows)</td>
</tr>
<tr>
<td>0x1a</td>
<td>Unsigned integer (four-byte uint32_t follows)</td>
</tr>
<tr>
<td>0x1b</td>
<td>Unsigned integer (eight-byte uint64_t follows)</td>
</tr>
<tr>
<td>0x20..0x37</td>
<td>Negative integer -1-0x00..-1-0x17 (-1..-24)</td>
</tr>
<tr>
<td>0x38</td>
<td>Negative integer -1-n (one-byte uint8_t for n follows)</td>
</tr>
<tr>
<td>0x39</td>
<td>Negative integer -1-n (two-byte uint16_t for n follows)</td>
</tr>
<tr>
<td>0x3a</td>
<td>Negative integer -1-n (four-byte uint32_t for n follows)</td>
</tr>
<tr>
<td>0x3b</td>
<td>Negative integer -1-n (eight-byte uint64_t for n follows)</td>
</tr>
<tr>
<td>0x40..0x57</td>
<td>byte string (0x00..0x17 bytes follow)</td>
</tr>
<tr>
<td>0x58</td>
<td>byte string (one-byte uint8_t for n, and then n bytes follow)</td>
</tr>
<tr>
<td>0x59</td>
<td>byte string (two-byte uint16_t for n, and then n bytes follow)</td>
</tr>
<tr>
<td>0x5a</td>
<td>byte string (four-byte uint32_t for n, and then n bytes follow)</td>
</tr>
<tr>
<td><strong>0x5b</strong></td>
<td>byte string (eight-byte uint64_t for n, and then n bytes follow)</td>
</tr>
<tr>
<td><strong>0x5f</strong></td>
<td>byte string, byte strings follow, terminated by &quot;break&quot;</td>
</tr>
<tr>
<td><strong>0x60..0x77</strong></td>
<td>UTF-8 string (0x00..0x17 bytes follow)</td>
</tr>
<tr>
<td><strong>0x78</strong></td>
<td>UTF-8 string (one-byte uint8_t for n, and then n bytes follow)</td>
</tr>
<tr>
<td><strong>0x79</strong></td>
<td>UTF-8 string (two-byte uint16_t for n, and then n bytes follow)</td>
</tr>
<tr>
<td><strong>0x7a</strong></td>
<td>UTF-8 string (four-byte uint32_t for n, and then n bytes follow)</td>
</tr>
<tr>
<td><strong>0x7b</strong></td>
<td>UTF-8 string (eight-byte uint64_t for n, and then n bytes follow)</td>
</tr>
<tr>
<td><strong>0x7f</strong></td>
<td>UTF-8 string, UTF-8 strings follow, terminated by &quot;break&quot;</td>
</tr>
<tr>
<td><strong>0x80..0x97</strong></td>
<td>array (0x00..0x17 data items follow)</td>
</tr>
<tr>
<td><strong>0x98</strong></td>
<td>array (one-byte uint8_t for n, and then n data items follow)</td>
</tr>
<tr>
<td><strong>0x99</strong></td>
<td>array (two-byte uint16_t for n, and then n data items follow)</td>
</tr>
<tr>
<td><strong>0x9a</strong></td>
<td>array (four-byte uint32_t for n, and then n data items follow)</td>
</tr>
<tr>
<td><strong>0x9b</strong></td>
<td>array (eight-byte uint64_t for n, and then n data items follow)</td>
</tr>
<tr>
<td><strong>0x9f</strong></td>
<td>array, data items follow, terminated by &quot;break&quot;</td>
</tr>
<tr>
<td><strong>0xa0..0xb7</strong></td>
<td>map (0x00..0x17 pairs of data items follow)</td>
</tr>
<tr>
<td><strong>0xb8</strong></td>
<td>map (one-byte uint8_t for n, and then n pairs of data items follow)</td>
</tr>
<tr>
<td><strong>0xb9</strong></td>
<td>map (two-byte uint16_t for n, and then n pairs of data items follow)</td>
</tr>
<tr>
<td>0xba</td>
<td>map (four-byte uint32_t for n, and then n pairs of data items follow)</td>
</tr>
<tr>
<td>0xbb</td>
<td>map (eight-byte uint64_t for n, and then n pairs of data items follow)</td>
</tr>
<tr>
<td>0xbf</td>
<td>map, pairs of data items follow, terminated by &quot;break&quot;</td>
</tr>
<tr>
<td>0xc0</td>
<td>Text-based date/time (data item follows; see Section 3.4.2)</td>
</tr>
<tr>
<td>0xc1</td>
<td>Epoch-based date/time (data item follows; see Section 3.4.3)</td>
</tr>
<tr>
<td>0xc2</td>
<td>Positive bignum (data item &quot;byte string&quot; follows)</td>
</tr>
<tr>
<td>0xc3</td>
<td>Negative bignum (data item &quot;byte string&quot; follows)</td>
</tr>
<tr>
<td>0xc4</td>
<td>Decimal Fraction (data item &quot;array&quot; follows; see Section 3.4.5)</td>
</tr>
<tr>
<td>0xc5</td>
<td>Bigfloat (data item &quot;array&quot; follows; see Section 3.4.5)</td>
</tr>
<tr>
<td>0xc6..0xd4</td>
<td>(tagged item)</td>
</tr>
<tr>
<td>0xd5..0xd7</td>
<td>Expected Conversion (data item follows; see Section 3.4.6.2)</td>
</tr>
<tr>
<td>0xd8..0xdb</td>
<td>(more tagged items, 1/2/4/8 bytes and then a data item follow)</td>
</tr>
<tr>
<td>0xe0..0xf3</td>
<td>(simple value)</td>
</tr>
<tr>
<td>0xf4</td>
<td>False</td>
</tr>
<tr>
<td>0xf5</td>
<td>True</td>
</tr>
<tr>
<td>0xf6</td>
<td>Null</td>
</tr>
<tr>
<td>0xf7</td>
<td>Undefined</td>
</tr>
<tr>
<td>0xf8</td>
<td>(simple value, one byte follows)</td>
</tr>
<tr>
<td>0xf9</td>
<td>Half-Precision Float (two-byte IEEE 754)</td>
</tr>
<tr>
<td>0xfa</td>
<td>Single-Precision Float (four-byte IEEE 754)</td>
</tr>
</tbody>
</table>
## Table 6: Jump Table for Initial Byte

