

Network Working Group
Internet-Draft
Intended status: Informational
Expires: December 16, 2017

C. Bormann
Universitaet Bremen TZI
B. Gamari
Well-Typed
H. Birkholz
Fraunhofer SIT
June 14, 2017

Concise Binary Object Representation (CBOR) Tags for Time, Duration, and
Period
draft-bormann-cbor-time-tag-01

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, and anticipates the definition of related CBOR tags for duration and time period. It is intended as the reference document for the IANA registration of the CBOR tags defined.

Note to Readers

Version -00 of the present draft opened up the possibilities provided by extended representations of time in CBOR. The present version -01 consolidates this draft to non-speculative content, the normative parts of which are believed will stay unchanged during further development of the draft. This version is provided to aid the registration of the CBOR tag immediately needed. Further versions will re-introduce some of the material from -00, but in a more concrete form.

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 December 16, 2017.

Copyright Notice

Copyright (c) 2017 IETF Trust and the persons identified as the document authors. All rights reserved.

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

Table of Contents

1. Introduction	3
1.1. Terminology	3
1.2. Background	3
2. Objectives	3
3. Time Format	4
3.1. Key 1	5
3.2. Keys 4 and 5	5
3.3. Keys -3, -6, -9, -12, -15, -18	5
4. CDDL typenames	5
5. IANA Considerations	6
6. Security Considerations	6
7. References	7
7.1. Normative References	7
7.2. Informative References	8
Acknowledgements	8
Authors' Addresses	8

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++'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:

- o 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).

Not currently addressed, but possibly covered by the definition of additional map keys for the map inside the tag:

- o 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).

- o 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.)
- o 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.

Additional tags might later be defined for duration and period. The objectives for such duration and period tags are likely 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 for negative integer and text string values of the key. Not understanding key/value for unsigned keys is an error.

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:

- o a higher precision time offset to be added to the base time,

Future keys may add:

- o a reference time scale and epoch different from the default UTC and 1970-01-01
- o information about clock source and precision, accuracy, and resolution
- o 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. Key 1

Key 1 indicates a value that is exactly like the data item that would be tagged by CBOR tag 1 (Posix time [TIME_T] as int or float).

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, -15, -18

The keys -3, -6, -9, -12, -15 and -18 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, 1e-12, 1e-15, and 1e-18, 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
-15	femtoseconds	(future)
-18	attoseconds	(future)

Table 1: Key for decimally scaled Fractions

4. 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})
```

Figure 1: Recommended type names for CDDL

5. IANA Considerations

IANA is requested to allocate the tags in Table 2 from the FCFS space, 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

Although duration and period are not yet defined in the present version of this document, the tag values for duration and period are requested at the same time as the value for extended time in order to achieve allocation of all three values as a contiguous set.

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

6. 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).

7. References

7.1. Normative References

- [I-D.greevenbosch-appsawg-cbor-cddl]
Birkholz, H., Vigano, C., and C. Bormann, "CBOR data definition language (CDDL): a notational convention to express CBOR data structures", draft-greevenbosch-appsawg-cbor-cddl-10 (work in progress), March 2017.
- [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>.

7.2. Informative References

[TIME] Touch, J., "Resolving Multiple Time Scales in the Internet", draft-touch-time-02 (work in progress), April 2017.

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
117 Middle Rd.
Portsmouth, NH 03801
United States

Email: ben@well-typed.com

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: July 15, 2017

C. Bormann
Universitaet Bremen TZI
January 11, 2017

Concise Binary Object Representation (CBOR) Tag for CBOR Templates
draft-bormann-lpwan-cbor-template-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.

The present document makes use of this extensibility to define a CBOR tag for a variable within a CBOR data item, which then could be filled in by a separate process (e.g., from another CBOR data item).

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 July 15, 2017.

Copyright Notice

Copyright (c) 2017 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<http://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must

include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

Table of Contents

1. Introduction	2
2. Variable	2
2.1. Example	3
3. CDDL typename	3
4. IANA Considerations	4
5. Security Considerations	4
6. References	5
6.1. Normative References	5
6.2. Informative References	5
Contributors	5
Acknowledgements	5
Author's Address	5

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.

With the work on static compression for CoAP [RFC7252] and CoAP application protocols [I-D.vanderstok-core-comi] on LPWANs [I-D.ietf-lpwan-coap-static-context-hc], there appears to be a need for defining CBOR data items that have within them some open positions in them that can later be filled in from a separate source (such as another CBOR data item). The anchor points for this substitution are called "variables" in this specification.

This document defines a CBOR tag for a variable in a CBOR data item. It is intended as the reference document for the IANA registration of the tag defined.

2. Variable

A variable is a CBOR data item that typically is included as a part of a larger data item (the "CBOR template"). In a process that is outside the scope of this specification, the variable is then substituted by an actual value in order to yield an instance from the template.

A variable is identified by the data item within the tag, the "variable identifier". Typically, variables are numbered by integers. Some applications may also benefit from the use of strings as identifiers. The specification of the tag does not make a restriction on the type of the identifier; however note that very complex variable identifiers may benefit from canonicalization to enable their comparison, cf. section 3.9 of [RFC7049].

We term a CBOR data item that contains one or more variables as a "CBOR template"; generally processes that accept CBOR templates with variables will also accept CBOR data items without variables, so we accept this as a degenerate case for "CBOR template". Note that a template may use the same variable (i.e., a variable with the same identifier) in multiple positions, leading to multiple substitutions of the same value.

2.1. Example

An example for a CBOR template in diagnostic notation:

```
{ "name": "Carsten Bormann",  
  "place": 42(0) }
```

When this template undergoes substitution, with the variable 0 set to the value "Bremen", this would result in the data item:

```
{ "name": "Carsten Bormann",  
  "place": "Bremen" }
```

3. CDDL typename

CDDL [I-D.greevenbosch-appsawg-cbor-cddl] definitions will typically describe the structure of a data item after substitution.

However, when the CDDL definition needs to explicitly identify the positions where substitutions can occur, the typename defined in Figure 1 is recommended:

