Network Working Group
Internet-Draft
Updates: 7049 (if approved)
Intended status: Standards Track
Expires: September 14, 2017

C. Bormann
Universitaet Bremen TZI
S. Leonard
Penango, Inc.

March 13, 2017

Concise Binary Object Representation (CBOR) Tags and Techniques for Object Identifiers, UUIDs, Enumerations, Binary Entities, Regular Expressions, and Sets
draft-bormann-cbor-tags-oid-06

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.

Useful tags and techniques have emerged since the publication of RFC 7049; the present document makes use of CBOR's built-in major types to define and refine several useful constructs, without changing the wire protocol. This document adds object identifiers (OIDs) to CBOR with CBOR tags <<O>> and <<R>> [values TBD]. It is intended as the reference document for the IANA registration of the CBOR tags so defined. Useful techniques for enumerations and sets are presented (without new tags). As the documentation for binary UUIDs (tag 37), MIME entities (tag 36) and regular expressions (tag 35) RFC 7049 left much out, this document provides more comprehensive specifications.

Status of This Memo

Table of Contents

1. Introduction

The Concise Binary Object Representation (CBOR, [RFC7049]) provides for the interchange of structured data without a requirement for a pre-agreed schema. RFC 7049 defines a basic set of data types, as

well as a tagging mechanism that enables extending the set of data types supported via an IANA registry.

Useful tags and techniques have emerged since the publication of [RFC7049]. This document makes use of CBOR's built-in major types to provide for several useful constructs without changing the wire protocol.

The original focus of this work was to add support for object identifiers (OIDs, [X.660]), which many IETF protocols carry. The ASN.1 Basic Encoding Rules (BER, [X.690]) specify the binary encodings of both object identifiers and relative object identifiers. The contents of these encodings can be carried in a CBOR byte string. This document defines two CBOR tags that cover the two kinds of ASN.1 object identifiers encoded in this way. The tags can also be applied to arrays and maps for more articulated identification purposes. It is intended as the reference document for the IANA registration of the tags so defined. To promote the use and usefulness of OIDs in CBOR, a new arc is also proposed.

This document covers several useful techniques that have been or are being developed as implementers are applying CBOR to practical problems. Enumerations have found wide utility in CBOR, despite CBOR's lack of a native enumerated type. A section covers the advantages of choosing built-in types, with additional consideration for using the newly-defined object identifier (OID) and universally unique identifier (UUID) types in enumerations. CBOR also lacks a native set type (in the mathematical sense of an arbitrary unordered collection of items), but has a more powerful alternative in its native map type. A section covers how to adapt the map type to express set and multiset semantics.

Finally, this document covers the semantics of existing tags in [RFC7049] that were somewhat underspecified. "Tag 36 is for MIME messages", but the reference [RFC2045] actually defines a different construct, the MIME entity, that finds expression in a variety of message-oriented Internet protocols. Similarly, "Tag 35 is for regular expressions", but the references to Perl Compatible Regular Expressions (PCRE) and JavaScript syntax (ECMA-262) are not compatible with each other. Two sections cover the subtleties of items tagged with these tags, and so update [RFC7049] without changing the basic CBOR wire protocol. One section enhances UUIDs.

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 RFC 2119 [RFC2119].

The terminology of RFC 7049 applies; in particular the term "byte" is used in its now customary sense as a synonym for "octet".

2. Object Identifiers

The International Object Identifier tree [X.660] is a hierarchically managed space of identifiers, each of which is uniquely represented as a sequence of primary integer values [X.680]. While these sequences can easily be represented in CBOR arrays of unsigned integers, a more compact representation can often be achieved by adopting the widely used representation of object identifiers defined in BER; this representation may also be more amenable to processing by other software making use of object identifiers.

BER represents the sequence of unsigned integers by concatenating self-delimiting [RFC6256] representations of each of the primary integer values in sequence.

ASN.1 distinguishes absolute object identifiers (ASN.1 Type "OBJECT IDENTIFIER"), which begin at a root arc ([X.660] Clause 3.5.21), from relative object identifiers (ASN.1 Type "RELATIVE-OID"), which begin relative to some object identifier known from context ([X.680] Clause 3.8.63). As a special optimization, BER combines the first two integers in an absolute object identifier into one numeric identifier by making use of the property of the hierarchy that the first arc has only three integer values (0, 1, and 2), and the second arcs under 0 and 1 are limited to the integer values between 0 and 39. (The root arc "joint-iso-itu-t(2)" has no such limitations on its second arc.) If X and Y are the first two integers, the single integer actually encoded is computed as:

```
X * 40 + Y
```

The inverse transformation (again making use of the known ranges of X and Y) is applied when decoding the object identifier.

Since the semantics of absolute and relative object identifiers differ, this specification defines two tags:

Tag <<O>> (value TBD): tags a byte string as the [X.690] encoding of an absolute object identifier (simply "object identifier" or "OID").

Tag <<R>> (value TBD): tags a byte string as the [X.690] encoding of a relative object identifier (also "relative OID").

2.1. Requirements on the byte string being tagged

A byte string tagged by <<O>> or <<R>> MUST be a syntactically valid BER representation of an object identifier. Specifically:

- its first byte, and any byte that follows a byte that has the most significant bit unset, MUST NOT be 0x80 (this requirement excludes expressing the primary integer values with anything but the shortest form)

- its last byte MUST NOT have the most significant bit set (this requirement excludes an incomplete final primary integer value)

If either of these invalid conditions are encountered, they MUST be treated as decoding errors. Comparing two OIDs or relative OIDs for equality in a byte-for-byte fashion may not be safe before these checks succeed on at least one of them (this includes the case where one of them is a local constant); a process implementing an exclusion list MUST check for decoding errors first.

[X.680] restricts RELATIVE-OID values to have at least one arc. This specification permits empty relative object identifiers; they may still be excluded by application semantics.

[RFC7049] permits byte strings to be indefinite-length, with chunks divided at arbitrary byte boundaries. This contrasts with text strings, where each chunk in an indefinite-length text string is required be well-formed UTF-8 on its own: splitting the octets of a UTF-8 character encoding between chunks is not allowed.

By analogy to this principle and to Clauses 8.9.1 and 8.20.1 of [X.690], the byte strings carrying the OIDs and relative OIDs are also to be treated as indivisible units: They MUST be encoded in definite-length form; indefinite-length form is treated as an encoding error (and the same considerations as above apply). (An added convenience is that CBOR encodings can be searched through efficiently for specific object identifiers without initiating the decoding process.)

We provide "binary regular expression" forms for implementation convenience. Unlike typical regular expressions that operate on character sequences, the following regular expressions take bytes as their domain, so they can be applied directly to CBOR byte strings.

For byte strings with tag <<O>>:

```
"/^((?:[\x81-\xFF][\x80-\xFF]*)?[\x00-\x7F])+$/"
```

For byte strings with tag <<R>>:

```
"/^((?:[\x81-\xFF][\x80-\xFF]*)?[\x00-\x7F])*$/"
```

Putative CBOR data that fails these tests SHALL be rejected as improperly coded.

Another (possibly more efficient) way to validate the byte strings is to hunt for prohibited patterns.

For byte strings with tag <<O>>:

```
"/^$|(?:^|[\x00-\x7F])\x80|[\x80-\xFF]$/"
```

or with lookbehind:

```
"/^$|^\x80|(?<[\x00-\x7F])\x80|(?<[\x80-\xFF])$/"
```

For byte strings with tag <<R>>:

```
"/(?:^|[\x00-\x7F])\x80|[\x80-\xFF]$/"
```

or with lookbehind:

```
"/^\x80|(?<[\x00-\x7F])\x80|(?<[\x80-\xFF])$/"
```

Putative CBOR data that passes these tests SHALL be rejected as improperly coded.

(It is worth pointing out that these tests, when optimally implemented, ought to be markedly faster than UTF-8 validation.)

3. Examples

In the following examples, we are using tag number 6 for <<O>> and tag number 7 for <<R>>. See Section 17.2.

3.1. Encoding of the SHA-256 OID

ASN.1 Value Notation
{ joint-iso-itu-t(2) country(16) us(840) organization(1) gov(101) csor(3) nistalgorithm(4) hashalgs(2) sha256(1) }

Dotted Decimal Notation (also XML Value Notation)
2.16.840.1.101.3.4.2.1

```
   06                                  # UNIVERSAL TAG 6
      09                               # 9 bytes, primitive
         60 86 48 01 65 03 04 02 01    # X.690 Clause 8.19
   #      |   840  1  |  3  4  2  1     show component encoding
   #   2.16         101
```

Figure 1: SHA-256 OID in BER

```
   C6                                  # 0b110_00110: mt 6, tag 6
      49                               # 0b010_01001: mt 2, 9 bytes
         60 86 48 01 65 03 04 02 01    # X.690 Clause 8.19
```

Figure 2: SHA-256 OID in CBOR

3.2. Encoding of a UUID OID

UUID
8b0d1a20-dcc5-11d9-bda9-0002a5d5c51b

ASN.1 Value Notation
{ joint-iso-itu-t(2) uuid(25)
geomicaGPAS(184830721219540099336690027854602552603) }

Dotted Decimal Notation (also XML Value Notation)
2.25.184830721219540099336690027854602552603

```
   06                                  # UNIVERSAL TAG 6
      14                               # 20 bytes, primitive
         69 82 96 8D 8D 88 9B CC A8 C7 B3 BD D4 C0 80 AA AE D7 8A 1B
   #      |                  184830721219540099336690027854602552603
   #   2.25
```

Figure 3: UUID in an object identifier, in BER

```
   C6                                  # 0b110_00110: mt 6, tag 6
      54                               # 0b010_10100: mt 2, 20 bytes
         69 82 96 8D 8D 88 9B CC A8 C7 B3 BD D4 C0 80 AA AE D7 8A 1B
```

Figure 4: UUID in an object identifier, in CBOR

3.3. Encoding of a MIB Relative OID

Given some OID (e.g., "lowpanMib", assumed to be "1.3.6.1.2.1.226" [RFC7388]), to which the following is added:

ASN.1 Value Notation (not suitable for diagnostic notation)
{ lowpanObjects(1) lowpanStats(1) lowpanOutTransmits(29) }

```
Dotted Decimal Notation (diagnostic notation; see Section 5)
.1.1.29

0D                                # UNIVERSAL TAG 13
   03                             # 3 bytes, primitive
      01 01 1D                    # X.690 Clause 8.20
#      1  1 29                      show component encoding
```

Figure 5: MIB relative object identifier, in BER

```
C7                                # 0b110_00110: mt 6, tag 7
   43                             # 0b010_01001: mt 2 (bstr), 3 bytes
      01 01 1D                    # X.690 Clause 8.20
```

Figure 6: MIB relative object identifier, in CBOR

This relative OID saves seven bytes compared to the full OID encoding.