<table>
<thead>
<tr>
<th>0xfb</th>
<th>Double-Precision Float (eight-byte IEEE 754)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xff</td>
<td>&quot;break&quot; stop code</td>
</tr>
</tbody>
</table>

### 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:

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

The pseudocode has the following prerequisites:

1. `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.
2. `uint()` converts a byte string into an unsigned integer by interpreting the byte string in network byte order.
3. Arithmetic works as in C.
4. All variables are unsigned integers of sufficient range.

Note that "well_formed" returns the major type for well-formed definite length items, but 0 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;                       // 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-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 0;                        // no break out
}

Figure 1: Pseudocode for Well-Formedness Check
Note that the remaining complexity of a complete CBOR decoder is about presenting data that has been decoded to the application in an appropriate form.

Major types 0 and 1 are designed in such a way that they can be encoded in C from a signed integer without actually doing an if-then-else for positive/negative (Figure 2). This uses the fact that (-1-n), the transformation for major type 1, is the same as ^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).

```c
void encode_sint(int64_t n) {
    uint64_t ui = n >> 63;    // extend sign to whole length
    mt = ui & 0x20;          // extract major type
    ui ^= n;                 // complement negatives
    if (ui < 24)
        *p++ = mt + ui;
    else if (ui < 256) {
        *p++ = mt + 24;
        *p++ = ui;
    } else
...
```

Figure 2: Pseudocode for Encoding a Signed Integer

Appendix D. Half-Precision

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

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

Figure 3: C Code for a Half-Precision Decoder

import struct
from math import ldexp

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

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

Figure 4: Python Code for a Half-Precision Decoder

Appendix E. Comparison of Other Binary Formats to CBOR’s Design

Objectives

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

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

Note that the discussion here is not meant to be a criticism of any format: to the best of our knowledge, no format before CBOR was meant
to cover CBOR’s objectives in the priority we have assigned them. A brief recap of the objectives from Section 1.1 is:

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

E.1. ASN.1 DER, BER, and PER

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

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

E.2. MessagePack

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

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

E.3. BSON

[BSON] is a data format that was developed for the storage of JSON-like maps (JSON objects) in the MongoDB database. Its major distinguishing feature is the capability for in-place update, 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 7 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.
<table>
<thead>
<tr>
<th>Format</th>
<th>[1, [2, 3]]</th>
<th>[_ 1, [2, 3]]</th>
</tr>
</thead>
<tbody>
<tr>
<td>RFC 713</td>
<td>c2 05 81 c2 02 82 83</td>
<td>30 80 02 01 01 30 06 02</td>
</tr>
<tr>
<td>ASN.1 BER</td>
<td>30 0b 02 01 01 30 06 02 01 02 01 03</td>
<td>01 02 01 03 00 00</td>
</tr>
<tr>
<td>MessagePack</td>
<td>92 01 92 02 03</td>
<td></td>
</tr>
<tr>
<td>BSON</td>
<td>22 00 00 00 10 30 00 01 00 00 00 04 31 00 13 00 00 00 00 00 03 00 00 00</td>
<td></td>
</tr>
<tr>
<td>CBOR</td>
<td>82 01 82 02 03</td>
<td>9f 01 82 02 03 ff</td>
</tr>
</tbody>
</table>