```
variable<varid> = #6.42(varid)
```

Figure 1: Recommended typenames for CDDL

4. IANA Considerations

RFC-Editor note: Please replace "42" throughout this document by the actual tag allocated and delete this note.

IANA is requested to allocate the tag 42 as in Table 1, with the present document as the specification reference.

Tag	Data Item	Semantics
42	any	CBOR variable

Table 1: Values for Tags

5. Security Considerations

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

Obviously, any process for performing variable substitution as outlined in Section 2 needs to ensure that all of its inputs are derived considering the security objectives, and that the inputs are actually intended to fit together for this substitution.

6. References

6.1. 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.
- [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>>.

6.2. Informative References

- [I-D.ietf-lpwan-coap-static-context-hc]
Minaburo, A. and L. Toutain, "6LPWA Static Context Header Compression (SCHC) for CoAP", draft-ietf-lpwan-coap-static-context-hc-00 (work in progress), December 2016.
- [I-D.vanderstok-core-comi]
Stok, P., Bierman, A., Veillette, M., and A. Pelov, "CoAP Management Interface", draft-vanderstok-core-comi-10 (work in progress), October 2016.
- [RFC7252] Shelby, Z., Hartke, K., and C. Bormann, "The Constrained Application Protocol (CoAP)", RFC 7252, DOI 10.17487/RFC7252, June 2014, <<http://www.rfc-editor.org/info/rfc7252>>.

Contributors

Laurent Toutain suggested the creation of a mechanism for indicating which parts of a CBOR data item are not yet available or subject to change.

Acknowledgements

TBD

Author's Address

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

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

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.

Copyright Notice

Copyright (c) 2017 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<http://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must

include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

Table of Contents

1. Introduction	3
1.1. Requirements notation	4
1.2. Terminology	4
2. The Style of Data Structure Specification	4
2.1. Groups and Composition in CDDL	6
2.1.1. Usage	8
2.1.2. Syntax	8
2.2. Types	8
2.2.1. Values	9
2.2.2. Choices	9
2.2.3. Representation Types	10
2.2.4. Root type	11
3. Syntax	11
3.1. General conventions	11
3.2. Occurrence	13
3.3. Predefined names for types	14
3.4. Arrays	14
3.5. Maps	15
3.5.1. Structs	15
3.5.2. Tables	18
3.6. Tags	19
3.7. Unwrapping	19
3.8. Controls	20
3.8.1. Control operator .size	21
3.8.2. Control operator .bits	21
3.8.3. Control operator .regexp	22
3.8.4. Control operators .cbor and .cborseq	23
3.8.5. Control operators .within and .and	23
3.8.6. Control operators .lt, .le, .gt, .ge, .eq, .ne, and .default	24
3.9. Socket/Plug	24
3.10. Generics	26
3.11. Operator Precedence	26
4. Making Use of CDDL	28
4.1. As a guide to a human user	28
4.2. For automated checking of CBOR data structure	28
4.3. For data analysis tools	29
5. Security considerations	29
6. IANA considerations	29
7. Acknowledgements	30
8. References	30
8.1. Normative References	30

8.2. Informative References	31
Appendix A. Cemetery	31
A.1. Resolved Issues	32
Appendix B. (Not used.)	32
Appendix C. Change Log	32
Appendix D. ABNF grammar	35
Appendix E. Standard Prelude	37
E.1. Use with JSON	39
Appendix F. The CDDL tool	41
Appendix G. Extended Diagnostic Notation	41
G.1. White space in byte string notation	42
G.2. Text in byte string notation	42
G.3. Embedded CBOR and CBOR sequences in byte strings	42
G.4. Concatenated Strings	43
G.5. Hexadecimal, octal, and binary numbers	43
G.6. Comments	44
Appendix H. Examples	44
H.1. RFC 7071	45
H.1.1. Examples from JSON Content Rules	48
Authors' Addresses	50

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:

- o 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.
- o 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.
- o 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.
- o 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:

- o The start of a group can be marked by '('
- o The end of a group can be marked by ')'
- o 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_`:

- o `_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.
- o `_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

- o 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.
- o 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.
- o Comments are started by a ';' (semicolon) character and finish at the end of a line (LF or CRLF).
- o outside strings, whitespace (spaces, newlines, and comments) is used to separate syntactic elements for readability (and to separate identifiers or numbers that follow each other); it is otherwise completely optional.
- o Hexadecimal numbers are preceded by '0x' (without quotes, lower case x), and are case insensitive. Similarly, binary numbers are preceded by '0b'.
- o Text strings are enclosed by double quotation '"' characters. They follow the conventions for strings as defined in section 7 of [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).)
- o Byte strings are enclosed by single quotation "'" characters and may be prefixed by "h" or "b64". If unprefixed, the string is interpreted as with a text string, except that single quotes must be escaped and that the UTF-8 bytes resulting are marked as a byte string (major type 2). If prefixed as "h" or "b64", the string is interpreted as a sequence of 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]
- o 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 this 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 = &(amp;
    fin: 8,
    syn: 9,
    rst: 10,
    psh: 11,
    ack: 12,
    urg: 13,
    ece: 14,
    cwr: 15,
    ns: 0,
) / (4..7) ; data offset bits

rxwbits = uint .bits rxw
rxw = &(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, greater 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 control 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/plugin 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:

- o 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:

- o <...> 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

- o 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.
- o 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:

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

Changes from version 01 to version 02:

- o Fixed example

Changes from version 02 to version 03:

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

- o Added ABNF

Changes from version 03 to version 04:

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

Changes from version 04 to version 05:

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

Changes from version 05 to version 05:

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

Changes from 06 to 07:

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

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

Changes from 07 to 08:

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

Changes from 08 to 09:

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

Changes from 09 to 10:

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

Changes from 10 to 11:

- o Define unwrapping operator `~`
- o 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'636666F6FF6'
<<>>           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": "Dasypsecta",
      "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": "intragyril"
    }
  ]
}
```

H.1.1.1. 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.
- [_bitsemdian] 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 .regex 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

Network Working Group
Internet-Draft
Intended status: Standards Track
Expires: October 14, 2017

C. Bormann
Universitaet Bremen TZI
P. Hoffman
ICANN
April 12, 2017

Concise Binary Object Representation (CBOR)
draft-ietf-cbor-7049bis-00

Abstract

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

Contributing

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

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

Status of This Memo

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

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at <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 October 14, 2017.

Copyright Notice

Copyright (c) 2017 IETF Trust and the persons identified as the document authors. All rights reserved.

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

Table of Contents

1. Introduction	3
1.1. Objectives	4
1.2. Terminology	5
2. Specification of the CBOR Encoding	6
2.1. Major Types	7
2.2. Indefinite Lengths for Some Major Types	9
2.2.1. Indefinite-Length Arrays and Maps	9
2.2.2. Indefinite-Length Byte Strings and Text Strings	11
2.3. Floating-Point Numbers and Values with No Content	12
2.4. Optional Tagging of Items	14
2.4.1. Date and Time	16
2.4.2. Bignums	16
2.4.3. Decimal Fractions and Bigfloats	16
2.4.4. Content Hints	18
2.4.4.1. Encoded CBOR Data Item	18
2.4.4.2. Expected Later Encoding for CBOR-to-JSON Converters	18
2.4.4.3. Encoded Text	18
2.4.5. Self-Describe CBOR	19
3. Creating CBOR-Based Protocols	19
3.1. CBOR in Streaming Applications	20
3.2. Generic Encoders and Decoders	20
3.3. Syntax Errors	21
3.3.1. Incomplete CBOR Data Items	21
3.3.2. Malformed Indefinite-Length Items	22
3.3.3. Unknown Additional Information Values	22
3.4. Other Decoding Errors	22
3.5. Handling Unknown Simple Values and Tags	23
3.6. Numbers	23
3.7. Specifying Keys for Maps	24
3.8. Undefined Values	25

3.9. Canonical CBOR	26
3.10. Strict Mode	27
4. Converting Data between CBOR and JSON	28
4.1. Converting from CBOR to JSON	29
4.2. Converting from JSON to CBOR	30
5. Future Evolution of CBOR	31
5.1. Extension Points	31
5.2. Curating the Additional Information Space	32
6. Diagnostic Notation	32
6.1. Encoding Indicators	33
7. IANA Considerations	34
7.1. Simple Values Registry	34
7.2. Tags Registry	34
7.3. Media Type ("MIME Type")	35
7.4. CoAP Content-Format	36
7.5. The +cbor Structured Syntax Suffix Registration	36
8. Security Considerations	37
9. Acknowledgements	38
10. References	38
10.1. Normative References	38
10.2. Informative References	39
Appendix A. Examples	41
Appendix B. Jump Table	45
Appendix C. Pseudocode	48
Appendix D. Half-Precision	50
Appendix E. Comparison of Other Binary Formats to CBOR's Design Objectives	51
E.1. ASN.1 DER, BER, and PER	52
E.2. MessagePack	52
E.3. BSON	53
E.4. UBJSON	53
E.5. MSDTP: RFC 713	53
E.6. Conciseness on the Wire	53
Authors' Addresses	54

1. Introduction

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

The format defined here follows some specific design goals that are not well met by current formats. The underlying data model is an extended version of the JSON data model [RFC7159]. It is important to note that this is not a proposal that the grammar in RFC 7159 be extended in general, since doing so would cause a significant

backwards incompatibility with already deployed JSON documents. Instead, this document simply defines its own data model that starts from JSON.

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

1.1. Objectives

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

1. The representation must be able to unambiguously encode most common data formats used in Internet standards.
 - * It must represent a reasonable set of basic data types and structures using binary encoding. "Reasonable" here is largely influenced by the capabilities of JSON, with the major addition of binary byte strings. The structures supported are limited to arrays and trees; loops and lattice-style graphs are not supported.
 - * There is no requirement that all data formats be uniquely encoded; that is, it is acceptable that the number "7" might be encoded in multiple different ways.
2. The code for an encoder or decoder must be able to be compact in order to support systems with very limited memory, processor power, and instruction sets.
 - * An encoder and a decoder need to be implementable in a very small amount of code (for example, in class 1 constrained nodes as defined in [RFC7228]).
 - * The format should use contemporary machine representations of data (for example, not requiring binary-to-decimal conversion).
3. Data must be able to be decoded without a schema description.
 - * Similar to JSON, encoded data should be self-describing so that a generic decoder can be written.
4. The serialization must be reasonably compact, but data compactness is secondary to code compactness for the encoder and decoder.