4. Discussion

Staying close to the way object identifiers are encoded in ASN.1 BER makes back-and-forth translation easy. Object identifiers in IETF protocols are serialized in dotted decimal form or BER form, so there is an advantage in not inventing a third form. Also, expectations of the cost of encoding object identifiers are based on BER; using a different encoding might not be aligned with these expectations. If additional information about an OID is desired, lookup services such as the OID Resolution Service (ORS) [X.672] and the OID Repository [OID-INFO] are available.

This specification allocates two numbers out of the single-byte tag space. This use of code point space is justified by the wide use of object identifiers in data interchange. For most common OIDs in use (namely those whose contents encode to less than 24 bytes), the CBOR encoding will match the efficiency of [X.690]. (This preliminary conclusion is likely to generate some discussion, see Section 17.2.)

5. Diagnostic Notation

Implementers will likely want to see OIDs and relative OIDs in their "natural forms" (as sequences of decimal unsigned integers) for diagnostic purposes. Accordingly, this section defines additional syntactic elements that can be used in conjunction with the diagnostic notation described in Section 6 of [RFC7049].

An object identifier may be written in ASN.1 value notation (with enclosing braces and secondary identifiers, ObjectIdentifierValue of

Clause 32.3 of [X.680]), or in dotted decimal notation with at least three arcs. Both examples are shown in Section 3. The surrounding tag notation is not to be used, because the tag is implied. The ASN.1 value notation for OIDs does not overlap with JSON object notation for CBOR maps, because at least two arcs are required for a valid OID.

A relative object identifier may be written in dotted decimal notation or in ASN.1 value notation, in both cases prefixed with a dot as shown in Section 3.3. The surrounding tag notation is not to be used, because the tag is implied.

The notation in this section may be employed in addition to the basic notation, which would be a tagged binary string.

RFC 7049 diagnostic notation	6(h'2b0601')	7(h'0601')
Dotted decimal notation	1.3.6.1	.6.1
ASN.1 value notation	{1 3 6 1}	.{6 1}

Table 1: Examples for extended diagnostic notation

6. A New Arc for Concise OIDs

Object identifiers in [X.690] form are remarkably compact. Nevertheless, for some applications (and engineers), they are simply not compact enough, at least when compared to certain alternatives such as very small unsigned integers (see Section 10). The shortest object identifier under the IETF's control is 1.3.6.1 (4 bytes), although an assignment directly under that arc has not happened since 1999 [RFC2506], and no assignments directly under that arc have ever been assigned directly to protocol elements. The shortest IETF-controlled, First-Come, First-Served OID arc is 8 bytes by getting a Private Enterprise Number from IANA, an OID for which is assigned under 1.3.6.1.4.1. To promote object identifier usage in CBOR and to make OIDs as competitive as possible, (the authors / the IETF / ISOC) have secured a very short arc "{ x y z }" that only occupies (1, 2, 3) byte(s).

[[NB: Registration procedures under that arc.]]

The history of OIDs suggests that the human mind tends to excessive taxonomy around them. "Excessive taxonomy" means that while classifying purposes are served, the detailed taxonomy comes at the expense of concise encoding to the point that other implementers complain that the OIDs are "too long". OIDs also lose mnemonic

properties when the arcs are so long that implementers cannot keep track of all of the divisions. Unlike assignments in the 1.3.6.1 range, this document suggests that registrants acquire OIDs under this short arc "laterally" rather than hierarchically, in keeping with CBOR's design goal to have concise serializations.

7. Tag Factoring and Tag Stacking with OID Arrays and Maps

A common use of object identifiers in ASN.1 is to identify the kind of data in an open type (Clause 3.8.57 of [X.680]), using information object classes [X.681]. CBOR is schema-neutral, and (although not fully discussed in [RFC7049]) semantic tagging was originally intended to identify items in a global, context-free way (i.e., where a specification would not repurpose a tag with different semantics than its IANA registration). Therefore, using OIDs to identify contextual data in a similar fashion to [X.681] is RECOMMENDED.

7.1. Tag Factoring

<<O>> and <<R>> can tag CBOR arrays and maps. The idea is that the tag is factored out from each individual byte string; the tag is placed in front of the array or map instead. The tags <<O>> and <<R>> are left-distributive.

When the <<O>> or <<R>> tag is applied to an array, it means that the respective tag is imputed to all items in the array. For example, when the array is tagged with <<O>>, every array item that is a binary string is an OID.

When the <<O>> or <<R>> tag is applied to a map, it means that the respective tag is imputed to all keys in the map. The values in the map are not considered specially tagged.

Array and map stacking is permitted. For example, a 3-dimensional array of OIDs can be composed by using a single <<O>> tag, followed by an array of arrays of arrays of binary strings. All such binary strings are considered OIDs.

7.2. Switching OID and Relative OID

If an individual item in a <<O>> or <<R>> tagged array, or an individual key in a <<O>> or <<R>> tagged map, is tagged with the opposite tag (<<R>> or <<O>>) of the array or map itself, that tag cancels and replaces the outer tag for that item. Like tags MUST NOT be used on such individual items; such tagging is a coding error. For example, if <<R>> is the outer tag on an array and <<O>> is the inner tag on a binary string, semantically the inner item is treated as a regular OID, not as a relative OID.

The purpose is to create more compact and flexible identifier spaces, especially when object identifiers are used as enumerated items. Examples:

<<R>> outside, <<O>> inside: An implementation that strives for a compact representation, does not have to emit base OID arcs repeatedly for each item. At the same time, if a private organization or standards body separate from the specification needs to identify something that the specification maintainers disagree with, the separate body does not need to request registration of an identifier under a controlled arc (i.e., the base arc of the relative OIDs).

<<O>> outside, <<R>> inside: A collection of OIDs is supposed to be open to all-comers, but a certain set of OIDs issued under a particular arc is foreseeable for the majority of implementations. For example, an OID protocol slot may identify cryptographic algorithms: anyone can write (and has written) an algorithm with an arbitrary OID. However, the protocol slot designer may wish to privilege certain algorithms (and therefore OIDs) that are well-known in that field of use.

7.3. Tag Stacking

CBOR permits tag stacking (tagging a tagged item), although this technique has not been used much yet. This specification anticipates that OIDs and relative OIDs will be associated with values with uniform semantics. This section provides specific semantics when tags are "stacked", that is, a CBOR item starts with tag <<O>> or <<R>>, followed by one or more arbitrary tags ("subsequent tags"), followed by a map or array.

7.3.1. Map

The overall gist is that the first tag applies to the keys in a map; the subsequent tags apply to the values in a map.

When <<O>> or <<R>> is the first tag in a stack of tags, followed by a map:

- The <<O>> or <<R>> tag indicates that the keys of the map are byte string OIDs, byte string relative OIDs, or tag-factored arrays or maps of the same.

- The subsequent tags uniformly apply to all of the values.

For example, if tag 32 (URL) is the subsequent tag, then all values in the map are treated semantically as if tag 32 is applied to them individually. See Figure 7.

It is possible that individual values can be tagged. Semantically, these tags cumulate with the outer subsequent tags; inner value tags do not cancel or replace the outer tags.

7.3.2. Array

The overall gist is that the first tag applies to the ordered "keys" in the array (even-numbered items, assuming that the index starts at 0); the subsequent tags apply to the ordered "values" in the array (odd-numbered items). This tagging technique creates an ordered associative array. [[NB: Some call this the FORTRAN approach. need to cite]]

When <<O>> or <<R>> is the first tag in a stack of tags, followed by an array:

- The <<O>> or <<R>> tag indicates that alternating items, starting with the first item, are byte string OIDs, byte string relative OIDs, or tag-factored arrays or maps of the same.

- The subsequent tags uniformly apply to the alternating items, starting with the second item.

- The array MUST have an even number of items; an array that has an odd number of items is a coding error.

To create an ordered associative array wherein the values (even elements) are arbitrarily tagged, stack tag 55799, self-describe CBOR (Section 2.4.5 of [RFC7049]), after the <<O>> or <<R>> tag. Tag 55799 imparts no special semantics, so it is an effective placeholder. (This sequence is mainly provided for completeness: it is a more compact alternative to an array of duple-arrays that each contain an OID or relative OID, and an arbitrary value.)

7.4. Diagnostic Notation for OID Arrays and Maps

There are no syntactic changes to diagnostic notation beyond Section 5. Using <<O>> or <<R>> with arrays and maps, however, leads to some sublime results.

When an array or map is tagged, that item is embraced with the usual tag format: "<<O>>(<item>)" or "<<R>>(<item>)". This syntax indicates the presence of the tag on the outer item. Inner items in the array or keys in the map are noted in Section 5 form, but are not

individually tagged on-the-wire when the tag is the same as the outer tag, because like-tagging is a coding error.