Table 7: 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 [RFC4627] to [RFC8259] 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_11001" -> "0b000_11001")

Acknowledgements

CBOR was inspired by MessagePack. MessagePack was developed and promoted by Sadayuki Furuhashi ("frsyuki"). This reference to MessagePack is solely for attribution; CBOR is not intended as a version of or replacement for MessagePack, as it has different design goals and requirements.
The need for functionality beyond the original MessagePack Specification became obvious to many people at about the same time around the year 2012. BinaryPack is a minor derivation of MessagePack that was developed by Eric Zhang for the binaryjs project. A similar, but different, extension was made by Tim Caswell for his msgpack-js and msgpack-js-browser projects. Many people have contributed to the discussion about extending MessagePack to separate text string representation from byte string representation.

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

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

Authors’ Addresses

Carsten Bormann
Universitaet Bremen T2I
Postfach 330440
D-28359 Bremen
Germany
Phone: +49-421-218-63921
EMail: cabo@tzi.org

Paul Hoffman
ICANN

EMail: paul.hoffman@icann.org
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.
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 has 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 can also save space overhead with encoding a type for each element of an array.

This document defines a number of interrelated CBOR tags that cover these typed arrays, as well as two additional tags for multi-

1.1. Terminology
dimensional and homogeneous arrays. It is intended as the reference document for the IANA registration of the tags defined.

Note that an application that generates CBOR with these tags has considerable freedom in choosing variants, e.g., with respect to endianness, embedded type (signed vs. unsigned), and number of bits per element, or whether a tag defined in this specification is used at all instead of more basic CBOR. In contrast to representation variants of single CBOR numbers, there is no representation that could be identified as "preferred". If deterministic encoding is desired in a CBOR-based protocol making use of these tags, the protocol has to define which of the encoding variants are used in which case.

1.1. 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". Where bit arithmetic is explained, this document uses the notation familiar from the programming language C [C] (including C++14’s 0bnnn binary literals [Cplusplus]), except that the operator "**" stands for exponentiation.

The term "array" is used in a general sense in this document, unless further specified. The term "classical CBOR array" describes an array represented with CBOR major type 4. A "homogeneous array" is an array of elements that are all of the same type (the term is neutral as to whether that is a representation type or an application data model type).

The terms "big endian" and "little endian" are used to indicate a most significant byte first (MSB first) representation of integers, and a least significant byte first (LSB first) representation, respectively.

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 (two’s 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:

<table>
<thead>
<tr>
<th>Length ll</th>
<th>uint</th>
<th>sint</th>
<th>float</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>uint8</td>
<td>sint8</td>
<td>binary16</td>
</tr>
<tr>
<td>1</td>
<td>uint16</td>
<td>sint16</td>
<td>binary32</td>
</tr>
<tr>
<td>2</td>
<td>uint32</td>
<td>sint32</td>
<td>binary64</td>
</tr>
<tr>
<td>3</td>
<td>uint64</td>
<td>sint64</td>
<td>binary128</td>
</tr>
</tbody>
</table>

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 [IEEE754].

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 64 to 87.

The value is split up into 5 bit fields: 0b010_f_s_e_ll, as detailed in Table 2.
### Table 2: Bit fields in the low 8 bits of the tag

<table>
<thead>
<tr>
<th>Field</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b010</td>
<td>the constant bits 0, 1, 0</td>
</tr>
<tr>
<td>f</td>
<td>0 for integer, 1 for float</td>
</tr>
<tr>
<td>s</td>
<td>0 for float or unsigned integer, 1 for signed integer</td>
</tr>
<tr>
<td>e</td>
<td>0 for big endian, 1 for little endian</td>
</tr>
<tr>
<td>ll</td>
<td>A number for the length (Table 1).</td>
</tr>
</tbody>
</table>

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 0f and ll are the two least significant bits, respectively, 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 from the length, in bytes, of the byte string comprising the representation of the array by inverting the previous formula:  
"bytelength >> (f + ll)".

For the uint8/sint8 values, the endianness is redundant. Only the tag for the big endian variant is used and assigned as such. The Tag that would signify the little endian variant of sint8 MUST NOT be used, its tag number is marked as reserved. As a special case, the Tag that would signify the little endian variant of uint8 is instead assigned to signify that the numbers in the array are using clamped conversion from integers, as described in more detail in Section 7.1.11 ("ToUint8Clamp") of the ES6 JavaScript specification [TypedArrayES6]; the assumption here is that a program-internal representation of this array after decoding would be marked this way for further processing, providing "roundtripping" of JavaScript typed arrays through CBOR.