- * "Reasonable" here is bounded by JSON as an upper bound in size, and by implementation complexity maintaining a lower bound. Using either general compression schemes or extensive bit-fiddling violates the complexity goals.
5. The format must be applicable to both constrained nodes and high-volume applications.
 - * This means it must be reasonably frugal in CPU usage for both encoding and decoding. This is relevant both for constrained nodes and for potential usage in applications with a very high volume of data.
 6. The format must support all JSON data types for conversion to and from JSON.
 - * It must support a reasonable level of conversion as long as the data represented is within the capabilities of JSON. It must be possible to define a unidirectional mapping towards JSON for all types of data.
 7. The format must be extensible, and the extended data must be decodable by earlier decoders.
 - * The format is designed for decades of use.
 - * The format must support a form of extensibility that allows fallback so that a decoder that does not understand an extension can still decode the message.
 - * The format must be able to be extended in the future by later IETF standards.

1.2. Terminology

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

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

This specification makes use of the following terminology:

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

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

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

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

Well-formed: A data item that follows the syntactic structure of CBOR. A well-formed data item uses the initial bytes and the byte strings and/or data items that are implied by their values as defined in CBOR and is not followed by extraneous data.

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

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

Where bit arithmetic or data types are explained, this document uses the notation familiar from the programming language C, except that "***" denotes exponentiation. Similar to the "0x" notation for hexadecimal numbers, numbers in binary notation are prefixed with "0b". Underscores can be added to such a number solely for readability, so 0b00100001 (0x21) might be written 0b001_00001 to emphasize the desired interpretation of the bits in the byte; in this case, it is split into three bits and five bits.

2. Specification of the CBOR Encoding

A CBOR-encoded data item is structured and encoded as described in this section. The encoding is summarized in Table 5.

The initial byte of each data item contains both information about the major type (the high-order 3 bits, described in Section 2.1) and additional information (the low-order 5 bits). When the value of the

additional information is less than 24, it is directly used as a small unsigned integer. When it is 24 to 27, the additional bytes for a variable-length integer immediately follow; the values 24 to 27 of the additional information specify that its length is a 1-, 2-, 4-, or 8-byte unsigned integer, respectively. Additional information value 31 is used for indefinite-length items, described in Section 2.2. Additional information values 28 to 30 are reserved for future expansion.

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

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

2.1. Major Types

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

Major type 0: an unsigned integer. The 5-bit additional information is either the integer itself (for additional information values 0 through 23) or the length of additional data. Additional information 24 means the value is represented in an additional uint8_t, 25 means a uint16_t, 26 means a uint32_t, and 27 means a uint64_t. For example, the integer 10 is denoted as the one byte 0b000_01010 (major type 0, additional information 10). The integer 500 would be 0b000_11001 (major type 0, additional information 25) followed by the two bytes 0x01f4, which is 500 in decimal.

Major type 1: a negative integer. The encoding follows the rules for unsigned integers (major type 0), except that the value is then -1 minus the encoded unsigned integer. For example, the integer -500 would be 0b001_11001 (major type 1, additional information 25) followed by the two bytes 0x01f3, which is 499 in decimal.

Major type 2: a byte string. The string's length in bytes is represented following the rules for positive integers (major type 0). For example, a byte string whose length is 5 would have an

initial byte of 0b010_00101 (major type 2, additional information 5 for the length), followed by 5 bytes of binary content. A byte string whose length is 500 would have 3 initial bytes of 0b010_11001 (major type 2, additional information 25 to indicate a two-byte length) followed by the two bytes 0x01f4 for a length of 500, followed by 500 bytes of binary content.

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

Major type 4: an array of data items. Arrays are also called lists, sequences, or tuples. The array's length follows the rules for byte strings (major type 2), except that the length denotes the number of data items, not the length in bytes that the array takes up. Items in an array do not need to all be of the same type. For example, an array that contains 10 items of any type would have an initial byte of 0b100_01010 (major type of 4, additional information of 10 for the length) followed by the 10 remaining items.

Major type 5: a map of pairs of data items. Maps are also called tables, dictionaries, hashes, or objects (in JSON). A map is comprised of pairs of data items, each pair consisting of a key that is immediately followed by a value. The map's length follows the rules for byte strings (major type 2), except that the length denotes the number of pairs, not the length in bytes that the map takes up. For example, a map that contains 9 pairs would have an initial byte of 0b101_01001 (major type of 5, additional information of 9 for the number of pairs) followed by the 18 remaining items. The first item is the first key, the second item is the first value, the third item is the second key, and so on. A map that has duplicate keys may be well-formed, but it is not valid, and thus it causes indeterminate decoding; see also Section 3.7.

Major type 6: optional semantic tagging of other major types. See Section 2.4.

Major type 7: floating-point numbers and simple data types that need no content, as well as the "break" stop code. See Section 2.3.

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

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

2.2. Indefinite Lengths for Some Major Types

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

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

2.2.1. Indefinite-Length Arrays and Maps

Indefinite-length arrays and maps are simply opened without indicating the number of data items that will be included in the array or map, using the additional information value of 31. The initial major type and additional information byte is followed by the elements of the array or map, just as they would be in other arrays or maps. The end of the array or map is indicated by encoding a "break" stop code in a place where the next data item would normally have been included. The "break" is encoded with major type 7 and additional information value 31 (0b111_1111) but is not itself a data item: it is just a syntactic feature to close the array or map. That is, the "break" stop code comes after the last item in the array or map, and it cannot occur anywhere else in place of a data item. In this way, indefinite-length arrays and maps look identical to other arrays and maps except for beginning with the additional information value 31 and ending with the "break" stop code.

Arrays and maps with indefinite lengths allow any number of items (for arrays) and key/value pairs (for maps) to be given before the "break" stop code. There is no restriction against nesting indefinite-length array or map items. A "break" only terminates a single item, so nested indefinite-length items need exactly as many "break" stop codes as there are type bytes starting an indefinite-length item.

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

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

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

```
0x9f018202039f0405ffff
9F      -- Start indefinite-length array
  01      -- 1
  82      -- Array of length 2
    02    -- 2
    03    -- 3
  9F      -- Start indefinite-length array
    04    -- 4
    05    -- 5
  FF      -- "break" (inner array)
FF      -- "break" (outer array)
```

```
0x9f01820203820405ff
9F      -- Start indefinite-length array
  01      -- 1
  82      -- Array of length 2
    02    -- 2
    03    -- 3
  82      -- Array of length 2
    04    -- 4
    05    -- 5
FF      -- "break"
```

```

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

```

```

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

```

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

```

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

```

2.2.2. Indefinite-Length Byte Strings and Text Strings

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

length string is encoded similarly to how an indefinite-length array of its chunks would be encoded, except that the major type of the indefinite-length string is that of a (text or byte) string and matches the major types of its chunks.

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

For example, assume the sequence:

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

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

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

Text strings with indefinite lengths act the same as byte strings with indefinite lengths, except that all their chunks MUST be definite-length text strings. Note that this implies that the bytes of a single UTF-8 character cannot be spread between chunks: a new chunk can only be started at a character boundary.

2.3. Floating-Point Numbers and Values with No Content

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

5-Bit Value	Semantics
0..23	Simple value (value 0..23)
24	Simple value (value 32..255 in following byte)
25	IEEE 754 Half-Precision Float (16 bits follow)
26	IEEE 754 Single-Precision Float (32 bits follow)
27	IEEE 754 Double-Precision Float (64 bits follow)
28-30	(Unassigned)
31	"break" stop code for indefinite-length items

Table 1: Values for Additional Information in Major Type 7

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

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

Table 2: Simple Values

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

2.4. Optional Tagging of Items

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

The initial bytes of the tag follow the rules for positive integers (major type 0). The tag is followed by a single data item of any type. For example, assume that a byte string of length 12 is marked with a tag to indicate it is a positive bignum (Section 2.4.2). This would be marked as 0b110_00010 (major type 6, additional information 2 for the tag) followed by 0b010_01100 (major type 2, additional information of 12 for the length) followed by the 12 bytes of the bignum.

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

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

IANA maintains a registry of tag values as described in Section 7.2. Table 3 provides a list of initial values, with definitions in the rest of this section.