An array or map that involves a stack of tags is notated the usual way. For example, the CBOR diagnostic notation of a map of OIDs to URIs is:

```
6(32({0.9.2342.7776.1: "http://example.com/",
      0.9.2342.7776.2: "ftp://ftp.example.com/pub/"}))
```

Figure 7: Map of OIDs to URIs, in CBOR Diagnostic Diagnostic Notation

8. Applications and Examples of OIDs

8.1. GPU Farm

Consider a 3-dimensional OID array, indicating certain operations to perform on a matrix of values in a GPU farm. Default operations are under the OID arc 0.9.2342.7777 (such as .1, .2, .124, etc.); the arc 0.9.2342.7777 itself represents the identity operation. Certain cryptographic operations like SHA-256 hashing (2.16.840.1.101.3.4.2.1) are also permitted. The resulting notation would be:

```
7([[[.1,   .2,   .3],
    [.1,   .2,   .3],
    [.1,   .2,   .3]],
   [[.124, .125, .126],
    [.95,  .96,  .97 ],
    [.11,  .12,  .13 ]],
   [[h'',  .6,   .4.2],
    [.6,   h'',  .4.2],
    [.6,   2.16.840.1.101.3.4.2.1, h'']]])
```

Figure 8: GPU Farm Matrix Operations, in CBOR Diagnostic Notation

```
c7                                      # tag(7)
   83                                   # array(3)
      83                                # array(3)
         83                             # array(3)
            41 01                       # .1 (2)
            41 02                       # .2 (2)
            41 03                       # .3 (2)
         83                             # array(3)
            41 01                       # .1 (2)
            41 02                       # .2 (2)
            41 03                       # .3 (2)
         83                             # array(3)
            41 01                       # .1 (2)
            41 02                       # .2 (2)
            41 03                       # .3 (2)
      83                                # array(3)
         83                             # array(3)
            41 7c                       # .124 (2)
            41 7d                       # .125 (2)
            41 7e                       # .126 (2)
         83                             # array(3)
            41 5f                       # .95 (2)
            41 60                       # .96 (2)
            41 61                       # .97 (2)
         83                             # array(3)
            41 0b                       # .11 (2)
            41 0c                       # .12 (2)
            41 0d                       # .13 (2)
      83                                # array(3)
         83                             # array(3)
            40                          # (empty) (1)
            41 06                       # .6 (2)
            42 0402                     # .4.2 (3)
         83                             # array(3)
            41 06                       # .6 (2)
            40                          # (empty) (1)
            42 0402                     # .4.2 (3)
         83                             # array(3)
            41 06                       # .6 (2)
            c6 49 608648016503040201 # 2.16.840.1.101.3.4.2.1 (10)
            40                          # (empty) (1)
```

Figure 9: GPU Farm Matrix Operations, in CBOR (76 bytes)

8.2. X.500 Distinguished Name

Consider the X.500 distinguished name:

Attribute Types	Attribute Values
c (2.5.4.6)	US
l (2.5.4.7)	Los Angeles
s (2.5.4.8)	CA
postalCode (2.5.4.17)	90013
street (2.5.4.9)	532 S Olive St
businessCategory (2.5.4.15)	Public Park
buildingName (0.9.2342.19200300.100.1.48)	Pershing Square

Table 2: Example X.500 Distinguished Name

Table 2 has four RDNs. The country and street RDNs are single-valued. The second and fourth RDNs are multi-valued.

The equivalent representations in CBOR diagnostic notation and CBOR are:

```
6([{ 2.5.4.6: "US" },
   { 2.5.4.7: "Los Angeles", 2.5.4.8: "CA", 2.5.4.17: "90013" },
   { 2.5.4.9: "532 S Olive St" },
   { 2.5.4.15: "Public Park",
     0.9.2342.19200300.100.1.48: "Pershing Square" }])
```

Figure 10: Distinguished Name, in CBOR Diagnostic Notation

```
6([{ h'550406': "US" },
   { h'550407': "Los Angeles", h'550408': "CA", h'550411': "90013" },
   { h'550409': "532 S Olive St" },
   { h'55040f': "Public Park",
     h'0992268993f22c640130': "Pershing Square" }])
```

Figure 11: Distinguished Name, in CBOR Diagnostic Notation (RFC 7049 only)

```
c6                                      # tag(6)
   84                                   # array(4)
      a1                                # map(1)
         43 550406                      # 2.5.4.6 (4)
         62                             # text(2)
            5553                        # "US"
      a3                                # map(3)
         43 550407                      # 2.5.4.7 (4)
         6b                             # text(11)
            4c6f7320416e67656c6573      # "Los Angeles"
         43 550408                      # 2.5.4.8 (4)
         62                             # text(2)
            4341                        # "CA"
         43 550411                      # 2.5.4.17 (4)
         65                             # text(5)
            3930303133                  # "90013"
      a1                                # map(1)
         43 550409                      # 2.5.4.9 (4)
         6e                             # text(14)
            3533322053204f6c697665205374   # "532 S Olive St"
      a2                                # map(2)
         43 55040f                      # 2.5.4.15 (4)
         6b                             # text(11)
            5075626c6963205061726b      # "Public Park"
         4a 0992268993f22c640130    # 0.9.2342.19200300.100.1.48 (11)
         6f                             # text(15)
            5065727368696e6720537175617265 # "Pershing Square"
```

Figure 12: Distinguished Name, in CBOR (108 bytes)

(This example encoding assumes that all attribute values are UTF-8 strings, or can be represented as UTF-8 strings with no loss of information.)

For reference, the [RFC4514] LDAP string encoding of such data would be:

```
buildingName=Pershing Square+businessCategory=Public Park,
street=532 S Olive St,l=Los Angeles+postalCode=90013+st=CA,c=US
```

Figure 13: Distinguished Name, in LDAP String Encoding (121 bytes)

9. Universally Unique Identifiers in CBOR

This section provides guidance on the Universally Unique Identifier (UUID) type, which was introduced into CBOR with tag <<U>> (currently tag 37, reassignment to be discussed in view of this section). A UUID [RFC4122] is 128 bits long and requires no central registration

process. UUIDs were originally used in the Apollo Network Computing System and later in the Open Software Foundation's (OSF) Distributed Computing Environment (DCE), for Remote Procedure Calls (RPC) [DCE-RPC].

As a tagged binary string identifier type in CBOR, the UUID type shares several characteristics with OID types. The main differences are that a UUID is always 16 bytes (anything less or more is a coding error), there is no central assignment process, and every 128-bit combination is valid. ([RFC4122] calls out the nil UUID, which is special but perfectly valid.) Optional registries have cropped up over the years; one such registry is [OID-INFO]. Users who use UUIDs in CBOR are strongly encouraged to document their UUIDs in such registries.

To provide parity with OIDs, UUIDs MUST be encoded in definite-length form (see Section 2). Consequently, individual UUIDs can be easily searched for by looking for "d8 25" (major type 6, tag 37), "50" (major type 2, additional information 16), and 16 bytes. Therefore, a directly encoded UUID in CBOR occupies 19 bytes. In contrast, stuffing a UUID in an OID in CBOR requires 22 bytes (see Figure 4); conversion between OID-UUID form and binary or string UUID forms requires bit-shifting (but mercifcully not base-shifting, see Section 18.1). An example based on Figure 4 is below:

```
D8 25                           # tag(37)
   54                           # 0b010_10000: mt 2, 16 bytes
      8B 0D 1A 20 DC C5 11 D9 BD A9 00 02 A5 D5 C5 1B
```

Figure 14: Binary UUID in CBOR

9.1. Diagnostic Notation

Implementers will likely want to see UUIDs in their "natural forms" for diagnostic purposes. Accordingly, this section defines additional syntactic elements that can be used in conjunction with the diagnostic notation described in Section 6 of [RFC7049].

A universally unique identifier may be written in "string representation" as that term is defined in [RFC4122]. An example of such a string is "8b0d1a20-dcc5-11d9-bda9-0002a5d5c51b" (see Figure 4 and Figure 14). Lowercase is the preferred form. (TBD: permit, require, or prohibit curly brace form?)

The notation in this section may be employed in addition to the basic notation, which would be a tagged binary string.

9.2. Tag Factoring and Tag Stacking

Tag Factoring and Tag Stacking are hereby permitted with the UUID type, with the same semantics as Section 7.

10. Enumerations in CBOR

This section provides a roadmap to using enumerated items in CBOR, including design considerations for choosing between OIDs, UUIDs, integers, and UTF-8 strings.

CBOR does not have an ENUMERATED type like ASN.1 to identify named values in a protocol element with three or more states (Clause 20 and Clause G.2.3 of [X.680]). ASN.1 ENUMERATED turns out to be superfluous because ASN.1 INTEGER values can get named (and have historically been used for finite, multistate variables, such as version numbers), while ASN.1 ENUMERATED types can be defined to be extensible with the ellipsis lexical item. Practically, the named integers are not serialized in the binary encodings anyway; they merely serve as a semantic hints for designers and debuggers.

CBOR expects that protocol designers will use one of the basic major types for multistate variables, assigning semantics to particular values using higher-level schemas. The obvious choices for the basic types are integers (particularly unsigned integers) and UTF-8 strings. However, these major types are not without drawbacks.

Integers are compact for small values, but have a flat namespace so there are mis-assignment and collision risks that can only be mitigated with protocol-specific registries. Arrays of integers are possible, but arrays require more processing logic for equality comparisons, and the JSON conversion is not intuitive when the enumerated value serves as a key in a map.

UTF-8 strings are less compact when the strings are supposed to resemble their semantics, and there are normalization issues if the strings contain characters beyond the ASCII range. UTF-8 strings also comprise a flat namespace like integers unless the higher-level schema employs delimiters, which makes the string even larger. If conciseness is a design goal, other perceived advantages of a string as an identifier are pretty much blown out the moment one has to tack "https://" onto the front.

This section provides novel alternatives in OIDs and UUIDs. It compares and contrasts these binary types to other enumerants, namely integers and text (UTF-8) strings.

10.1. Factors Favoring OID Enumerations

A protocol designer might choose OIDs or relative OIDs for an enumerated item in view of the following observations:

1. OIDs and relative OIDs are quite compact: a single-arc relative OID encoded according to this specification occupies just two bytes for primary integer values 0-127 (excluding the semantic tag <<R>>), and three bytes for primary integer values 128-16383. (In contrast, an unsigned integer requires one byte for 0-23, two bytes for 24-255, and three bytes for 256-65535.)