IEEE 754 binary floating numbers are always signed. Therefore, for the float variants ("f" == 1), there is no need to distinguish between signed and unsigned variants; the "s" bit is always zero. The Tag numbers where "s" would be one (which would have Tag values 88 to 95) remain free to use by other specifications.

### 3. Additional Array Tags

This specification defines three additional array tags. The Multi-dimensional Array tags 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 as a signal by an application to identify a classical CBOR array as a homogeneous array, even when a Typed Array does not apply.

3.1. Multi-dimensional Array

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.

Two tags are defined by this document, one for elements arranged in row-major order, and one for column-major order [RowColMajor].

3.1.1. Row-major Order

Tag: 40

Data Item: array (major type 4) of two arrays, one array (major type 4) of dimensions, which are unsigned integers distinct from zero, and one array (either a CBOR array of major type 4, or a Typed Array, or a Homogeneous Array) of elements

Data in the second array consists of consecutive values where the last dimension is considered contiguous (row-major order).

Figure 1 shows a declaration of a two-dimensional array in the C language, a representation of that in CBOR using both a multidimensional array tag and a typed array tag.
uint16_t a[2][3] = {
    {2, 4, 8},  /* row 0 */
    {4, 16, 256},
};

<Tag 40> # multi-dimensional array tag
82       # array(2)
82       # array(2)
02       # unsigned(2) 1st Dimension
03       # unsigned(3) 2nd Dimension
<Tag 65> # uint16 array
4c       # byte string(12)
0002     # unsigned(2)
0004     # unsigned(4)
0008     # unsigned(8)
0004     # unsigned(4)
0010     # unsigned(16)
0100     # unsigned(256)

Figure 1: Multi-dimensional array in C and CBOR

Figure 2 shows the same two-dimensional array using the multidimensional array tag in conjunction with a basic CBOR array (which, with the small numbers chosen for the example, happens to be shorter).

<Tag 40> # multi-dimensional array tag
82       # array(2)
82       # array(2)
02       # unsigned(2) 1st Dimension
03       # unsigned(3) 2nd Dimension
86       # array(6)
02       # unsigned(2)
04       # unsigned(4)
08       # unsigned(8)
04       # unsigned(4)
10       # unsigned(16)
19 0100  # unsigned(256)

Figure 2: Multi-dimensional array using basic CBOR array

3.1.2. Column-Major order

The multidimensional arrays specified in the previous sub-subsection are in "row major" order, which is the preferred order for the purposes of this specification. An analogous representation that uses "column major" order arrays is provided in this subsection under the tag 1040, as illustrated in Figure 3.
Tag: 1040

Data Item: as with tag 40, except that the data in the second array consists of consecutive values where the first dimension is considered contiguous (column-major order).

<Tag 1040> # multi-dimensional array tag, column major order
  82     # array(2)
  82     # array(2)
  02     # unsigned(2) 1st Dimension
  03     # unsigned(3) 2nd Dimension
  86     # array(6)
  02     # unsigned(2)
  04     # unsigned(4)
  04     # unsigned(4)
  10     # unsigned(16)
  08     # unsigned(8)
  19 0100 # unsigned(256)

Figure 3: Multi-dimensional array using basic CBOR array, column major order

3.2. Homogeneous Array

Tag: 41

Data Item: array (major type 4)

This tag identifies the classical CBOR array (a one-dimensional array) tagged by it as a homogeneous array, that is, it has elements that are all of the same application model data type. The element type of the array is thus determined by the application model data type of the first array element.

This can be used in application data models that apply specific semantics to homogeneous arrays. Also, in certain cases, implementations in strongly typed languages may be able to create native homogeneous arrays of specific types instead of ordered lists while decoding. Which CBOR data items constitute elements of the same application type is specific to the application.

Figure 4 shows an example for a homogeneous array of booleans in C++ [Cplusplus] and CBOR.
bool boolArray[2] = { true, false };

<Tag 41>  # Homogeneous Array Tag
82       # array(2)
F5       # true
F4       # false

Figure 4: Homogeneous array in C++ and CBOR

Figure 5 extends the example with a more complex structure.

typedef struct {
    bool active;
    int value;
} foo;
foo myArray[2] = { {true, 3}, {true, -4} };

<Tag 41>
82  # array(2)
82  # array(2)
F5  # true
03  # 3
82  # array(2)
F5  # true
23  # -4

Figure 5: 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.

Providing a column-major order variant of the multi-dimensional array may seem superfluous to some, and useful to others. It is cheap to define the additional tag so it is available when actually needed. Allocating it out of a different number space makes the preference for row-major evident.