+-----+-----+-----+-----+			
Tag	Data Item	Semantics	

0	UTF-8 string	Standard date/time string; see Section 2.4.1
1	multiple	Epoch-based date/time; see Section 2.4.1
2	byte string	Positive bignum; see Section 2.4.2
3	byte string	Negative bignum; see Section 2.4.2
4	array	Decimal fraction; see Section 2.4.3
5	array	Bigfloat; see Section 2.4.3
6..20	(Unassigned)	(Unassigned)
21	multiple	Expected conversion to base64url encoding; see Section 2.4.4.2
22	multiple	Expected conversion to base64 encoding; see Section 2.4.4.2
23	multiple	Expected conversion to base16 encoding; see Section 2.4.4.2
24	byte string	Encoded CBOR data item; see Section 2.4.4.1
25..31	(Unassigned)	(Unassigned)
32	UTF-8 string	URI; see Section 2.4.4.3
33	UTF-8 string	base64url; see Section 2.4.4.3
34	UTF-8 string	base64; see Section 2.4.4.3
35	UTF-8 string	Regular expression; see Section 2.4.4.3
36	UTF-8 string	MIME message; see Section 2.4.4.3
37..55798	(Unassigned)	(Unassigned)
55799	multiple	Self-describe CBOR; see Section 2.4.5
55800+	(Unassigned)	(Unassigned)

Table 3: Values for Tags

2.4.1. Date and Time

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

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

2.4.2. Bignums

Bignums are integers that do not fit into the basic integer representations provided by major types 0 and 1. They are encoded as a byte string data item, which is interpreted as an unsigned integer n in network byte order. For tag value 2, the value of the bignum is n . For tag value 3, the value of the bignum is $-1 - n$. Decoders that understand these tags MUST be able to decode bignums that have leading zeroes.

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

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

2.4.3. Decimal Fractions and Bigfloats

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

Bigfloats combine an integer mantissa with a base-2 scaling factor. They are binary floating-point values that can exceed the range or the precision of the three IEEE 754 formats supported by CBOR (Section 2.3). Bigfloats may also be used by constrained

applications that need some basic binary floating-point capability without the need for supporting IEEE 754.

A decimal fraction or a bigfloat is represented as a tagged array that contains exactly two integer numbers: an exponent e and a mantissa m . Decimal fractions (tag 4) use base-10 exponents; the value of a decimal fraction data item is $m \cdot (10^e)$. Bigfloats (tag 5) use base-2 exponents; the value of a bigfloat data item is $m \cdot (2^e)$. The exponent e MUST be represented in an integer of major type 0 or 1, while the mantissa also can be a bignum (Section 2.4.2).

An example of a decimal fraction is that the number 273.15 could be represented as 0b110_00100 (major type of 6 for the tag, additional information of 4 for the type of tag), followed by 0b100_00010 (major type of 4 for the array, additional information of 2 for the length of the array), followed by 0b001_00001 (major type of 1 for the first integer, additional information of 1 for the value of -2), followed by 0b000_11001 (major type of 0 for the second integer, additional information of 25 for a two-byte value), followed by 0b0110101010110011 (27315 in two bytes). In hexadecimal:

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

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

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

Decimal fractions and bigfloats provide no representation of Infinity, -Infinity, or NaN; if these are needed in place of a decimal fraction or bigfloat, the IEEE 754 half-precision representations from Section 2.3 can be used. For constrained applications, where there is a choice between representing a specific number as an integer and as a decimal fraction or bigfloat (such as when the exponent is small and non-negative), there is a quality-of-

implementation expectation that the integer representation is used directly.

2.4.4. Content Hints

The tags in this section are for content hints that might be used by generic CBOR processors.

2.4.4.1. Encoded CBOR Data Item

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

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

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

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

These three tag types suggest conversions to three of the base data encodings defined in [RFC4648]. For base64url encoding, padding is not used (see Section 3.2 of RFC 4648); that is, all trailing equals signs ("=") are removed from the base64url-encoded string. Later tags might be defined for other data encodings of RFC 4648 or for other ways to encode binary data in strings.

2.4.4.3. Encoded Text

Some text strings hold data that have formats widely used on the Internet, and sometimes those formats can be validated and presented to the application in appropriate form by the decoder. There are tags for some of these formats.

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

- o Tags 33 and 34 are for base64url- and base64-encoded text strings, as defined in [RFC4648];
- o Tag 35 is for regular expressions in Perl Compatible Regular Expressions (PCRE) / JavaScript syntax [ECMA262].
- o Tag 36 is for MIME messages (including all headers), as defined in [RFC2045];

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

2.4.5. Self-Describe CBOR

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

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

The serialization of this tag is 0xd9d9f7, which appears not to be in use as a distinguishing mark for frequently used file types. In particular, it is not a valid start of a Unicode text in any Unicode encoding if followed by a valid CBOR data item.

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

3. Creating CBOR-Based Protocols

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

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

This specification puts no restrictions on CBOR-based protocols. An encoder can be capable of encoding as many or as few types of values as is required by the protocol in which it is used; a decoder can be capable of understanding as many or as few types of values as is required by the protocols in which it is used. This lack of restrictions allows CBOR to be used in extremely constrained environments.

This section discusses some considerations in creating CBOR-based protocols. It is advisory only and explicitly excludes any language from RFC 2119 other than words that could be interpreted as "MAY" in the sense of RFC 2119.

3.1. CBOR in Streaming Applications

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

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

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

3.2. Generic Encoders and Decoders

A generic CBOR decoder can decode all well-formed CBOR data and present them to an application. CBOR data is well-formed if it uses the initial bytes, as well as the byte strings and/or data items that

are implied by their values, in the manner defined by CBOR, and no extraneous data follows (Appendix C).

Even though CBOR attempts to minimize these cases, not all well-formed CBOR data is valid: for example, the format excludes simple values below 32 that are encoded with an extension byte. Also, specific tags may make semantic constraints that may be violated, such as by including a tag in a bignum tag or by following a byte string within a date tag. Finally, the data may be invalid, such as invalid UTF-8 strings or date strings that do not conform to [RFC3339]. There is no requirement that generic encoders and decoders make unnatural choices for their application interface to enable the processing of invalid data. Generic encoders and decoders are expected to forward simple values and tags even if their specific codepoints are not registered at the time the encoder/decoder is written (Section 3.5).

Generic decoders provide ways to present well-formed CBOR values, both valid and invalid, to an application. The diagnostic notation (Section 6) may be used to present well-formed CBOR values to humans.

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

3.3. Syntax Errors

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

3.3.1. Incomplete CBOR Data Items

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

Examples of incomplete data items include:

- o A decoder expects a certain number of array or map entries but instead encounters the end of the data.

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

3.3.2. Malformed Indefinite-Length Items

Examples of malformed indefinite-length data items include:

- o Within an indefinite-length byte string or text, a decoder finds an item that is not of the appropriate major type before it finds the "break" stop code.
- o Within an indefinite-length map, a decoder encounters the "break" stop code immediately after reading a key (the value is missing).

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

3.3.3. Unknown Additional Information Values

At the time of writing, some additional information values are unassigned and reserved for future versions of this document (see Section 5.2). Since the overall syntax for these additional information values is not yet defined, a decoder that sees an additional information value that it does not understand cannot continue parsing.

3.4. Other Decoding Errors

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

Such problems might include:

Duplicate keys in a map: Generic decoders (Section 3.2) make data available to applications using the native CBOR data model. That

data model includes maps (key-value mappings with unique keys), not multimaps (key-value mappings where multiple entries can have the same key). Thus, a generic decoder that gets a CBOR map item that has duplicate keys will decode to a map with only one instance of that key, or it might stop processing altogether. On the other hand, a "streaming decoder" may not even be able to notice (Section 3.7).

Inadmissible type on the value following a tag: Tags (Section 2.4) specify what type of data item is supposed to follow the tag; for example, the tags for positive or negative bignums are supposed to be put on byte strings. A decoder that decodes the tagged data item into a native representation (a native big integer in this example) is expected to check the type of the data item being tagged. Even decoders that don't have such native representations available in their environment may perform the check on those tags known to them and react appropriately.

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

3.5. Handling Unknown Simple Values and Tags

A decoder that comes across a simple value (Section 2.3) that it does not recognize, such as a value that was added to the IANA registry after the decoder was deployed or a value that the decoder chose not to implement, might issue a warning, might stop processing altogether, might handle the error by making the unknown value available to the application as such (as is expected of generic decoders), or take some other type of action.