2. OIDs and relative OIDs (with base) are persistent and globally unambiguous.

3. OIDs and relative OIDs have built-in semantics for designers and debuggers. Specifically, the advent of universal OID repositories such as [OID-INFO] makes it easy for a designer or debugger to pull up useful information about the object of interest (Clause 3.5.10 of [X.660]). This useful information (for humans) does not have to bleed into the encoded representation (for machines).

4. OIDs and relative OIDs are always compared for exact equality: no need to deal with case folding, case sensitivity, or other normalization issues. ("Overlong" encodings are PROHIBITED; therefore overlong encodings MUST be treated as coding errors.)

5. OIDs and relative OIDs have a built-in hierarchy, so if implementers want to extend an enumeration without assigning new values "horizontally", they have the option of assigning new values "vertically", possibly with more or less stringent assignment rules.

6. Because OIDs and relative OIDs (with base) are part of the so-called International Object Identifier tree [X.660], any other protocol specification can reuse the enumeration if the designers find it useful.

7. OIDs and relative OIDs have natural JSON representations in the dotted decimal notations prescribed in Section 5. OIDs and relative OIDs can be distinguished from each other by the presence or absence of the leading dot ".". As the resulting JSON string is entirely numeric in the ASCII range, case and normalization are irrelevant to the comparison. (An object identifier also has a semantic string representation in the form of an OID-IRI [X.680], for those who really want that type of thing.)

8. OIDs and relative OIDs are human language-neutral. A protocol designer working in US-English might name an enumerated value "sig" for "signature", but "sig" could also stand for "significand", "signal", or "special interest group". In Swedish and Norwegian, "sig" is a pronoun that means "himself, herself, itself, one, them", etc.--an entirely different meaning.

10.2. Factors Favoring UUID Enumerations

A Universally Unique Identifier (UUID) is a 128-bit identifier that is unique across both space and time with a very high degree of probability; one intent is to identify "very persistent objects across a network", such as remote procedure call interfaces [DCE-RPC].

A protocol designer might choose UUIDs for an enumerated item in view of the following observations:

1. UUIDs are always 16 bytes. This means that while they are not particularly short, they also cannot be overly long. Space is constant and predictable. (As great as OIDs are, an OID that exceeds 17 bytes is simply excessive compared to a randomly-assigned UUID.)

2. Any 128-bit combination is a valid UUID. The other types in this section have to be validated, even integers (e.g., to avoid overflow and out-of-range conditions).

3. There is no registration authority that serves as a roadblock, and (for all practical purposes) no semantic or aesthetic values are implied by lower bit combinations.

4. Many platforms can compare UUIDs (128-bit values) in one atomic operation. The comparison can be done without regard to endianness, provided that the endianness is the same between two UUIDs in memory. (On the wire, a CBOR UUID is big-endian.) For this reason, UUIDs may be faster than (naive) integer enumerations.

5. UUIDs have natural JSON representations in the string representations prescribed by [RFC4122]. The resulting JSON strings are entirely in the ASCII range and occupy exactly 36 characters; however, normalization (to lowercase) is a complicating factor.

6. UUIDs are human language-neutral. (However, unlike OIDs, UUIDs are too long to be described as mnemonic in any practical sense.)

10.3. Factors Favoring Integer Enumerations

A protocol designer might choose integers for an enumerated item in view of the following observations:

1. The CBOR encoding of unsigned integers 0-23 is the most compact, occupying exactly one byte (excluding any semantic tags).

2. A protocol designer may wish to prohibit extensibility as a matter of course. Integers comprise a single flat namespace: there is no hierarchy.

3. If greater range is desired while sticking to one byte, a protocol designer may double the range of possible values by allowing negative integers. However, enumerating values using negative integers may have unintended side-effects, because some programming environments (e.g., C/C++) make implementation-defined assumptions about the number of bits needed for an enumerated type.

10.4. Factors Favoring UTF-8 String Enumerations

A protocol designer might choose UTF-8 strings for an enumerated item in view of the following observations:

1. A specification can practically limit the content of UTF-8 strings to the ASCII range (or narrower), mitigating some normalization problems.

2. UTF-8 strings are easier to read on-the-wire for humans.

3. UTF-8 strings can contain arbitrary textual identifiers, which can be hierarchical, e.g., URIs.

10.5. OID Enumeration Example

An enumerated item indicates the revision level of a data format. Revision levels are issued by year, such as 2011, 2012, etc. However, in the year 2013, two revisions were issued: the first one and an important update in June that needs to be distinguished. The revision levels are assigned to some OID arc:

"{2 25 6464646464 revs(4)}"

In this arc, the following sub-arcs are assigned:

Sub-Arc
{v2011(1)}
{v2012(2)}
{v2013(3)}
{v2013(3) june(6)}
{v2014(4)}
{v2015(5)}

Table 3: Example Sub-Arcs

In CBOR, the enumeration is encoded as a relative OID. The schema specifies the base OID arc, which is omitted:

```
c7         # tag(7)
   41 03   # .3

c7         # tag(7)
   42 0306 # .3.6
```

Figure 15: Enumerated Items in CBOR

```
.3
.{v2013(3) june(6)}
```

Figure 16: Enumerated Items in CBOR Diagnostic Notation

```
".3"
".3.6"
```

Figure 17: Enumerated Items in JSON (possibility 1)

```
"v2013"
"v2013/june"
```

Figure 18: Enumerated Items in JSON (possibility 2)

11. Binary Internet Messages and MIME Entities

Section 2.4.4.3 of [RFC7049] assigns tag 36 to "MIME messages (including all headers)" [RFC2045], and prescribes UTF-8 strings, without further elaboration. Actually MIME encircles several different formats, and is not limited to UTF-8 strings. This section updates tag 36.

11.1. CBOR Byte String and Binary MIME

Tag 36 is to be used with byte strings. When the tagged item is a byte string, any octet can be used in the content. Arbitrary octets are supported by [RFC2045] and can be supported in protocols such as SMTP using BINARYMIME [RFC3030].

A conforming implementation that purports to process tag 36-tagged items, MUST accept byte strings as well as UTF-8 strings. Byte strings, rather than UTF-8 strings, SHOULD be considered the default. (While binary Content-Transfer-Encoding is not particularly common as of this writing, 8-bit encoding is, and it is foreseeable that many 8-bit encoded messages will still have charsets other than UTF-8.)

11.2. Internet Messages, MIME Messages, and MIME Entities

Definitions: "MIME message" is not explicitly defined in [RFC2045], but a careful read suggests that a MIME message is: "either a (complete or "top-level") RFC 822 message being transferred on a network, or a message encapsulated in a body of type "message/rfc822" or "message/partial"," that also contains MIME header fields, namely, MIME-Version field, which MUST be present (Section 4 of [RFC2045]. Other MIME header fields such as Content-Type and Content-Transfer-Encoding are assumed to be their [RFC2045] default values, if not present in the data.

When the contents have a From field (a type of "originator address field") and a Date field (the lone "origination date field") (Section 3.6 of [RFC5322]), the item is concluded to have a Content-Type of message/rfc822 or message/global, as appropriate, except as otherwise specified in this section.

(TBD: Do we need a separate tag for a MIME entity?) (Alternate proposal: When the tagged data does not include a MIME-Version field or other fields required by RFC822 (5322) (e.g., no From field), it is presumed to be a MIME entity, rather than a MIME message. Therefore, it has no top-level content-type: instead it is simply a "MIME entity", consisting of one element, whose Content-Type is the content of the Content-Type header field, if present, or the [RFC2045] default of "text/plain; charset=us-ascii", if absent. Content-Transfer-Encoding SHALL be assumed to be 8bit when the CBOR item is a UTF-8 string, and SHALL be assumed to be binary when the CBOR item is a byte string. (Or should all be considered CTE: binary?) And, when the tagged data has RFC822 required fields but no MIME-Version, shall we assume it’s a MIME entity, or shall we assume it’s an Internet message that does not conform to MIME?)

Content that has no headers whatsoever is valid, and implementations that process tag 36 MUST permit this case: in such a case, the data starts with CRLF CRLF, followed by the body. In such a case, the content is assumed to be a MIME entity of Content-Type "text/plain; charset=us-ascii", and not an RFC822 (RFC5322) Internet message. (TBD: Confirm.)

11.3. Netnews, HTTP, and SIP Messages

Other message types that are MIME-related are message/news, message/http, and message/sip.

[RFC5537] specifies that message/news is deprecated (marked as obsolete) and that message/rfc822 SHOULD be used in its place; presumably this also extends to message/global over time. Netnews Article Format [RFC5536] is a strict subset of Internet Message Format; it can be detected by the presence of the six mandatory header fields: Date, From, Message-ID, Newsgroups, Path, and Subject. (Newsgroups and Path fields are specific to Netnews.)

message/http [RFC7230] is the media type for HTTP requests and responses. It can be detected by analyzing the first line of the body, which is an HTTP Start Line (Section 3.1 of [RFC7230]): it does not conform to the syntax of an Internet Message Format header field. The optional parameter "msgtype" can be inferred from the Start Line. Implementers need to be aware that the default character encoding for message/http is ISO-8859-1, not UTF-8. Therefore, implementations SHOULD NOT encode HTTP messages with CBOR UTF-8 strings.

Similarly, message/sip [RFC3261] is the media type of SIP request and response messages. It can be detected by analyzing the first line of the body, which is a SIP start-line (Section 7.1 of [RFC3261]): it does not conform to the syntax of an Internet Message Format header field. The optional parameter can be inferred from the start-line.