Applying a Homogeneous Array tag to a Typed Array would usually be redundant and is therefore not provided by the present specification.
5. CDDL typenames

For the use with CDDL [RFC8610], the typenames defined in Figure 6 are recommended:

```plaintext
ta-uint8 = #6.64(bstr)
ta-uint16be = #6.65(bstr)
ta-uint32be = #6.66(bstr)
ta-uint64be = #6.67(bstr)
ta-uint8-clamped = #6.68(bstr)
ta-uint16le = #6.69(bstr)
ta-uint32le = #6.70(bstr)
ta-uint64le = #6.71(bstr)
ta-sint0 = #6.72(bstr)
ta-sint16be = #6.73(bstr)
ta-sint32be = #6.74(bstr)
ta-sint64be = #6.75(bstr)
; reserved: #6.76(bstr)
ta-sint16le = #6.77(bstr)
ta-sint32le = #6.78(bstr)
ta-sint64le = #6.79(bstr)
ta-float16be = #6.80(bstr)
ta-float32be = #6.81(bstr)
ta-float64be = #6.82(bstr)
ta-float128be = #6.83(bstr)
ta-float16le = #6.84(bstr)
ta-float32le = #6.85(bstr)
ta-float64le = #6.86(bstr)
ta-float128le = #6.87(bstr)
```

homogeneous<array> = #6.41(array)
multi-dim<dim, array> = #6.40([dim, array])
multi-dim-column-major<dim, array> = #6.1040([dim, array])

Figure 6: Recommended typenames for CDDL
6. IANA Considerations

IANA has allocated the tags in Table 3, with the present document as the specification reference. (The reserved value is reserved for a future revision of typed array tags.)

The allocations came out of the "specification required" space (24..255), with the exception of 1040, which came out of the "first come first served" space (256..).
<table>
<thead>
<tr>
<th>Tag</th>
<th>Data Item</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>64</td>
<td>byte string</td>
<td>uint8 Typed Array</td>
</tr>
<tr>
<td>65</td>
<td>byte string</td>
<td>uint16, big endian, Typed Array</td>
</tr>
<tr>
<td>66</td>
<td>byte string</td>
<td>uint32, big endian, Typed Array</td>
</tr>
<tr>
<td>67</td>
<td>byte string</td>
<td>uint64, big endian, Typed Array</td>
</tr>
<tr>
<td>68</td>
<td>byte string</td>
<td>uint8 Typed Array, clamped arithmetic</td>
</tr>
<tr>
<td>69</td>
<td>byte string</td>
<td>uint16, little endian, Typed Array</td>
</tr>
<tr>
<td>70</td>
<td>byte string</td>
<td>uint32, little endian, Typed Array</td>
</tr>
<tr>
<td>71</td>
<td>byte string</td>
<td>uint64, little endian, Typed Array</td>
</tr>
<tr>
<td>72</td>
<td>byte string</td>
<td>sint8 Typed Array</td>
</tr>
<tr>
<td>73</td>
<td>byte string</td>
<td>sint16, big endian, Typed Array</td>
</tr>
<tr>
<td>74</td>
<td>byte string</td>
<td>sint32, big endian, Typed Array</td>
</tr>
<tr>
<td>75</td>
<td>byte string</td>
<td>sint64, big endian, Typed Array</td>
</tr>
<tr>
<td>76</td>
<td>byte string</td>
<td>(reserved)</td>
</tr>
<tr>
<td>77</td>
<td>byte string</td>
<td>sint16, little endian, Typed Array</td>
</tr>
<tr>
<td>78</td>
<td>byte string</td>
<td>sint32, little endian, Typed Array</td>
</tr>
<tr>
<td>79</td>
<td>byte string</td>
<td>sint64, little endian, Typed Array</td>
</tr>
<tr>
<td>80</td>
<td>byte string</td>
<td>IEEE 754 binary16, big endian, Typed Array</td>
</tr>
<tr>
<td>81</td>
<td>byte string</td>
<td>IEEE 754 binary32, big endian, Typed Array</td>
</tr>
<tr>
<td>82</td>
<td>byte string</td>
<td>IEEE 754 binary64, big endian, Typed Array</td>
</tr>
<tr>
<td>83</td>
<td>byte string</td>
<td>IEEE 754 binary128, big endian, Typed Array</td>
</tr>
<tr>
<td>84</td>
<td>byte string</td>
<td>IEEE 754 binary16, little endian, Typed Array</td>
</tr>
<tr>
<td>85</td>
<td>byte string</td>
<td>IEEE 754 binary32, little endian, Typed Array</td>
</tr>
<tr>
<td>86</td>
<td>byte string</td>
<td>IEEE 754 binary64, little endian, Typed Array</td>
</tr>
<tr>
<td>87</td>
<td>byte string</td>
<td>IEEE 754 binary128, little endian, Typed Array</td>
</tr>
<tr>
<td>40</td>
<td>array of two</td>
<td>Multi-dimensional Array, row-major order</td>
</tr>
<tr>
<td></td>
<td>arrays*</td>
<td>Multi-dimensional Array, column-major order</td>
</tr>
<tr>
<td>1040</td>
<td>array of two</td>
<td></td>
</tr>
<tr>
<td></td>
<td>arrays*</td>
<td></td>
</tr>
<tr>
<td>41</td>
<td>array</td>
<td>Homogeneous Array</td>
</tr>
</tbody>
</table>