A decoder that comes across a tag (Section 2.4) that it does not recognize, such as a tag that was added to the IANA registry after the decoder was deployed or a tag that the decoder chose not to implement, might issue a warning, might stop processing altogether, might handle the error and present the unknown tag value together with the contained data item to the application (as is expected of generic decoders), might ignore the tag and simply present the contained data item only to the application, or take some other type of action.

3.6. Numbers

For the purposes of this specification, all number representations for the same numeric value are equivalent. This means that an encoder can encode a floating-point value of 0.0 as the integer 0. It, however, also means that an application that expects to find

integer values only might find floating-point values if the encoder decides these are desirable, such as when the floating-point value is more compact than a 64-bit integer.

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

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

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

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

3.7. Specifying Keys for Maps

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

If multiple types of keys are to be used, consideration should be given to how these types would be represented in the specific programming environments that are to be used. For example, in JavaScript objects, a key of integer 1 cannot be distinguished from a key of string "1". This means that, if integer keys are used, the simultaneous use of string keys that look like numbers needs to be avoided. Again, this leads to the conclusion that keys should be of a single CBOR type.

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

A CBOR-based protocol should make an intentional decision about what to do when a receiving application does see multiple identical keys in a map. The resulting rule in the protocol should respect the CBOR data model: it cannot prescribe a specific handling of the entries with the identical keys, except that it might have a rule that having identical keys in a map indicates a malformed map and that the decoder has to stop with an error. Duplicate keys are also prohibited by CBOR decoders that are using strict mode (Section 3.10).

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

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

3.8. Undefined Values

In some CBOR-based protocols, the simple value (Section 2.3) of Undefined might be used by an encoder as a substitute for a data item with an encoding problem, in order to allow the rest of the enclosing data items to be encoded without harm.

3.9. Canonical CBOR

Some protocols may want encoders to only emit CBOR in a particular canonical format; those protocols might also have the decoders check that their input is canonical. Those protocols are free to define what they mean by a canonical format and what encoders and decoders are expected to do. This section lists some suggestions for such protocols.

If a protocol considers "canonical" to mean that two encoder implementations starting with the same input data will produce the same CBOR output, the following four rules would suffice:

- o Integers must be as small as possible.
 - * 0 to 23 and -1 to -24 must be expressed in the same byte as the major type;
 - * 24 to 255 and -25 to -256 must be expressed only with an additional `uint8_t`;
 - * 256 to 65535 and -257 to -65536 must be expressed only with an additional `uint16_t`;
 - * 65536 to 4294967295 and -65537 to -4294967296 must be expressed only with an additional `uint32_t`.
- o The expression of lengths in major types 2 through 5 must be as short as possible. The rules for these lengths follow the above rule for integers.
- o The keys in every map must be sorted lowest value to highest. Sorting is performed on the bytes of the representation of the key data items without paying attention to the 3/5 bit splitting for major types. (Note that this rule allows maps that have keys of different types, even though that is probably a bad practice that could lead to errors in some canonicalization implementations.) The sorting rules are:
 - * If two keys have different lengths, the shorter one sorts earlier;
 - * If two keys have the same length, the one with the lower value in (byte-wise) lexical order sorts earlier.
- o Indefinite-length items must be made into definite-length items.

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

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

3.10. Strict Mode

Some areas of application of CBOR do not require canonicalization (Section 3.9) but may require that different decoders reach the same (semantically equivalent) results, even in the presence of potentially malicious data. This can be required if one application (such as a firewall or other protecting entity) makes a decision based on the data that another application, which independently decodes the data, relies on.

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

A decoder in strict mode will reliably reject any data that could be interpreted by other decoders in different ways. It will reliably reject data items with syntax errors (Section 3.3). It will also expend the effort to reliably detect other decoding errors (Section 3.4). In particular, a strict decoder needs to have an API that reports an error (and does not return data) for a CBOR data item that contains any of the following:

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

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

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

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

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

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

4. Converting Data between CBOR and JSON

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

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

4.1. Converting from CBOR to JSON

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

- o An integer (major type 0 or 1) becomes a JSON number.
- o A byte string (major type 2) that is not embedded in a tag that specifies a proposed encoding is encoded in base64url without padding and becomes a JSON string.
- o A UTF-8 string (major type 3) becomes a JSON string. Note that JSON requires escaping certain characters (RFC 7159, Section 7): quotation mark (U+0022), reverse solidus (U+005C), and the "C0 control characters" (U+0000 through U+001F). All other characters are copied unchanged into the JSON UTF-8 string.
- o An array (major type 4) becomes a JSON array.
- o A map (major type 5) becomes a JSON object. This is possible directly only if all keys are UTF-8 strings. A converter might also convert other keys into UTF-8 strings (such as by converting integers into strings containing their decimal representation); however, doing so introduces a danger of key collision.
- o False (major type 7, additional information 20) becomes a JSON false.
- o True (major type 7, additional information 21) becomes a JSON true.
- o Null (major type 7, additional information 22) becomes a JSON null.
- o A floating-point value (major type 7, additional information 25 through 27) becomes a JSON number if it is finite (that is, it can be represented in a JSON number); if the value is non-finite (NaN, or positive or negative Infinity), it is represented by the substitute value.
- o Any other simple value (major type 7, any additional information value not yet discussed) is represented by the substitute value.
- o A bignum (major type 6, tag value 2 or 3) is represented by encoding its byte string in base64url without padding and becomes

a JSON string. For tag value 3 (negative bignum), a "~" (ASCII tilde) is inserted before the base-encoded value. (The conversion to a binary blob instead of a number is to prevent a likely numeric overflow for the JSON decoder.)

- o A byte string with an encoding hint (major type 6, tag value 21 through 23) is encoded as described and becomes a JSON string.
- o For all other tags (major type 6, any other tag value), the embedded CBOR item is represented as a JSON value; the tag value is ignored.
- o Indefinite-length items are made definite before conversion.

4.2. Converting from JSON to CBOR

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

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

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

decimal representations (1.1, 1e9) in JSON. This may be hard to get right, and any ensuing vulnerabilities may be exploited by an attacker.

5. Future Evolution of CBOR

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

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

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

5.1. Extension Points

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

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

CBOR has three major extension points:

- o the "simple" space (values in major type 7). Of the 24 efficient (and 224 slightly less efficient) values, only a small number have been allocated. Implementations receiving an unknown simple data item may be able to process it as such, given that the structure of the value is indeed simple. The IANA registry in Section 7.1

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

- o the "tag" space (values in major type 6). Again, only a small part of the codepoint space has been allocated, and the space is abundant (although the early numbers are more efficient than the later ones). Implementations receiving an unknown tag can choose to simply ignore it or to process it as an unknown tag wrapping the following data item. The IANA registry in Section 7.2 is the appropriate way to address the extensibility of this codepoint space.
- o the "additional information" space. An implementation receiving an unknown additional information value has no way to continue parsing, so allocating codepoints to this space is a major step. There are also very few codepoints left.

5.2. Curating the Additional Information Space

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

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

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

6. Diagnostic Notation

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

Note that this truly is a diagnostic format; it is not meant to be parsed. Therefore, no formal definition (as in ABNF) is given in this document. (Implementers looking for a text-based format for

representing CBOR data items in configuration files may also want to consider YAML [YAML].)

The diagnostic notation is loosely based on JSON as it is defined in RFC 7159, extending it where needed.

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

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

or the equivalent relative time as

```
1(1363896240)
```

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

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

6.1. Encoding Indicators

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

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

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

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

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

7. IANA Considerations

IANA has created two registries for new CBOR values. The registries are separate, that is, not under an umbrella registry, and follow the rules in [RFC5226]. IANA has also assigned a new MIME media type and an associated Constrained Application Protocol (CoAP) Content-Format entry.

7.1. Simple Values Registry

IANA has created the "Concise Binary Object Representation (CBOR) Simple Values" registry. The initial values are shown in Table 2.