11.4. Other Messages

The CBOR binary or UTF-8 string MAY contain other types of messages. An implementation MAY send such a message as a MIME entity with the Content-Type field appropriately set, or alternatively, MAY send the message at the top-level directly. However, if a purported message type is ambiguous with a message/rfc822 (or message/global) message, a receiver SHALL treat the message as message/rfc822 (or message/global). If a purported message type is ambiguous with a MIME entity (and unambiguously not message/rfc822 or message/global), a receiver SHALL treat the message as a MIME entity.

12. Applications and Examples of Messages and Entities

Tag 36 is the RECOMMENDED way to convey data with MIME-related metadata, including messages (which may or may not actually be MIME-enabled) and MIME entities.

Example 1: A legacy RFC822 message is encoded as a UTF-8 string or byte string with tag 36. The contents have From, To, Date, and Subject header fields, two CRLFs, and a single line "Hello World!", terminated with a CRLF.

Example 2a: A [RFC5280] certificate is encoded as a byte string with tag 36. The contents are comprised of "Content-Type: application/pkix-cert", two CRLFs, and the DER encoding of the certificate. (The "Content-Transfer-Encoding: binary" header is not necessary.)

Example 2b: A [RFC5280] certificate is encoded as a UTF-8 string or byte string with tag 36. The contents are comprised of "Content-Type: application/pkix-cert", a CRLF, "Content-Transfer-Encoding: base64", two CRLFs, and the base64 encoding of the DER encoding of the certificate, conforming to Section 6.8 of [RFC2045]. In particular, base64 lines are limited to 76 characters, separated by CRLF, and the final line is supposed to end with CRLF. Needless to say, this is not nearly as efficient as Example 2a.

13. X.690 Series Tags

[[NB: Carsten probably won’t like this. Plan on removing this section. It is mainly provided to contrast with Section 10.]]

It is foreseeable that CBOR applications will need to send and receive ASN.1 data, for example, for legacy or security applications. While a native representation in CBOR is preferred, preserving the data in an ASN.1 encoding may be necessary, for example, to preserve cryptographic verification. A tag <<X>> is allocated for this purpose.

When the tagged item is a byte string, the byte string contents are encoded according to [X.690], i.e., BER, CER, or DER. CBOR implementations are not required to validate conformance of the contained data to [X.690].

When the tagged item is an array with 3 items:

1. The first item SHALL be an OID (with tag <<O>> omitted; it SHALL NOT be a relative OID), indicating the ASN.1 module containing the type of the PDU. [[NB: this is a good example of a non-trivial structure in which an element is well-defined to be an

OID, which has a tag. Is the CBOR philosophy to tag the item, or omit the tag on the item, when the item's semantics are already fixed by the outer tag? Similar situations can apply to tag 32 (URI), etc.]]

2. The second item SHALL be a UTF-8 string indicating the ASN.1 value's _type reference name_ (Clause 3.8.88 of [X.680]) conforming to the "typereference" production (Clause 12.2 of [X.680]).

3. The third item SHALL be a byte string, whose contents are encoded per the prior paragraph.

(TBD: Use of tagged UTF-8 string is reserved for ASN.1 textual formats such as XER and ASN.1 value notation? Probably not necessary. Just omit.)

Implementation note: DER-encoded items are always definite-length, so there is very little reason to use CBOR byte string indefinite encoding when encoding such DER-encoded items.

Example: A [RFC5280] certificate can be encoded:

1. as a byte string with tag <<X>>, or

2. as an array with tag <<X>>, with three elements:

   (1) a byte string "h'2B 06 01 05 05 07 00 12'", which is the BER encoding of 1.3.6.1.5.5.7.0.18,

   (2) a UTF-8 string "Certificate", and

   (3) a byte string containing the DER encoding of the certificate.

14. Regular Expression Clarification

(TODO: better specify conformance to actual regular expression standards with tag 35. PCRE and JavaScript/ECMAScript regular expressions are very different; [RFC7049] is not specific enough about this.)

15. Set and Multiset Technique

CBOR has no native type for a set, which is an arbitrary unordered collection of items. The following technique is RECOMMENDED to express set and multiset semantics concisely in native CBOR data.

In computer science, a _set_ is a collection of distinct items; there is no ordering to the items. Thus, implementations can optimize set storage in many ways that are not available with ordered elements in arrays. Sets can be stored in hashtables, bit fields, trees, or other abstract data types.

In computer science, a _multiset_ allows multiple instances of a set's elements. Put another way, each distinct item has a cardinality property indicating the number of these items in the multiset.

To store items in a set or multiset, it is RECOMMENDED to store the CBOR items as keys in a map; the values SHALL all be positive integers (major type 0, value/additional information greater than or equal to 1). In the special case of a set, the values SHALL be the integer 1. This technique has no special tag associated with it. As with arrays that schemas classify as "records" (i.e., arrays with positionally defined elements), schemas are likewise free to classify maps as sets in particular instances.

16. Fruits Basket Example

Consider a basket of fruits. The basket can contain any number of fruits; each fruit of the same species is considered identical. This basket has two apples, four bananas, six pears, and one pineapple:

```
{"\u{1F34E}": 2, "\u{1F34C}": 4,
 "\u{1F350}": 6, "\u{1F34D}": 1}
```

Figure 19: Fruits Basket in CBOR Diagnostic Notation

```
A4                      # map(4)
   64                   # text(4)
      f09f8d8e          # "\u{1F34E}"
   02                   # unsigned(2)
   64                   # text(4)
      f09f8d8c          # "\u{1F34C}"
   04                   # unsigned(4)
   64                   # text(4)
      f09f8d90          # "\u{1F350}"
   06                   # unsigned(6)
   64                   # text(4)
      f09f8d8d          # "\u{1F34D}"
   01                   # unsigned(1)
```

Figure 20: Fruits Basket in CBOR (33 bytes)

[[TODO: Consider a Merkle Tree example: set of sets of sets of sets of things. ???]]

17. IANA Considerations

(This section to be edited by the RFC editor.)

17.1. CBOR Tags

IANA is requested to assign the CBOR tags in Table 4, with the present document as the specification reference.

Tag	Data Item	Semantics
6<<TBD>>	multiple	object identifier (BER encoding)
7<<TBD>>	multiple	relative object identifier (BER encoding)

Table 4: Values for New Tags

17.2. Discussion

(This subsection to be removed by the RFC editor.)

The space for single-byte tags in CBOR (0..23) is severely limited. It is not clear that the benefits of encoding OIDs/relative OIDs with one less byte per instance outweigh the consumption of two values in this code point space.

Procedurally, this space is also reserved for standards action.

An alternative would be to go for the specification required space, e.g. tag number 40 for <<O>> and tag number 41 for <<R>>. As an example this would change Figure 2 into:

```
d8 28                           # tag(40)
   49                           # bytes(9)
      60 86 48 01 65 03 04 02 01 #
```

Figure 21: SHA-256 OID in cbor (using specification required tag)

17.3. Pre-Existing Tags

(TODO: complete.) IANA is requested to modify the registrations for the following CBOR tags:

Tag	Data Item	Semantics
35	<<TBD>>	regular expression <<TBD>>
36	multiple	message or MIME entity
37	multiple	binary UUID

Table 5: Values for Existing Tags

17.4. New Tags

(TODO: complete.)

18. Security Considerations

The security considerations of RFC 7049 apply.

The encodings in Clauses 8.19 and 8.20 of [X.690] are extremely compact and unambiguous, but MUST be followed precisely to avoid security pitfalls. In particular, the requirements set out in Section 2.1 of this document need to be followed; otherwise, an attacker may be able to subvert a checking process by submitting alternative representations that are later taken as the original (or even something else entirely) by another decoder supposed to be protected by the checking process.

OIDs and relative OIDs can always be treated as opaque byte strings. Actually understanding the structure that was used for generating them is not necessary, and, except for checking the structure requirements, it is strongly NOT RECOMMENDED to perform any processing of this kind (e.g., converting into dotted notation and back) unless absolutely necessary. If the OIDs are translated into other representations, the usual security considerations for non-trivial representation conversions apply; the primary integer values are unlimited in range (cf. Figure 4).

18.1. Conversions Between BER and Dotted Decimal Notation

[PKILCAKE] uncovers exploit vectors for the illegal values above, as well as for cases in which conversion to or from the dotted decimal notation goes awry. Neither [X.660] nor [X.680] place an upper bound on the range of unsigned integer values for an arc; the integers are arbitrarily valued. An implementation SHOULD NOT attempt to convert each component using a fixed-size accumulator, as an attacker will certainly be able to cause the accumulator to overflow. Compact and efficient techniques for such conversions, such as the double dabble

algorithm [DOUBLEDABBLE] are well-known in the art; their application to this field is left as an exercise to the reader.

19. References

19.1. Normative References

[RFC2045] Freed, N. and N. Borenstein, "Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies", RFC 2045, DOI 10.17487/RFC2045, November 1996, <http://www.rfc-editor.org/info/rfc2045>.

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

[RFC3261] Rosenberg, J., Schulzrinne, H., Camarillo, G., Johnston, A., Peterson, J., Sparks, R., Handley, M., and E. Schooler, "SIP: Session Initiation Protocol", RFC 3261, DOI 10.17487/RFC3261, June 2002, <http://www.rfc-editor.org/info/rfc3261>.

[RFC4122] Leach, P., Mealling, M., and R. Salz, "A Universally Unique IDentifier (UUID) URN Namespace", RFC 4122, DOI 10.17487/RFC4122, July 2005, <http://www.rfc-editor.org/info/rfc4122>.

[RFC5322] Resnick, P., Ed., "Internet Message Format", RFC 5322, DOI 10.17487/RFC5322, October 2008, <http://www.rfc-editor.org/info/rfc5322>.

[RFC5536] Murchison, K., Ed., Lindsey, C., and D. Kohn, "Netnews Article Format", RFC 5536, DOI 10.17487/RFC5536, November 2009, <http://www.rfc-editor.org/info/rfc5536>.

[RFC5537] Allbery, R., Ed. and C. Lindsey, "Netnews Architecture and Protocols", RFC 5537, DOI 10.17487/RFC5537, November 2009, <http://www.rfc-editor.org/info/rfc5537>.