Table 3: Values for Tags

*) 40 or 1040 data item: second element of outer array in data item is native CBOR array (major type 4) or Typed Array (one of Tag 64..87)
7. Security Considerations

The security considerations of RFC 7049 apply; special attention is drawn to the second paragraph of Section 8 of RFC 7049.

The Tag for homogeneous arrays makes a promise about its tagged data item that a maliciously constructed CBOR input can then choose to ignore. As always, the decoder therefore has to ensure that it is not driven into an undefined state by array elements that do not fulfill the promise and that it does continue to fulfill its API contract in this case as well.

As with all formats that are used for data interchange, an attacker may have control over the shape of the data delivered as input to the application, which therefore needs to validate that shape before it makes it the basis of its further processing. One unique aspect that typed arrays add to this is that an attacker might substitute a Uint8ClampedArray for where the application expects a Uint8Array, or vice versa, potentially leading to very different (and unexpected) processing semantics of the in-memory data structures constructed. Applications that could be affected by this therefore will need to be careful about making this distinction in their input validation.
8. References

8.1. Normative References


8.2. Informative References

Contributors

The initial draft for this specification was written by Johnathan Roatch (roatch@gmail.com). Many thanks for getting this ball rolling.

Glenn Engel suggested the tags for multi-dimensional arrays and homogeneous arrays.

Acknowledgements

Jim Schaad provided helpful comments and reminded us that column-major order still is in use. Jeffrey Yaskin helped improve the definition of homogeneous arrays. IANA helped correct an error in a previous version. Francesca Palombini acted as a shepherd, and Alexey Melnikov as responsible area director. Elwyn Davies as Gen-ART reviewer and IESG members Martin Vigoureux, Adam Roach, Roman Danyliw, and Benjamin Kaduk helped finding further improvements of the text; thanks also to the other reviewers.

Author’s Address

Carsten Bormann (editor)
Universitaet Bremen TZI
Postfach 330440
Bremen D-28359
Germany

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

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 25, 2019.

Copyright Notice

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

This document is subject to BCP 78 and the IETF Trust’s Legal Provisions Relating to IETF Documents (https://trustee.ietf.org/license-info) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect
Internet-Draft                    CDDL                        March 2019

to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

Table of Contents

1.  Introduction ........................................ 3
   1.1.  Requirements notation .......................... 4
   1.2.  Terminology .................................... 4
2.  The Style of Data Structure Specification .............. 4
   2.1.  Groups and Composition in CDDL .................. 6
       2.1.1.  Usage .................................... 9
       2.1.2.  Syntax ................................... 9
   2.2.  Types .......................................... 9
       2.2.1.  Values .................................. 10
       2.2.2.  Choices .................................. 10
       2.2.3.  Representation Types ....................... 12
       2.2.4.  Root type ................................ 13
3.  Syntax ............................................. 13
   3.1.  General conventions ................................ 13
   3.2.  Occurrence ..................................... 15
   3.3.  Predefined names for types ..................... 16
   3.4.  Arrays ......................................... 17
   3.5.  Maps ........................................... 17
       3.5.1.  Structs .................................. 18
       3.5.2.  Tables ................................... 20
       3.5.3.  Non-deterministic order .................... 21
       3.5.4.  Cuts in Maps ................................ 22
   3.6.  Tags ............................................ 23
   3.7.  Unwrapping ..................................... 24
   3.8.  Controls ....................................... 25
       3.8.1.  Control operator .size ...................... 25
       3.8.2.  Control operator .bits ..................... 26
       3.8.3.  Control operator .regexp .................... 26
       3.8.4.  Control operators .cbor and .cborseq ....... 28
       3.8.5.  Control operators .within and .and ........ 28
       3.8.6.  Control operators .lt, .le, .gt, .ge, .eq, .ne, and .default ......................... 29
   3.9.  Socket/Plug ..................................... 30
   3.10. Generics ........................................ 31
   3.11. Operator Precedence ................................ 32
4.  Making Use of CDDL .................................... 33
   4.1.  As a guide to a human user ........................ 33
   4.2.  For automated checking of CBOR data structure .... 34
   4.3.  For data analysis tools .......................... 34
5.  Security considerations ................................ 34
6.  IANA Considerations ................................... 35

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.
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
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:
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).