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

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

7.2. Tags Registry

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

New entries in the range 0 to 23 are assigned by Standards Action. New entries in the range 24 to 255 are assigned by Specification Required. New entries in the range 256 to 18446744073709551615 are

assigned by First Come First Served. The template for registration requests is:

- o Data item
- o Semantics (short form)

In addition, First Come First Served requests should include:

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

7.3. Media Type ("MIME Type")

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

Type name: application

Subtype name: cbor

Required parameters: n/a

Optional parameters: n/a

Encoding considerations: binary

Security considerations: See Section 8 of this document

Interoperability considerations: n/a

Published specification: This document

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

Additional information:

Magic number(s): n/a

File extension(s): .cbor

Macintosh file type code(s): n/a

Person & email address to contact for further information:

Carsten Bormann

cabo@tzi.org

Intended usage: COMMON

Restrictions on usage: none

Author:

Carsten Bormann <cabo@tzi.org>

Change controller:

The IESG <iesg@ietf.org>

7.4. CoAP Content-Format

Media Type: application/cbor

Encoding: -

Id: 60

Reference: [RFCthis]

7.5. The +cbor Structured Syntax Suffix Registration

Name: Concise Binary Object Representation (CBOR)

+suffix: +cbor

References: [RFCthis]

Encoding Considerations: CBOR is a binary format.

Interoperability Considerations: n/a

Fragment Identifier Considerations:

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

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

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

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

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

Security Considerations: See Section 8 of this document

Contact:

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

Author/Change Controller:

The Apps Area Working Group.

The IESG has change control over this registration.

8. Security Considerations

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

Resource exhaustion attacks might attempt to lure a decoder into allocating very big data items (strings, arrays, maps) or exhaust the stack depth by setting up deeply nested items. Decoders need to have appropriate resource management to mitigate these attacks. (Items for which very large sizes are given can also attempt to exploit integer overflow vulnerabilities.)

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

possible. For example, an attacker could make use of duplicate keys in maps and precision issues in numbers to make the gatekeeper base its decisions on a different interpretation than the one that will be used by the second application. Protocols that are used in a security context should be defined in such a way that these multiple interpretations are reliably reduced to a single one. To facilitate this, encoder and decoder implementations used in such contexts should provide at least one strict mode of operation (Section 3.10).

9. Acknowledgements

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

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

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

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

10. References

10.1. Normative References

- [ECMA262] European Computer Manufacturers Association, "ECMAScript Language Specification 5.1 Edition", ECMA Standard ECMA-262, June 2011, <<http://www.ecma-international.org/publications/files/ecma-st/ECMA-262.pdf>>.
- [RFC2045] Freed, N. and N. Borenstein, "Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies", RFC 2045, DOI 10.17487/RFC2045, November 1996, <<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>>.
- [RFC3339] Klyne, G. and C. Newman, "Date and Time on the Internet: Timestamps", RFC 3339, DOI 10.17487/RFC3339, July 2002, <<http://www.rfc-editor.org/info/rfc3339>>.
- [RFC3629] Yergeau, F., "UTF-8, a transformation format of ISO 10646", STD 63, RFC 3629, DOI 10.17487/RFC3629, November 2003, <<http://www.rfc-editor.org/info/rfc3629>>.
- [RFC3986] Berners-Lee, T., Fielding, R., and L. Masinter, "Uniform Resource Identifier (URI): Generic Syntax", STD 66, RFC 3986, DOI 10.17487/RFC3986, January 2005, <<http://www.rfc-editor.org/info/rfc3986>>.
- [RFC4287] Nottingham, M., Ed. and R. Sayre, Ed., "The Atom Syndication Format", RFC 4287, DOI 10.17487/RFC4287, December 2005, <<http://www.rfc-editor.org/info/rfc4287>>.
- [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>>.
- [RFC5226] Narten, T. and H. Alvestrand, "Guidelines for Writing an IANA Considerations Section in RFCs", BCP 26, RFC 5226, DOI 10.17487/RFC5226, May 2008, <<http://www.rfc-editor.org/info/rfc5226>>.
- [TIME_T] The Open Group Base Specifications, "Vol. 1: Base Definitions, Issue 7", Section 4.15 'Seconds Since the Epoch', IEEE Std 1003.1, 2013 Edition, 2013, <http://pubs.opengroup.org/onlinepubs/9699919799/basedefs/V1_chap04.html#tag_04_15>.

10.2. Informative References

- [ASN.1] International Telecommunication Union, "Information Technology -- ASN.1 encoding rules: Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER) and Distinguished Encoding Rules (DER)", ITU-T Recommendation X.690, 1994.
- [BSON] Various, "BSON - Binary JSON", 2013, <<http://bsonspec.org/>>.

- [MessagePack] Furuhashi, S., "MessagePack", 2013, <<http://msgpack.org/>>.
- [RFC0713] Haverty, J., "MSDTP-Message Services Data Transmission Protocol", RFC 713, DOI 10.17487/RFC0713, April 1976, <<http://www.rfc-editor.org/info/rfc713>>.
- [RFC6838] Freed, N., Klensin, J., and T. Hansen, "Media Type Specifications and Registration Procedures", BCP 13, RFC 6838, DOI 10.17487/RFC6838, January 2013, <<http://www.rfc-editor.org/info/rfc6838>>.
- [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>>.
- [RFC7228] Bormann, C., Ersue, M., and A. Keranen, "Terminology for Constrained-Node Networks", RFC 7228, DOI 10.17487/RFC7228, May 2014, <<http://www.rfc-editor.org/info/rfc7228>>.
- [UBJSON] The Buzz Media, "Universal Binary JSON Specification", 2013, <<http://ubjson.org/>>.
- [YAML] Ben-Kiki, O., Evans, C., and I. Net, "YAML Ain't Markup Language (YAML[TM]) Version 1.2", 3rd Edition, October 2009, <<http://www.yaml.org/spec/1.2/spec.html>>.

Appendix A. Examples

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

Diagnostic	Encoded
0	0x00
1	0x01
10	0x0a
23	0x17
24	0x1818
25	0x1819
100	0x1864
1000	0x1903e8
1000000	0x1a000f4240
10000000000000	0x1b000000e8d4a51000
18446744073709551615	0x1bffffffffffffffffffff
18446744073709551616	0xc249010000000000000000
-18446744073709551616	0x3bffffffffffffffffffff

-18446744073709551617	0xc34901000000000000000000
-1	0x20
-10	0x29
-100	0x3863
-1000	0x3903e7
0.0	0xf90000
-0.0	0xf98000
1.0	0xf93c00
1.1	0xfb3fff1999999999999a
1.5	0xf93e00
65504.0	0xf97bff
100000.0	0xfa47c35000
3.4028234663852886e+38	0xfa7f7fffff
1.0e+300	0xfb7e37e43c8800759c
5.960464477539063e-8	0xf90001
0.00006103515625	0xf90400
-4.0	0xf9c400
-4.1	0xfbc01066666666666666
Infinity	0xf97c00
NaN	0xf97e00
-Infinity	0xf9fc00
Infinity	0xfa7f800000
NaN	0xfa7fc00000
-Infinity	0xfaff800000

Infinity	0xfb7ff0000000000000
NaN	0xfb7ff8000000000000
-Infinity	0xfbfff0000000000000
false	0xf4
true	0xf5
null	0xf6
undefined	0xf7
simple(16)	0xf0
simple(24)	0xf818
simple(255)	0xf8ff
0("2013-03-21T20:04:00Z")	0xc074323031332d30332d32315432303a30343a30305a
1(1363896240)	0xc11a514b67b0
1(1363896240.5)	0xc1fb41d452d9ec200000
23(h'01020304')	0xd74401020304
24(h'6449455446')	0xd818456449455446
32("http://www.example.com")	0xd82076687474703a2f2f7777772e6578616d706c652e636f6d
h''	0x40
h'01020304'	0x4401020304
" "	0x60
"a"	0x6161
"IETF"	0x6449455446
"\"\\\""	0x62225c
"\u00fc"	0x62c3bc