[RFC7049] Bormann, C. and P. Hoffman, "Concise Binary Object Representation (CBOR)", RFC 7049, DOI 10.17487/RFC7049, October 2013, <http://www.rfc-editor.org/info/rfc7049>.

[RFC7230] Fielding, R., Ed. and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing", RFC 7230, DOI 10.17487/RFC7230, June 2014, <http://www.rfc-editor.org/info/rfc7230>.

[X.660] International Telecommunications Union, "Information technology -- Procedures for the operation of object identifier registration authorities: General procedures and top arcs of the international object identifier tree", ITU-T Recommendation X.660, July 2011.

[X.680] International Telecommunications Union, "Information technology -- Abstract Syntax Notation One (ASN.1): Specification of basic notation", ITU-T Recommendation X.680, August 2015.

[X.690] International Telecommunications Union, "Information technology -- ASN.1 encoding rules: Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER) and Distinguished Encoding Rules (DER)", ITU-T Recommendation X.690, August 2015.

19.2. Informative References

[DCE-RPC] Open Group CAE, "DCE: Remote Procedure Call", Specification C309, ISBN 1-85912-041-5, August 1994.

[DOUBLEDABBLE] Gao, S., Al-Khalili, D., and N. Chabini, "An improved BCD adder using 6-LUT FPGAs", IEEE 10th International New Circuits and Systems Conference (NEWCAS 2012), pp. 13-16, DOI: 10.1109/NEWCAS.2012.6328944, June 2012.

[OID-INFO] Orange SA, "OID Repository", 2016, <http://www.oid-info.com/>.

[PKILCAKE] Kaminsky, D., Patterson, M., and L. Sassaman, "PKI Layer Cake: New Collision Attacks Against the Global X.509 Infrastructure", FC 2010, Lecture Notes in Computer Science 6052 289-303, DOI: 10.1007/978-3-642-14577-3_22, January 2010, <http://dl.acm.org/citation.cfm?id=2163593>.

[RFC2506] Holtman, K., Mutz, A., and T. Hardie, "Media Feature Tag Registration Procedure", BCP 31, RFC 2506, DOI 10.17487/RFC2506, March 1999, <http://www.rfc-editor.org/info/rfc2506>.

[RFC3030] Vaudreuil, G., "SMTP Service Extensions for Transmission of Large and Binary MIME Messages", RFC 3030, DOI 10.17487/RFC3030, December 2000, <http://www.rfc-editor.org/info/rfc3030>.

[RFC4514] Zeilenga, K., Ed., "Lightweight Directory Access Protocol (LDAP): String Representation of Distinguished Names", RFC 4514, DOI 10.17487/RFC4514, June 2006, <http://www.rfc-editor.org/info/rfc4514>.

[RFC5280] Cooper, D., Santesson, S., Farrell, S., Boeyen, S., Housley, R., and W. Polk, "Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile", RFC 5280, DOI 10.17487/RFC5280, May 2008, <http://www.rfc-editor.org/info/rfc5280>.

[RFC6256] Eddy, W. and E. Davies, "Using Self-Delimiting Numeric Values in Protocols", RFC 6256, DOI 10.17487/RFC6256, May 2011, <http://www.rfc-editor.org/info/rfc6256>.

[RFC7388] Schoenwaelder, J., Sehgal, A., Tsou, T., and C. Zhou, "Definition of Managed Objects for IPv6 over Low-Power Wireless Personal Area Networks (6LoWPANs)", RFC 7388, DOI 10.17487/RFC7388, October 2014, <http://www.rfc-editor.org/info/rfc7388>.

[X.672] International Telecommunications Union, "Information technology -- Open systems interconnection -- Object identifier resolution system", ITU-T Recommendation X.672, August 2010.

[X.681] International Telecommunications Union, "Information technology -- Abstract Syntax Notation One (ASN.1): Information object specification", ITU-T Recommendation X.681, August 2015.

Appendix A. Changes from -05 to -06

Refreshed the draft to the current date ("keep-alive").

Appendix B. Changes from -04 to -05

Discussed UUID usage in CBOR, and incorporated fixes proposed by Olivier Dubuisson, including fixes regarding OID nomenclature.

Appendix C. Changes from -03 to -04

Changes occurred based on limited feedback, mainly centered around the abstract and introduction, rather than substantive technical changes. These changes include:

- Changed the title so that it is about tags and techniques.

- Rewrote the abstract to describe the content more accurately, and to point out that no changes to the wire protocol are being proposed.
- Removed "ASN.1" from "object identifiers", as OIDs are independent of ASN.1.
- Rewrote the introduction to be more about the present text.
- Proposed a concise OID arc.
- Provided binary regular expression forms for OID validation.
- Updated IANA registration tables.

Appendix D. Changes from -02 to -03

Many significant changes occurred in this version. These changes include:

- Expanded the draft scope to be a comprehensive CBOR update.
- Added OID-related sections: OID Enumerations, OID Maps and Arrays, and Applications and Examples of OIDs.
- Added Tag 36 update (binary MIME, better definitions).
- Added stub/experimental sections for X.690 Series Tags (tag <<X>>) and Regular Expressions (tag 35).
- Added technique for representing sets and multisets.
- Added references and fixed typos.

Authors' Addresses

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

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

Sean Leonard
Penango, Inc.
5900 Wilshire Boulevard
21st Floor
Los Angeles, CA 90036
USA

Email: dev+ietf@seantek.com
URI: http://www.penango.com/

Network Working Group
Internet-Draft
Intended status: Informational
Expires: September 13, 2017

C. Bormann
Universitaet Bremen TZI
B. Gamari
Well-Typed
H. Birkholz
Fraunhofer SIT

March 12, 2017

Concise Binary Object Representation (CBOR) Tags for Time, Duration, and Period

draft-bormann-cbor-time-tag-00

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.

In CBOR, one point of extensibility is the definition of CBOR tags. RFC 7049 defines two tags for time: CBOR tag 0 (RFC3339 time) and tag 1 (Posix time [TIME_T], int or float). Since then, additional requirements have become known. The present document defines a CBOR tag for time that allows a more elaborate representation of time, as well as CBOR tags for duration and time period. It is intended as the reference document for the IANA registration of the CBOR tags defined.

Table of Contents

1. Introduction

The Concise Binary Object Representation (CBOR, [RFC7049]) provides for the interchange of structured data without a requirement for a pre-agreed schema. RFC 7049 defines a basic set of data types, as

well as a tagging mechanism that enables extending the set of data types supported via an IANA registry.

(TBD: Expand on text from abstract here.)

1.1. Terminology

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119 [RFC2119].

The term "byte" is used in its now customary sense as a synonym for "octet". Where bit arithmetic is explained, this document uses the notation familiar from the programming language C (including C++14's 0bnnn binary literals), except that the operator "**" stands for exponentiation.

1.2. Background

Additional information about the complexities of time representation can be found in [TIME]. This specification uses a number of terms that should be familiar to connoisseurs of precise time; references for these may need to be added.

2. Objectives

For the time tag, the present specification addresses the following objectives that go beyond the original tags 0 and 1:

- Indication of time scale. Tags 0 and 1 are for UTC; however, some interchanges are better performed on TAI. Other time scales may be registered once they become relevant (e.g., one of the proposed successors to UTC that might no longer use leap seconds, or a scale based on smeared leap seconds).

- Additional resolution for epoch-based time (as in tag 1). CBOR tag 1 only provides for integer and up to binary64 floating point representation of times, limiting resolution to approximately microseconds at the time of writing (and progressively becoming worse over time).

- Direct representation of natural platform time formats. Some platforms use epoch-based time formats that require some computation to convert them into the representations allowed by tag 1; these computations can also lose precision and cause ambiguities. (TBD: The present specification does not take a position on whether tag 1 can be "fixed" to include, e.g., Decimal

or BigFloat representations. It does define how to use these with the extended time format.)

- Additional indication of intents about the interpretation of the time given, in particular for future times. Intents might include information about time zones, daylight savings times, etc. (TBD: This is not yet a well-developed part of the spec; there needs to be some effort to avoid the kitchen sink.)

The objectives for the duration and period tags are similar.

3. Time Format

An extended time is indicated by CBOR tag TBDET, which tags a map data item (CBOR major type 5). The map may contain integer (major types 0 and 1) or text string (major type 3) keys, with the value type determined by each specific key. Implementations MUST ignore key/value types they do not understand. (Discussion: Do we need "critical" keys?)

The map must contain exactly one unsigned integer key, which specifies the "base time", and may also contain one or more negative integer or text-string keys, which may encode supplementary information such as,

- a higher precision time offset to be added to the base time,
- a reference time scale,
- information about clock source and precision, accuracy, and resolution
- intent information such as timezone and daylight savings time, and/or possibly positioning coordinates, to express information that would indicate a local time.

While this document does not define supplementary text keys, a number of unsigned and negative-integer keys are defined below.

3.1. Keys 0 and 1

Keys 0 and 1 indicate values that are exactly like the data items that would be tagged by CBOR tag 0 (RFC 3339 date/time string) or tag 1 (Posix time [TIME_T] as int or float), respectively.

3.2. Keys 4 and 5

Keys 4 and 5 are like key 1, except that the data item is an array as defined for CBOR tag 4 or 5, respectively. This can be used to include a Decimal or Bigfloat epoch-based float [TIME_T] in an extended time.

3.3. Keys -3, -6, -9, -12

The keys -3, -6, -9, -12 indicate additional decimal fractions by giving an unsigned integer (major type 0) and scaling this with the scale factor 1e-3, 1e-6, 1e-9, and 1e-12, respectively (see Table 1). More than one of these keys MUST NOT be present in one extended time data item. These additional fractions are added to a base time in seconds [SI-SECOND] indicated by a Key 1, which then MUST also be present and MUST have an integer value.