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

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

Figure 5: Maps with copy/paste
```

```plaintext
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
bytel = 0...first-non-byte ; bytel 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 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:

```cddl
; 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:

```cddl
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]
awt-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 encountered 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 embedded:

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

```plaintext
GpsCoordinates = {
    longitude : uint, ; degrees, scaled by 10^7
    latitude : uint, ; degress, 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.:

```plaintext
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:

\[
\text{PersonalData} = \{
\quad ? \text{displayName}: \text{tstr},
\quad \text{NameComponents},
\quad ? \text{age}: \text{uint},
\quad * \text{tstr} \Rightarrow \text{any}
\}
\]

\[
\text{NameComponents} = \{
\quad ? \text{firstName}: \text{tstr},
\quad ? \text{familyName}: \text{tstr},
\}
\]

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.:

\[
\text{square-roots} = \{ * \text{x} \Rightarrow \text{y} \}
\]
\[
\quad \text{x} = \text{int}
\]
\[
\quad \text{y} = \text{float}
\]

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 wildcard 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:

```plaintext
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:

```plaintext
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'
```

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'00'" 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].


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

\[
t = \{\text{group1}\}
\]

\[
\text{group1} = (a / b // c / d)
\]

\[
a = 1 \quad b = 2 \quad c = 3 \quad d = 4
\]

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

\[
t = \{\text{group2}\}
\]

\[
\text{group2} = (? \text{ab}: a / b // \text{cd}: c / d)
\]

\[
a = 1 \quad b = 2 \quad c = 3 \quad d = 4
\]

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

\[
t = \{\text{group3}\}
\]

\[
\text{group3} = (+ a / b / c)
\]

\[
a = 1 \quad b = 2 \quad c = 3
\]

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

\[
(A \text{comment has been that this could be counter-intuitive. The specification writer is encouraged to use parentheses liberally to guide readers that are not familiar with CDDL precedence rules.})
\]

\[
t = \{\text{group4}\}
\]

\[
\text{group4} = (+ a / b // c)
\]

\[
a = 1 \quad b = 2 \quad c = 3
\]
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:

\[
t = [\text{group4a}]
\]
\[
\text{group4a} = ((+ a) \div (b \div c))
\]
\[
a = 1 \, b = 2 \, c = 3
\]

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.)

<table>
<thead>
<tr>
<th>Operator</th>
<th>Ar</th>
<th>Operates on</th>
<th>Prec</th>
</tr>
</thead>
<tbody>
<tr>
<td>=</td>
<td>2</td>
<td>name = type, name = group</td>
<td>1</td>
</tr>
<tr>
<td>/=</td>
<td>2</td>
<td>name /= type</td>
<td>1</td>
</tr>
<tr>
<td>//=</td>
<td>2</td>
<td>name //= group</td>
<td>1</td>
</tr>
<tr>
<td>//</td>
<td>2</td>
<td>group // group</td>
<td>2</td>
</tr>
<tr>
<td>,</td>
<td>2</td>
<td>group, group</td>
<td>3</td>
</tr>
<tr>
<td>*</td>
<td>1</td>
<td>* group</td>
<td>4</td>
</tr>
<tr>
<td>N*M</td>
<td>1</td>
<td>N*M group</td>
<td>4</td>
</tr>
<tr>
<td>+</td>
<td>1</td>
<td>+ group</td>
<td>4</td>
</tr>
<tr>
<td>?</td>
<td>1</td>
<td>? group</td>
<td>4</td>
</tr>
<tr>
<td>=&gt;</td>
<td>2</td>
<td>type =&gt; type</td>
<td>5</td>
</tr>
<tr>
<td>:</td>
<td>2</td>
<td>name: type</td>
<td>5</td>
</tr>
<tr>
<td>/</td>
<td>2</td>
<td>type / type</td>
<td>6</td>
</tr>
<tr>
<td>..</td>
<td>2</td>
<td>type..type</td>
<td>7</td>
</tr>
<tr>
<td>...</td>
<td>2</td>
<td>type...type</td>
<td>7</td>
</tr>
<tr>
<td>.ctrl</td>
<td>2</td>
<td>type .ctrl type</td>
<td>7</td>
</tr>
<tr>
<td>&amp;</td>
<td>1</td>
<td>&amp;group</td>
<td>8</td>
</tr>
<tr>
<td>-</td>
<td>1</td>
<td>-type</td>
<td>8</td>
</tr>
</tbody>
</table>

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:
<table>
<thead>
<tr>
<th>name</th>
<th>documentation</th>
</tr>
</thead>
<tbody>
<tr>
<td>.size</td>
<td>[RFCthis]</td>
</tr>
<tr>
<td>.bits</td>
<td>[RFCthis]</td>
</tr>
<tr>
<td>.regexp</td>
<td>[RFCthis]</td>
</tr>
<tr>
<td>.cbor</td>
<td>[RFCthis]</td>
</tr>
<tr>
<td>.cborseq</td>
<td>[RFCthis]</td>
</tr>
<tr>
<td>.within</td>
<td>[RFCthis]</td>
</tr>
<tr>
<td>.and</td>
<td>[RFCthis]</td>
</tr>
<tr>
<td>.lt</td>
<td>[RFCthis]</td>
</tr>
<tr>
<td>.le</td>
<td>[RFCthis]</td>
</tr>
<tr>
<td>.gt</td>
<td>[RFCthis]</td>
</tr>
<tr>
<td>.ge</td>
<td>[RFCthis]</td>
</tr>
<tr>
<td>.eq</td>
<td>[RFCthis]</td>
</tr>
<tr>
<td>.ne</td>
<td>[RFCthis]</td>
</tr>
<tr>
<td>.default</td>
<td>[RFCthis]</td>
</tr>
</tbody>
</table>

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


7.2. Informative References


7.2. Informative References

7.3. URIs

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:

<table>
<thead>
<tr>
<th>CDDL</th>
<th>PEG</th>
<th>Remark</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;=&quot;</td>
<td>&quot;&lt;-&quot;</td>
<td>/= and //= are abbreviations</td>
</tr>
<tr>
<td>&quot;/&quot;</td>
<td>&quot;/&quot;</td>
<td>prioritized choice</td>
</tr>
<tr>
<td>&quot;?&quot; P</td>
<td>P &quot;?&quot;</td>
<td>zero or one</td>
</tr>
<tr>
<td>&quot;*&quot; P</td>
<td>P &quot;*&quot;</td>
<td>zero or more</td>
</tr>
<tr>
<td>&quot;+&quot; P</td>
<td>P &quot;+&quot;</td>
<td>one or more</td>
</tr>
<tr>
<td>A B</td>
<td>A B</td>
<td>sequence</td>
</tr>
<tr>
<td>A, B</td>
<td>A B</td>
<td>sequence, comma is decoration only</td>
</tr>
</tbody>
</table>

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).