"\u6c34"	0x63e6b0b4
"\ud800\udd51"	0x64f0908591
[]	0x80
[1, 2, 3]	0x83010203
[1, [2, 3], [4, 5]]	0x8301820203820405
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25]	0x98190102030405060708090a0b0c0d0e0f101112131415161718181819
{}	0xa0
{1: 2, 3: 4}	0xa201020304
{"a": 1, "b": [2, 3]}	0xa26161016162820203
["a", {"b": "c"}]	0x826161a161626163
{"a": "A", "b": "B", "c": "C", "d": "D", "e": "E"}	0xa56161614161626142616361436164614461656145
(_ h'0102', h'030405')	0x5f42010243030405ff
(_ "strea", "ming")	0x7f657374726561646d696e67ff
[_]	0x9fff
[_ 1, [2, 3], [_ 4, 5]]	0x9f018202039f0405ffff
[_ 1, [2, 3], [4, 5]]	0x9f01820203820405ff
[1, [2, 3], [_ 4, 5]]	0x83018202039f0405ff
[1, [_ 2, 3], [4, 5]]	0x83019f0203ff820405
[_ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25]	0x9f0102030405060708090a0b0c0d0e0f101112131415161718181819ff
{_ "a": 1, "b": [_ 2, 3]}	0xbf61610161629f0203ffff
["a", {_ "b": "c"}]	0x826161bf61626163ff

{_ "Fun": true, "Amt": -2}	0xbf6346756ef563416d7421ff
-----	-----

Table 4: Examples of Encoded CBOR Data Items

Appendix B. Jump Table

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

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

0x5a	byte string (four-byte uint32_t for n, and then n bytes follow)
0x5b	byte string (eight-byte uint64_t for n, and then n bytes follow)
0x5f	byte string, byte strings follow, terminated by "break"
0x60..0x77	UTF-8 string (0x00..0x17 bytes follow)
0x78	UTF-8 string (one-byte uint8_t for n, and then n bytes follow)
0x79	UTF-8 string (two-byte uint16_t for n, and then n bytes follow)
0x7a	UTF-8 string (four-byte uint32_t for n, and then n bytes follow)
0x7b	UTF-8 string (eight-byte uint64_t for n, and then n bytes follow)
0x7f	UTF-8 string, UTF-8 strings follow, terminated by "break"
0x80..0x97	array (0x00..0x17 data items follow)
0x98	array (one-byte uint8_t for n, and then n data items follow)
0x99	array (two-byte uint16_t for n, and then n data items follow)
0x9a	array (four-byte uint32_t for n, and then n data items follow)
0x9b	array (eight-byte uint64_t for n, and then n data items follow)
0x9f	array, data items follow, terminated by "break"
0xa0..0xb7	map (0x00..0x17 pairs of data items follow)
0xb8	map (one-byte uint8_t for n, and then n pairs of data items follow)
0xb9	map (two-byte uint16_t for n, and then n pairs of

	data items follow)
0xba	map (four-byte uint32_t for n, and then n pairs of data items follow)
0xbb	map (eight-byte uint64_t for n, and then n pairs of data items follow)
0xbf	map, pairs of data items follow, terminated by "break"
0xc0	Text-based date/time (data item follows; see Section 2.4.1)
0xc1	Epoch-based date/time (data item follows; see Section 2.4.1)
0xc2	Positive bignum (data item "byte string" follows)
0xc3	Negative bignum (data item "byte string" follows)
0xc4	Decimal Fraction (data item "array" follows; see Section 2.4.3)
0xc5	Bigfloat (data item "array" follows; see Section 2.4.3)
0xc6..0xd4	(tagged item)
0xd5..0xd7	Expected Conversion (data item follows; see Section 2.4.4.2)
0xd8..0xdb	(more tagged items, 1/2/4/8 bytes and then a data item follow)
0xe0..0xf3	(simple value)
0xf4	False
0xf5	True
0xf6	Null
0xf7	Undefined
0xf8	(simple value, one byte follows)
0xf9	Half-Precision Float (two-byte IEEE 754)

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

Table 5: Jump Table for Initial Byte

Appendix C. Pseudocode

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

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

The pseudocode has the following prerequisites:

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

```

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

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

```

Figure 1: Pseudocode for Well-Formedness Check

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

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

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

Figure 2: Pseudocode for Encoding a Signed Integer

Appendix D. Half-Precision

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

```
#include <math.h>

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

Figure 3: C Code for a Half-Precision Decoder

```
import struct
from math import ldexp

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

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

Figure 4: Python Code for a Half-Precision Decoder

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

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

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

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

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

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

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

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

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

E.2. MessagePack

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

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

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

E.3. BSON

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

E.4. UBJSON

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

E.5. MSDTP: RFC 713

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

E.6. Conciseness on the Wire

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

Format	[1, [2, 3]]	[_ 1, [2, 3]]
RFC 713	c2 05 81 c2 02 82 83	
ASN.1 BER	30 0b 02 01 01 30 06 02 01 02 02 01 03	30 80 02 01 01 30 06 02 01 02 02 01 03 00 00
MessagePack	92 01 92 02 03	
BSON	22 00 00 00 10 30 00 01 00 00 00 04 31 00 13 00 00 00 10 30 00 02 00 00 00 10 31 00 03 00 00 00 00 00	
UBJSON	61 02 42 01 61 02 42 02 42 03	61 ff 42 01 61 02 42 02 42 03 45
CBOR	82 01 82 02 03	9f 01 82 02 03 ff

Table 6: Examples for Different Levels of Conciseness

Authors' Addresses

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

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

Paul Hoffman
 ICANN

EMail: paul.hoffman@icann.org

Network Working Group
Internet-Draft
Intended status: Informational
Expires: August 8, 2017

J. Roatch
C. Bormann
Universitaet Bremen TZI
February 04, 2017

Concise Binary Object Representation (CBOR) Tags for Typed Arrays
draft-jroatch-cbor-tags-05

Abstract

The Concise Binary Object Representation (CBOR, RFC 7049) is a data format whose design goals include the possibility of extremely small code size, fairly small message size, and extensibility without the need for version negotiation.

The present document makes use of this extensibility to define a number of CBOR tags for typed arrays of numeric data, as well as two additional tags for multi-dimensional and homogeneous arrays. It is intended as the reference document for the IANA registration of the CBOR tags defined.

Status of This Memo

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

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at <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 August 8, 2017.

Copyright Notice

Copyright (c) 2017 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<http://trustee.ietf.org/license-info>) in effect on the date of

publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

Table of Contents

1. Introduction	2
1.1. Terminology	3
2. Typed Arrays	3
2.1. Types of numbers	3
3. Additional Array Tags	4
3.1. Multi-dimensional Array	5
3.2. Homogeneous Array	5
4. Discussion	6
5. CDDL typenames	7
6. IANA Considerations	8
7. Security Considerations	9
8. References	10
8.1. Normative References	10
8.2. Informative References	10
Contributors	10
Acknowledgements	10
Authors' Addresses	11

1. Introduction

The Concise Binary Object Representation (CBOR, [RFC7049]) provides for the interchange of structured data without a requirement for a pre-agreed schema. RFC 7049 defines a basic set of data types, as well as a tagging mechanism that enables extending the set of data types supported via an IANA registry.

Recently, a simple form of typed arrays of numeric data have received interest both in the Web graphics community [TypedArray] and in the JavaScript specification [TypedArrayES6], as well as in corresponding implementations [ArrayBuffer].

Since these typed arrays may carry significant amounts of data, there is interest in interchanging them in CBOR without the need of lengthy conversion of each number in the array.

This document defines a number of interrelated CBOR tags that cover these typed arrays, as well as two additional tags for multi-dimensional and homogeneous arrays. It is intended as the reference document for the IANA registration of the tags defined.

1.1. Terminology

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

2. Typed Arrays

Typed arrays are homogeneous arrays of numbers, all of which are encoded in a single form of binary representation. The concatenation of these representations is encoded as a single CBOR byte string (major type 2), enclosed by a single tag indicating the type and encoding of all the numbers represented in the byte string.

2.1. Types of numbers

Three classes of numbers are of interest: unsigned integers (uint), signed integers (twos' complement, sint), and IEEE 754 binary floating point numbers (which are always signed). For each of these classes, there are multiple representation lengths in active use:

Length	uint	sint	float
0	uint8	sint8	binary16
1	uint16	sint16	binary32
2	uint32	sint32	binary64
3	uint64	sint64	binary128

Table 1: Length values

Here, sintN stands for a signed integer of exactly N bits (for instance, sint16), and uintN stands for an unsigned integer of exactly N bits (for instance, uint32). The name binaryN stands for the number form of the same name defined in IEEE 754.

Since one objective of these tags is to be able to directly ship the ArrayBuffers underlying the Typed Arrays without re-encoding them, and these may be either in big endian (network byte order) or in little endian form, we need to define tags for both variants.