Key	meaning	example usage
-3	milliseconds	Java time
-6	microseconds	(old) UNIX time
-9	nanoseconds	(new) UNIX time
-12	picoseconds	Haskell time

Table 1: Key for decimally scaled Fractions

3.4. Key -1

Key -1 is used to indicate a time scale. The value 0 indicates UTC, the value 1 indicates TAI. If key -1 is not present, time scale value 0 is implied. Additional values can be registered in the (TBD define name for time scale registry); values MUST be integers or text strings.

(Note that there should be no time scale "GPS" - instead, the time should be converted to TAI using a single subtraction.)

3.5. Key -2

Key -2 can be used to indicate the quality of the point in time: The value 0 indicates a time obtained from a clock (past or "current" time). The value -1 indicates a future time that has been scheduled by a human. The value 1 indicates a time derived from a time obtained from a clock (such as the timestamp of a record in a log file). (TBD: Is this well-defined enough? What other cases should be considered here?)

If key -2 is not present, no information is available about the quality of the time.

3.6. Key -4

Key -4 can be used to indicate the resolution of the time provided [RESOLUTION]: "The minimum time interval that a clock can measure or whose passage a timer can detect." The value is expressed in SI seconds [SI-SECOND] and can be any positive number, such as an integer, a floating point value (major type 7 or Tag 5), or a decimal value (Tag 4).

3.7. Key -5

Key -5 can be used to indicate the accuracy of the time [IEEE1588-2008]: "The mean of the time or frequency error between the clock under test and a perfect reference clock, over an ensemble of measurements." The value is expressed in SI seconds [SI-SECOND] and can be any positive number, such as an integer, a floating point value (major type 7 or Tag 5), or a decimal value (Tag 4).

(This could be extended into more information about the way the clock source is synchronized, e.g. manually, GPS, NTP, PTP, roughtime, ...)

3.8. Key -7

Key -7 can be used to indicate the time zone that would best fit for displaying the time given to humans. (TBD: Format for the time zone information; possibly including DST information. No default; generally, the time can by default be presented as UTC/"Zulu time".)

3.9. Key -8

Key -8 can be used to indicate the location in which the time given should be interpreted (e.g., for deriving time zone information). (TBD: Format for the coordinate information; may need to contain the Datum information.)

3.10. Key -10

Key -10 can be used to indicate the calendar that would best fit for displaying the time given to humans. (TBD: Format for the calendar information. This should probably default to Gregorian.)

4. Duration Format

(TBD; this can probably use most of the same keys as for time. Clearly, ISO8601 durations are a bit different.)

5. Period Format

(TBD; this could be a pair of times, a time and a duration, a duration and a time or v.v., or a RFC 3339 period.)

6. CDDL typenames

For the use with the CBOR Data Definition Language, CDDL [I-D.greevenbosch-appsawg-cbor-cddl], the type names defined in Figure 1 are recommended:

```
etime = #6.TBDET({* (int/tstr) => any})
duration = #6.TBDED({* (int/tstr) => any})
period = #6.TBDEP({* (int/tstr) => any}) ; ?
```

Figure 1: Recommended type names for CDDL

7. IANA Considerations

IANA is requested to allocate the tags in Table 2, with the present document as the specification reference.

Tag	Data Item	Semantics
TBDET	map	[RFCthis] extended time
TBDED	map	[RFCthis] duration
TBDEP	map (?)	[RFCthis] period

Table 2: Values for Tags

(TBD: Add registry for time scales. Add registry for map keys and allocation policies for additional keys.)

RFC editor note: Please replace TBDET, TBDED, and TBDEP by the tag numbers allocated by IANA throughout the document and delete this note.

8. Security Considerations

The security considerations of RFC 7049 apply; the tags introduced here are not expected to raise security considerations beyond those.

Time, of course, has significant security considerations; these include the exploitation of ambiguities where time is security relevant (e.g., for freshness or in a validity span) or the disclosure of characteristics of the emitting system (e.g., time zone, or clock resolution and wall clock offset).

9. References

9.1. Normative References

[I-D.greevenbosch-appsawg-cbor-cddl]
Vigano, C. and H. Birkholz, "CBOR data definition language (CDDL): a notational convention to express CBOR data structures", draft-greevenbosch-appsawg-cbor-cddl-09 (work in progress), September 2016.

[IEEE1588-2008]
IEEE, "1588-2008 - IEEE Standard for a Precision Clock Synchronization Protocol for Networked Measurement and Control Systems", July 2008, <http://standards.ieee.org/findstds/standard/1588-2008.html>.

[RESOLUTION]
The Open Group Base Specifications, "Vol. 1: Base Definitions, Issue 7", Section 3.328 '(Time) Resolution', IEEE Std 1003.1-2008, 2016 Edition, 2016, <http://pubs.opengroup.org/onlinepubs/9699919799/basedefs/V1_chap03.html#tag_03_328>.

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

[RFC7049] Bormann, C. and P. Hoffman, "Concise Binary Object Representation (CBOR)", RFC 7049, DOI 10.17487/RFC7049, October 2013, <http://www.rfc-editor.org/info/rfc7049>.

[SI-SECOND]
International Organization for Standardization (ISO), "Quantities and units -- Part 3: Space and time", ISO 80000-3, March 2006.

[TIME_T] The Open Group Base Specifications, "Vol. 1: Base Definitions, Issue 7", Section 4.15 'Seconds Since the Epoch', IEEE Std 1003.1-2008, 2016 Edition, 2016, <http://pubs.opengroup.org/onlinepubs/9699919799/basedefs/V1_chap04.html#tag_04_16>.

9.2. Informative References

[TIME] Touch, J., "... time ...", unpublished manuscript, n.d..

Acknowledgements

Authors' Addresses

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

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

Ben Gamari
Well-Typed

Henk Birkholz
Fraunhofer Institute for Secure Information Technology
Rheinstrasse 75
Darmstadt 64295
Germany

Email: henk.birkholz@sit.fraunhofer.de

Network Working Group
Internet-Draft
Intended status: Informational
Expires: January 4, 2018

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

July 03, 2017

Concise data definition language (CDDL): a notational convention to express CBOR data structures
draft-greevenbosch-appsawg-cbor-cddl-11

Abstract

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

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at http://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on January 4, 2018.

Table of Contents

1. Introduction

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

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

The CBOR notational convention has the following goals:

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

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

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

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

(G5) Human and machine readable and processable.

(G6) Automatic checking of data format compliance.

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

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

This document has the following structure:

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

1.1. Requirements notation

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119, BCP 14 [RFC2119].

1.2. Terminology

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

2. The Style of Data Structure Specification

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

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

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

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

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

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

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

Two important concepts provide the foundation for CDDL:

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

2. The other important concept is that of a _type_. The entire CDDL specification defines a type (the one defined by its first _rule_), which formally is the set of CBOR instances that are acceptable 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.

2.1. Groups and Composition in CDDL

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

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

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

A group can be placed in (round) parentheses, and given a name by using it in a rule:

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

Figure 1: A basic group

Or a group can just be used in the definition of something else:

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

Figure 2: Using a group in a map

which, given the above rule for pii, is identical to:

```
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.

The parentheses for groups are optional when there is some other set of brackets present, so it would be slightly more natural to express Figure 2 as:

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

Groups can be used to factor out common parts of structs, e.g., instead of writing:

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

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

one can choose a name for the common subgroup and write:

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

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

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

Figure 4: Using a group for factorization

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

2.1.1. Usage

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

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

2.1.2. Syntax

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

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

An entry consists of a _keytype_ and a _valuetype_:

- _keytype_ is either an atom used as the actual key or a type in general. The latter case may be needed when using groups in a table context, where the actual keys are of lesser importance than the key types, e.g in contexts verifying incoming data.
- _valuetype_ is a type, which could be derived from the major types defined in [RFC7049], could be a convenience valuetype defined in this document (Appendix E) or the name of a type defined in the specification.

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

2.2. Types

2.2.1. Values

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

2.2.2. Choices

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

Choices of values can be used to express enumerations:

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

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

```
address = { delivery }

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

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

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

```
attire /= "swimwear"

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

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

2.2.2.1. Ranges

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

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

CDDL currently only allows ranges between numbers [_range].

2.2.2.2. Turning a group into a choice

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

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

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

2.2.3. Representation Types

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

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

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

2.2.4. Root type

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

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

(Note that there is no way to use a group as a root - it must be a type. Using a group as the root might be employed as a way to specify a CBOR sequence in a future version of this specification; this would act as if that group is used in an array and the data items in that fictional array form the members of the CBOR sequence.)

3. Syntax

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

3.1. General conventions

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

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

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

  * Names are case sensitive.

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

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

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

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

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

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

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

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

- Text strings are enclosed by double quotation '"' characters. They follow the conventions for strings as defined in section 7 of [RFC7159]. (ABNF users may want to note that there is no support in CDDL for the concept of case insensitivity in text strings; if necessary, regular expressions can be used (Section 3.8.3).)

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

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

Example:

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

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

3.2. Occurrence

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

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

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

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

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

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

```
  ? bedroom: size,
```

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

3.3. Predefined names for types

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

The following keywords for primitive datatypes are defined:

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

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

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

"int" An unsigned integer or a negative integer.

"float16" IEEE 754 half-precision float (major type 7, additional information 25).

"float32" IEEE 754 single-precision float (major type 7, additional information 26).

"float64" IEEE 754 double-precision float (major type 7, additional information 27).

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

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

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

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

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

3.4. Arrays

Array definitions surround a group with square brackets.

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

For example:

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

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

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

3.5. Maps

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

3.5.1. Structs

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

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

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

The following is an example of a structure:

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

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

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

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

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

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

If a text string key does not match the syntax for an identifier (or if the specifier just happens to prefer using double quotes), the text string syntax can also be used in the member key position, followed by a colon. The above example could therefore have been written with quoted strings in the member key positions.

All the types defined can be used in a keytype position by following them with a double arrow. A string also is a (single-valued) type, so another form for this example is:

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

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

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

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

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

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

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

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

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

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

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

Figure 5: Personal Data: Example for extensibility

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

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

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

3.5.2. Tables

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

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

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

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

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

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

3.6. Tags

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

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

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

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

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

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

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

3.7. Unwrapping

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

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

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

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

basic-header = { basic-header-group }

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

Unwrapping simplifies this to:

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

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

(Note that leaving out the first unwrap operator in the latter example would lead to nesting the basic-header in its own map inside the advanced-header, while, with the unwrapped basic-header, the definition of the group inside basic-header is essentially repeated inside advanced-header, leading to a single map. This can be used for various applications often solved by inheritance in programming languages. The effect of unwrapping can also be described as "threading in" the group or type inside the referenced type, which suggested the thread-like "~" character.)

3.8. Controls

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

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

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

3.8.1. Control operator .size

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

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

Figure 6: Control for size in bytes

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

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

Figure 7: Control for integer size in bytes

Note that, as with value restrictions in CDDL, this control is not a representation constraint; a number that fits into fewer bytes can still be represented in that form, and an inefficient implementation could use a longer form (unless that is restricted by some format constraints outside of CDDL, such as the rules in Section 3.9 of [RFC7049]).

3.8.2. Control operator .bits

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

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

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

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

Figure 8: Control for what bits can be set

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

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

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

3.8.3. Control operator .regexp

A ".regexp" control indicates that the text string given as a target needs to match the PCRE regular expression given as a value in the control type, where that regular expression is anchored on both sides. (If anchoring is not desired for a side, ".*" needs to be inserted there.)