```plaintext
cddl = S 1*(rule S)
rule = typename [genericparm] S assignt S type
    / groupname [genericparm] S assigng S grpet

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

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

grpchoice *(S "/" S grpchoice)
grpchoice = *(grpet optcom)
grpet = [occur S] [memberkey S] type
```

Birkholz, et al. Expires September 25, 2019
/ [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"

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
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).

\[
type2 = \text{value}
\]

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),

\[
/ \text{typename} \ [\text{genericarg}]
\]

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

\[
/ (" S \text{type} S ")
\]

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

\[
/ (" S \text{group} S ")
\]

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

\[
/ [" S \text{group} S "]
\]

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

\[
/ "\-" S \text{typename} \ [\text{genericarg}]
\]

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

\[
/ "\&" S (" S \text{group} S ")
/ "\&" S \text{groupname} \ [\text{genericarg}]
\]

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

\[
/ "\#" "6" ["." \text{uint}] (" S \text{type} S ")
\]

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

\[
/ "\#" \text{DIGIT} ["." \text{uint}] \quad ; \text{major/ai}
\]
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 "...".

clop = "." 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).

groupchoice = *(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]
        / "+" [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:

```
'48656c6c6f20776f726c64'
'48 65 6c 6c 6f 20 77 6f 72 6c 64'
'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:

```
'h'hello world'
'hello world'
```

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:

\[
\begin{array}{ll}
<<1>> & h'01' \\
<<1, 2>> & h'0102' \\
<<"foo", null>> & h'63666F6F66' \\
<<>> & h''
\end{array}
\]

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"
"Hello” h’20’ “world"

Similarly, the following byte string values are equivalent

'Hello world'
'Hello ' 'world'
'Hello‘ h’776f726c64’
'Hello' h'20' 'world'
'’ h'48656c6f20776f726c64' ' 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  
0b100100110011

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:
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:
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:

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

size = {
    Width: 0..1280,
    Height: 0..1024
}

thumbnail = {
    Thumbnail: { 
        size, 
        Url: "uri"
    }
}
```

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 Keränen, 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