In total, this leads to 24 variants. In the tag, we need to express the choice between integer and floating point, the signedness (for integers), the endianness, and one of the four length values.

In order to simplify implementation, a range of tags is being allocated that allows retrieving all this information from the bits of the tag: Tag values from 64 to 87 (0x40 to 0x57).

The value is split up into 5 bit fields: 0b010_f_s_e_ll, as detailed in Table 2.

Field	Use
0b010	a constant '010'
f	0 for integer, 1 for float
s	0 for unsigned integer or float, 1 for signed integer
e	0 for big endian, 1 for little endian
ll	A number for the length (Table 1).

Table 2: Bit fields in the low 8 bits of the tag

The number of bytes in each array element can then be calculated by "2**(f + ll)" (or "1 << (f + ll)" in a typical programming language). (Notice that f and ll are the lsb of each nibble (4bit) in the byte.)

In the CBOR representation, the total number of elements in the array is not expressed explicitly, but implied from the length of the byte string and the length of each representation. It can be computed inversely to the previous formula: "bytelenlength >> (f + ll)".

For the uint8/sint8 values, the endianness is redundant. Only the big endian variant is used. As a special case, what would be the little endian variant of uint8 is used to signify that the numbers in the array are using clamped conversion from integers, as described in more detail in Section 7.1 of [TypedArrayUpdate].

3. Additional Array Tags

This specification defines two additional array tags. The Multi-dimensional Array tag can be combined with classical CBOR arrays as well as with Typed Arrays in order to build multi-dimensional arrays with constant numbers of elements in the sub-arrays. The Homogeneous Array tag can be used to facilitate the ingestion of homogeneous classical CBOR arrays, providing performance advantages even when a Typed Array does not apply.

3.1. Multi-dimensional Array

Tag: TBD40

Data Item: array (major type 4) of two arrays, one array (major type 4) of dimensions, and one array (major type 4, a Typed Array, or a Homogeneous Array) of elements

A multi-dimensional array is represented as a tagged array that contains two (one-dimensional) arrays. The first array defines the dimensions of the multi-dimensional array (in the sequence of outer dimensions towards inner dimensions) while the second array represents the contents of the multi-dimensional array. If the second array is itself tagged as a Typed Array then the element type of the multi-dimensional array is known to be the same type as that of the Typed Array. Data in the Typed Array byte string consists of consecutive values where the last dimension is considered contiguous (row-major order).

```
uint16_t a[2][3] = {
    {0, 1, 2}, /* row 0 */
    {3, 4, 5},
};
```

```
<Tag TBD40> # multi-dimensional array tag
82          # array(2)
82          # array(2)
02          # unsigned(2) 1st Dimension
03          # unsigned(3) 2nd Dimension
d8 41      # uint16 array
4a          # byte string(12)
00 00      # unsigned(0)
00 01      # unsigned(1)
00 02      # unsigned(2)
00 03      # unsigned(3)
00 04      # unsigned(4)
00 05      # unsigned(5)
```

Figure 1: Multi-dimensional array in C and CBOR

3.2. Homogeneous Array

Tag: TBD41

Data Item: array (major type 4)

This tag provides a hint to decoders that the array tagged by it has elements that are all of the same application type. The element type

of the array is thus determined by the application type of the first array element. This can be used by implementations in strongly typed languages while decoding to create native homogeneous arrays of specific types instead of ordered lists.

Which CBOR data items constitute elements of the same application type is specific to the application. However, type systems of programming languages have enough commonality that an application should be able to create portable homogeneous arrays.

```
bool boolArray[2] = { true, false };
```

```
<Tag TBD41>  # Homogeneous Array Tag
      82      #array(2)
      F5      # true
      F4      # false
```

Figure 2: Homogeneous array in C and CBOR

4. Discussion

Support for both little- and big-endian representation may seem out of character with CBOR, which is otherwise fully big endian. This support is in line with the intended use of the typed arrays and the objective not to require conversion of each array element.

This specification allocates a sizable chunk out of the single-byte tag space. This use of code point space is justified by the wide use of typed arrays in data interchange.

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

5. CDDL typenames

For the use with CDDL [I-D.greevenbosch-appsawg-cbor-cddl], the typenames defined in Figure 3 are recommended:

```
ta-uint8 = #6.64(bstr)
ta-uint16be = #6.65(bstr)
ta-uint32be = #6.66(bstr)
ta-uint64be = #6.67(bstr)
ta-uint8-clamped = #6.68(bstr)
ta-uint16le = #6.69(bstr)
ta-uint32le = #6.70(bstr)
ta-uint64le = #6.71(bstr)
ta-sint8 = #6.72(bstr)
ta-sint16be = #6.73(bstr)
ta-sint32be = #6.74(bstr)
ta-sint64be = #6.75(bstr)
; reserved: #6.76(bstr)
ta-sint16le = #6.77(bstr)
ta-sint32le = #6.78(bstr)
ta-sint64le = #6.79(bstr)
ta-float16be = #6.80(bstr)
ta-float32be = #6.81(bstr)
ta-float64be = #6.82(bstr)
ta-float128be = #6.83(bstr)
ta-float16le = #6.84(bstr)
ta-float32le = #6.85(bstr)
ta-float64le = #6.86(bstr)
ta-float128le = #6.87(bstr)
homogeneous<array> = #6.TBD41(array)
multi-dim<dim, array> = #6.TBD40([dim, array])
```

Figure 3: Recommended typenames for CDDL

6. IANA Considerations

IANA is requested to allocate the tags in Table 3, with the present document as the specification reference.

Tag	Data Item	Semantics
64	byte string	uint8 Typed Array
65	byte string	uint16, big endian, Typed Array
66	byte string	uint32, big endian, Typed Array
67	byte string	uint64, big endian, Typed Array
68	byte string	uint8 Typed Array, clamped arithmetic
69	byte string	uint16, little endian, Typed Array
70	byte string	uint32, little endian, Typed Array
71	byte string	uint64, little endian, Typed Array
72	byte string	sint8 Typed Array
73	byte string	sint16, big endian, Typed Array
74	byte string	sint32, big endian, Typed Array
75	byte string	sint64, big endian, Typed Array
76	byte string	(reserved)
77	byte string	sint16, little endian, Typed Array
78	byte string	sint32, little endian, Typed Array
79	byte string	sint64, little endian, Typed Array
80	byte string	IEEE 754 binary16, big endian, Typed Array
81	byte string	IEEE 754 binary32, big endian, Typed Array
82	byte string	IEEE 754 binary64, big endian, Typed Array
83	byte string	IEEE 754 binary128, big endian, Typed Array
84	byte string	IEEE 754 binary16, little endian, Typed Array
85	byte string	IEEE 754 binary32, little endian, Typed Array
86	byte string	IEEE 754 binary64, little endian, Typed Array
87	byte string	IEEE 754 binary128, little endian, Typed Array
TBD40	array of two arrays*	Multi-dimensional Array
TBD41	array	Homogeneous Array

Table 3: Values for Tags

*) TBD40 data item: second element of outer array in data item is native CBOR array (major type 4) or Typed Array (one of Tag 64..87)

RFC editor note: Please replace TBD40 and TBD41 by the tag numbers allocated by IANA throughout the document and delete this note.

7. Security Considerations

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

8. References

8.1. Normative References

- [I-D.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.
- [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>>.

8.2. Informative References

- [ArrayBuffer]
Mozilla Developer Network, "JavaScript typed arrays", 2013, <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Typed_arrays>.
- [TypedArray]
Vukicevic, V. and K. Russell, "Typed Array Specification", February 2011, <<https://www.khronos.org/registry/typedarray/specs/1.0/>>.
- [TypedArrayES6]
"22.2 TypedArray Objects", in: ECMA-262 6th Edition, The ECMAScript 2015 Language Specification, June 2015, <<http://www.ecma-international.org/ecma-262/6.0/#sec-typedarray-objects>>.
- [TypedArrayUpdate]
Herman, D. and K. Russell, "Typed Array Specification", July 2013, <<https://www.khronos.org/registry/typedarray/specs/latest/>>.

Contributors

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

Acknowledgements

TBD

Authors' Addresses

Johnathan Roatch

Email: jroatch@gmail.com

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

Phone: +49-421-218-63921

Email: cabo@tzi.org