```
nai = tstr .regexp "\\w+@\\w+(\\.\\w+)+"
```

Figure 9: Control with a PCRE regexp

The CDDL tool proposes:

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

3.8.4. Control operators .cbor and .cborseq

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

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

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

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

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

3.8.5. Control operators .within and .and

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

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

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

```
"type1 .within type2"
```

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

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

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

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

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

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

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

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

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

3.9. Socket/Plug

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

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

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

; later, in a different file

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

; and, maybe in another file

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

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

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

To pick up the example illustrated in Figure 5, the socket/plug mechanism could be used as shown in Figure 10:

```
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 10: Personal Data example: Using socket/plug extensibility

3.10. Generics

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

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

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

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

3.11. Operator Precedence

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

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

For instance, in

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

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

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

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

Similarly, in

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

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

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

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

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

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

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

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

Table 1: Summary of operator precedences

4. Making Use of CDDL

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

4.1. As a guide to a human user

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

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

4.2. For automated checking of CBOR data structure

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

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

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

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

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

4.3. For data analysis tools

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

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

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

5. Security considerations

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

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

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

6. IANA considerations

This document does not require any IANA registrations.

7. Acknowledgements

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

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

Useful feedback came from Joe Hildebrand, Sean Leonard and Jim Schaad.

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

8. References

8.1. Normative References

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

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

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

[RFC7049] Bormann, C. and P. Hoffman, "Concise Binary Object Representation (CBOR)", RFC 7049, DOI 10.17487/RFC7049, October 2013, <http://www.rfc-editor.org/info/rfc7049>.

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

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

8.2. Informative References

[I-D.ietf-anima-grasp]
Bormann, C., Carpenter, B., and B. Liu, "A Generic Autonomic Signaling Protocol (GRASP)", draft-ietf-anima-grasp-14 (work in progress), July 2017.

[I-D.ietf-core-senml]
Jennings, C., Shelby, Z., Arkko, J., Keranen, A., and C. Bormann, "Media Types for Sensor Measurement Lists (SenML)", draft-ietf-core-senml-10 (work in progress), July 2017.

[I-D.ietf-cose-msg]
Schaad, J., "CBOR Object Signing and Encryption (COSE)", draft-ietf-cose-msg-24 (work in progress), November 2016.

[I-D.newton-json-content-rules]
Newton, A. and P. Cordell, "A Language for Rules Describing JSON Content", draft-newton-json-content-rules-08 (work in progress), March 2017.

[RELAXNG] OASIS, "RELAX-NG Compact Syntax", November 2002, <http://relaxng.org/compact-20021121.html>.

[RFC4648] Josefsson, S., "The Base16, Base32, and Base64 Data Encodings", RFC 4648, DOI 10.17487/RFC4648, October 2006, <http://www.rfc-editor.org/info/rfc4648>.

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

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

Appendix A. Cemetery

The following ideas have been buried in the discussions leading up to the present specification:

- <...> as syntax for enumerations. We view values to be just another type (a very specific type with just one member), so that an enumeration can be denoted as a choice using "/" as the delimiter of choices. Because of this, no evidence is present that a separate syntax for enumerations is needed.

A.1. Resolved Issues

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

- CDDL distinguishes the various CBOR number types, but there is only one number type in JSON. There is no effect in specifying a precision (float16/float32/float64) when using CDDL for specifying JSON data structures. (The current validator implementation Appendix F does not handle this very well, either.)

Appendix B. (Not used.)

Appendix C. Change Log

Changes from version 00 to version 01:

- Removed constants
- Updated the tag mechanism
- Extended the map structure
- Added examples

Changes from version 01 to version 02:

- Fixed example

Changes from version 02 to version 03:

- Added information about characters used in names
- Added text about an overall data structure and order of definition of fields
- Added text about encoding of keys
- Added table with keywords
- Strings and integer writing conventions

- Added ABNF

Changes from version 03 to version 04:

- Removed optional fields for non-maps
- Defined all key/value pairs in maps are considered optional from the CDDL perspective
- Allow omission of type of keys for maps with only text string and integer keys
- Changed order of definitions
- Updated fruit and moves examples
- Renamed the "Philosophy" section to "Using CDDL", and added more text about CDDL usage
- Several editorials

Changes from version 04 to version 05:

- Added text about alternative datatypes and any datatype
- Fixed typos
- Restructured syntax and semantics

Changes from version 05 to version 05:

- Fixed the ABNF for choices (no longer need to write a: (b/c))
- Added group choices (//)
- Added /= and //=
- Added experimental socket/plug
- Added aliases text, bytes, null to prelude
- Documented generics
- Fixed more typos

Changes from 06 to 07:

- .cbor, .cborseq, .within, .and

- Define .size on uint
- Extended Diagnostic Notation
- Precedence discussion and table
- Remove some of the "issues" that can only be understood with historical context
- Prefer "text" over "tstr" in some of the examples
- Add "unsigned" to the prelude

Changes from 07 to 08:

- .lt, .le, .eq, .ne, .gt, .ge
- .default

Changes from 08 to 09:

- Take annotations and socket/plug out of the nursery; they have been battle-proven enough.
- Define a value notation for byte strings as well.
- Removed discussion section that was no longer relevant; move "Resolved Issues" to appendix.

Changes from 09 to 10:

- Remove a long but not very elucidating example. (Maybe we'll add back some shorter examples later.)
- A few clarifications.
- Updated author list.

Changes from 10 to 11:

- Define unwrapping operator ~
- Change term for annotation into "control" (but leave "annotate" for when it actually is meant in that sense)

Appendix D. ABNF grammar

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

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

typename = id
groupname = id

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

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

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

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

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

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

ctlop = "." id

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

grpchoice = *grpent

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

memberkey = type1 S "=>"
          / bareword S ":"
          / value S ":"
```

```
   bareword = id

   optcom = S ["," S]

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

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

   value = number
         / text
         / bytes

   int = ["-"] uint

   ; This is a float if it has fraction or exponent; int otherwise
   number = int ["." fraction] ["e" exponent ]
   fraction = 1*DIGIT
   exponent = int

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

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

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

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

Figure 11: CDDL ABNF

Appendix E. Standard Prelude

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

```
any = #

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

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

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

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

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

Figure 12: CDDL Prelude

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

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

E.1. Use with JSON

The JSON generic data model (implicit in [RFC7159]) 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 13: 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.)

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

JSON applications that want to stay compatible with I-JSON therefore may want to define integer types with more limited ranges, such as in Figure 14. 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 14: I-JSON types for CDDL (not part of prelude)

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

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

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

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

Appendix F. The CDDL tool

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

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

Figure 15: CDDL tool usage

Install on a system with a modern Ruby via:

```
gem install cddl
```

Figure 16: CDDL tool installation

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

Appendix G. Extended Diagnostic Notation

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

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

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

G.1. White space in byte string notation

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

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

G.2. Text in byte string notation

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

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

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

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

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

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

G.4. Concatenated Strings

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

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

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

Similarly, the following byte string values are equivalent

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

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

G.5. Hexadecimal, octal, and binary numbers

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

The following are equivalent:

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

As are:

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

G.6. Comments

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

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

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

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

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

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

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

Appendix H. Examples

This section contains various examples of structures defined using CDDL.

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

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

These examples all happen to describe data that is interchanged in JSON. Examples for CDDL definitions of data that is interchanged in

CBOR can be found in [I-D.ietf-cose-msg], [I-D.ietf-anima-grasp], or [I-D.ietf-core-senml].

H.1. RFC 7071

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

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

reputation-context = (
  application: text
)

reputon-list = (
  reputons: reputon-array
)

reputon-array = [* reputon]

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

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

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

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

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

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

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

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

H.1.1. Examples from JSON Content Rules

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

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

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

Figure 17: JCR, Figure 2, in CDDL

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

The CDDL tool creates the below example instance for this:

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

Figure 4 of the JCR I-D in CDDL:

```
root = { image }

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

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

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

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

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

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

The CDDL tool creates the below example instance for this:

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

Editorial Comments

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

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

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

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

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

[_abnftodo] Potential improvements: the prefixed byte strings are more liberally specified than they actually are. [^_abnfdontdo]: representation indicators are not supported. - and this will stay so.

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

Authors' Addresses

Henk Birkholz
Fraunhofer SIT
Rheinstrasse 75
Darmstadt 64295
Germany

Email: henk.birkholz@sit.fraunhofer.de

Christoph Vigano
Universitaet Bremen

Email: christoph.vigano@uni-bremen.de

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

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